

FieldView Reference Manual

**SOFTWARE RELEASE
VERSION 2023**

11/7/2023

Tecplot, Inc.

Copyright ©2023 Tecplot, Inc.
Revision 2023
All Rights Reserved
Printed in USA
First Printing November 2001

FieldView is a registered trademark of Tecplot, Inc.

Tecplot, Inc. reserves the right to make changes in specifications and other information contained in this publication without prior notice and the reader should in all cases consult Tecplot, Inc. to determine whether any such changes have been made.

The terms and conditions governing the licensing of Tecplot, Inc. software consist solely of those set forth in the Tecplot, Inc. Binary License Terms and Conditions set forth by Tecplot, Inc.. No representations or other affirmation of fact contained in this publication, including but not limited to statements regarding capacity, response-time performance, suitability for use or performance of products described herein shall be deemed to be a warranty of Tecplot, Inc. for any purpose, or give rise to any liability by Tecplot, Inc. whatsoever.

In no event shall Tecplot, Inc. be liable for any incidental, indirect, special or consequential damages whatsoever (including but not limited to lost profits) arising out of or relating to this publication or the information contained in it, even if Tecplot, Inc. has been advised, knew or should have known of the possibility of such damages.

The software programs described in this document are confidential information and proprietary products of Tecplot, Inc. or its licensors.

Reprise License Manager (RLM) v12.4, Copyright (C) 2006-2018, Reprise Software, Inc.
Copyright (C) 2006-2018, Reprise Software, Inc. All rights reserved.
RLM Build 12.4.2

Reprise License Manager is a registered trademark or trademark of Reprise Software, Inc. in the U.S. and/or other countries.
Copyright (c) 2000 - 2021, Lawrence Livermore National Security, LLC
All rights reserved.

Copyright © 2002 GraphicsMagick Group, an organization dedicated to making software imaging solutions freely available.

Copyright © 1999 E. I. du Pont de Nemours and Company

Copyright © 2000-2002, Ghostgum Software Pty Ltd. All rights reserved.

Copyright © 2000 Markus Friedl. All rights reserved.

Copyright © 1999 - 2003 Bob Friesenhahn <bfriesen@simple.dallas.tx.us>

FIG: Facility for Interactive Generation of figures Copyright © 1985-1988 by Supoj Sutanthavibul Parts Copyright © 1989-2000 by Brian V.

Smith Parts Copyright © 1991 by Paul King

The Graphics Interchange Format © is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated.

Copyright © 1994 - 2000 by the Massachusetts Institute of Technology. All rights reserved.

Copyright © 1994-2000 TeCGraf, PUC-Rio. All rights reserved.

Copyright © 1991-2000 by Bell Labs Innovations for Lucent Technologies.

Copyright © 1998-2008 The OpenSSL Project. All rights reserved.

Copyright © 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved.

Copyright © 2002 Niels Provos <provos@citi.umich.edu> All rights reserved.

This product uses parts of the SFL package, Copyright © 1996-2000 iMatix Corporation 1991-2000 iMatix Corporation <<http://www.imatix.com>>.

Copyright © 1995-2002 Jean-loup Gailly and Mark Adler

Copyright © 1993 University of Chicago

Copyright © 1993 Mississippi State University

Copyright © 1999 Serika Kurusugawa. All rights reserved.

Copyright © 1999-2000 Mizi Research Inc. All rights reserved.

Copyright © 2001, 2002 Turbolinux, Inc. Written by James Su.

Copyright © 2000 Turbolinux, Inc. Written by Justin Yu and Sean Chen.

Copyright © 2000 Ming-Che Chuang

Copyright © 2002 WU Yi, HancornLinux Inc.

Copyright © 2000 Ming-Che Chuang

Copyright © 2001, 2002 ThizLinux Laboratory Ltd. Written by Anthony Fok.

Copyright © 2003-2004 immodule for Qt Project. All rights reserved.

Copyright © 2003-2006 Ben van Klinken and the CLucene Team

Copyright © 2008 Nokia Corporation and/or its subsidiary(-ies)

Copyright © 2004, 2005 Daniel M. Duley

Copyright © 2005 Bjoern Bergstroem

Copyright © 2005 Roberto Raggi

Copyright © The Internet Society (2001). All Rights Reserved.

Copyright © 1991 by AT&T.

Copyright © 2000 Hans Petter Bieker. All rights reserved.

Copyright © 1996 Daniel Dardailler.
 Copyright © 2002 USC/Information Sciences Institute
 Copyright © 2005-2007 Matthias Kretz <kretz@kde.org>
 Copyright © 1998 by Bjorn Reese <brees@imada.ou.dk>
 Copyright © 2000-2007 Gerard Juyn (gerard@libmng.com)
 Copyright © 2004, 2006-2008 Glenn Randers-Pehrson
 Copyright © 1996, 1997 Andreas Dilger
 Copyright © 1995, 1996 Guy Eric Schalnat, Group 42, Inc.
 Copyright © 2002 Robert Osfield.
 Copyright © 1998 Julian Smart, Robert Roebling [, ...]
 Copyright © 1999-2007 Brian Paul All Rights Reserved.
 Copyright © 2012 The FreeBSD Foundation. All rights reserved.
 Copyright © 2003-2018 University of Illinois at Urbana-Champaign. All rights reserved.
 Copyright 1992, 1993, 1994 Henry Spencer. All rights reserved.
 Copyright © 1989, 1991, 1993, 1994 The Regents of the University of California. All rights reserved.
 Copyright © 2009-2012, 2016 Daniel Stone
 Copyright © 2012 Ran Benita <ran234@gmail.com>
 Copyright © 2010, 2012 Intel Corporation
 Copyright © 2008, 2009 Dan Nicholson
 Copyright © 2010 Francisco Jerez <currojerez@riseup.net>
 Copyright © 1991-2000 Silicon Graphics, Inc. All Rights Reserved.
 Copyright 1987, 1988 by Digital Equipment Corporation, Maynard, Massachusetts.
 Copyright © 2011 Joseph Adams <joejadams3.14159@gmail.com>
 Copyright 1996 by Joseph Moss
 Copyright © Dmitry Golubev <lastguru@mail.ru>, 2003-2004
 Copyright © 2004, Gregory Mokhin <mokhin@bog.msu.ru>
 Copyright © 2006 Erdal Ronahi
 Copyright © 1998-2007 Free Software Foundation, Inc.
 Copyright © 2005, 2006, 2010 Free Software Foundation, Inc.
 Copyright © 1991-2010, Thomas G. Lane, Guido Vollbeding.
 This software is based in part on the work of the Independent JPEG Group.
 Copyright © 1988-1997 Sam Leffler
 Copyright © 1985, 1986, 1987, 1992 X Consortium
 Copyright 1996-2002 by David Turner, Robert Wilhelm, and Werner Lemberg
 Copyright © 2003 The XFree86 Project, Inc. All Rights Reserved
 Copyright © 1992 by Oki Technosystems Laboratory, Inc.
 Copyright © 1992 by Fuji Xerox Co., Ltd
 Copyright © 1995-2009 International Business Machines Corporation and others
 Copyright © 2001-2006 Bart Massey, Jamey Sharp, and Josh Triplett.
 Copyright © 1985, 1987-1988, 1990, 1993, 1994, 1998 The Open Group

This distribution contains PDF3D software copyright Visual Technology Services Ltd., all rights reserved.

This software is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., Scriptics Corporation, and other parties.

Development tools and related technology provided under license from

FieldView includes Open MPI, from the Open MPI Project (See README_Copyrights.txt)

Copyright © 2011-2017, Christopher C. Hulbert. All rights reserved.

FieldView includes the font Noto Sans CJK JP from the Google Noto Fonts (See README_Copyrights.txt)

Copyright © 2003-2017, Troy D. Hanson <http://troydhanson.github.com/uthash/>. All rights reserved.

VTK Reader: Copyright © 1993-2015 Ken Martin, Will Schroeder, Bill Lorensen. All rights reserved.

MAT-File Format © COPYRIGHT 1999-2017 by The MathWorks, Inc.

OSF, OSF/Motif[®], and Motif[®] are registered trademarks of The Open Software Foundation, Inc.

OpenGL is a trademark of Silicon Graphics, Inc. in the United States and other countries.

IBM is a registered trademark of International Business Machines Corporation.

Silicon Graphics is a registered trademark of Silicon Graphics, Inc.

Sun Microsystems is a registered trademark of Sun Microsystems, Inc.

UNIX is a registered trademark of AT&T.

PostScript is a registered trademark of Adobe Systems, Inc.

FIDAP, FLUENT and RAMPANT are registered trademarks of Ansys, Inc.

Phoenix is a registered trademark of Cham, Limited.

STAR-CD is a trademark of Computational Dynamics Ltd.

FLOW-3D[®] is a registered trademark of Flow Science, Inc.

CFX and CFX-TASCflow are registered trademarks of ANSYS-CFX.

MATLAB is a registered trademark of The MathWorks, Inc.

Noto is a trademark of Google Inc.

U.S. GOVERNMENT RESTRICTED RIGHTS. The Licensed Software is deemed to be commercial computer software as defined in FAR 12.212 and subject to restricted rights as defined in FAR Section 52.227-19 "Commercial Computer Software License", as applicable, and any successor regulations. Any use, modification, reproduction release, performance, display or disclosure of the Licensed Software by the U.S. Government shall be solely in accordance with the terms of the Tecplot, Inc. Binary License Terms and Conditions.

THIS DOCUMENT AND ALL INFORMATION CONTAINED HEREIN, ALONG WITH ALL FILE FORMATS, EXAMPLE DATA AND ONLINE DOCUMENTATION, IS INTENDED FOR THE EXCLUSIVE USE OF END-USER LICENSORS OF FieldView.

Table of Contents

List of Figures	xiii
------------------------------	-------------

Chapter 1

Data Files.....	1
Overview on Reading Data in FieldView	4
GRID PROCESSING	4
Function Selection	5
Transient Data	6
PLOT3D & OVERFLOW-2 Auto-Detect Format	8
Read Boundary Data Only	9
Reading more than one Dataset at a time	11
Dataset Comparison for Multiple Datasets	11
Merged Transient Datasets	12
Appending Datasets to the same Server Process	12
Data Written On Different Systems	12
Support for Arbitrary Elements	13
FieldView Parallel Datasets	18
Direct Readers in FieldView	20
Working with the Data Input Menu	21
Reading Data Interactively with FieldView Parallel	21
AcuSolve Direct Reader	22
CGNS	25
CGNS Unstructured/Hybrid Reader	26
FIDAP	27
FLOW-3D® Animation Data	28
FLOW-3D® Restart Data	28
FLOW-3D®	28
ANSYS-Fluent CFF [Direct Reader]	30
FLUENT cas/dat Direct Reader	33
FLUENT Direct Reader	35
FLUENT Universal	36
FLUENT/UNS (and RAMPANT)	36
FV-UNS Data Input (Native FieldView Unstructured Format)	36
Ensight Reader	38
Tecplot 360 Reader	40
HAVOC	41
LS-DYNA d3plot Direct Reader	41
LS-DYNA	43
NPARC/WIND	44
OpenFOAM	46

OVERFLOW-2	48
PHOENICS - BFC Data	54
PHOENICS - non-BFC Data	54
PLOT3D	55
SC/Tetra	58
scFLOW	59
SCRYU	59
scSTREAM	59
.....	60
Surface Sampled Data	61
STL	61
UH3D	62
ultraFluidX	63
VTK	67
WIND US	68
XDB Import	70
Partitioned File Parallel Reader (PFPR)	72
Important points and limitations	72
Description of Layout File Format	73
Simple Layout File example	74
Limitations:	74
Partition File Parallel Reader Overload	78
Exports to FieldView Formats	79
AcuSolve	79
CFD-ACE	79
CFX	79
COBALT	80
CONVERGETM	81
DROP3D	84
FENSAP	84
FIDAP	84
Fine/Turbo	85
FLUENT	85
Exporting Particle Trajectories	90
FUN3D	92
GASP	92
POLYFLOW	92
.....	93
STAR-CCM+	93
Tetrex	93
ThermoAnalytics	93
Exports to FieldView Parallel Compatible Formats	93
Standalone Translators to FieldView Formats	95
BANFF	95
CFD++	95
CFX-TASCflow	95

COBALT60	95
PowerFlow	95
FIRE	95
GLACIER	95
USM3D	96
VECTIS	96
User Defined Plugin Readers for FieldView	96
AVUS	96
User Defined Plugin Readers for FieldView Parallel	97
Single file multigrid parallel	97
Partitioned file parallel	97
Unsupported features for Parallel Data Readers	98

Chapter 2

Functions.....	100
Function Specification Panel	100
Face Data and the Function Specification Panel	100
Face Data and the Function Selection Panel	101
Using the Functions Panel	102
Face Data and the Function Formula Specification Panel	103
Using the Function Formula Specification Panel	103
Frequently Asked Questions	105
Possible Issues	107
Differences between Datasets	107
Out of Range Handling	109

Chapter 3

Region Files	110
Introduction	110
Region Features	110
Region Subsetting	111
Converting Data into Cylindrical Coordinates	112
Region Hierarchy	113
Region Controls Panel	115
Region File Naming Convention	117
Transient FV-UNS and PLOT3D	117
Region File Version 2 Format	118
Omega Built-in Function	122
Region File Examples	123
Basic Coordinate Transform Example	123
Mirroring	124
Cylindrical Coordinate Example	125
Creating Smooth Radial Surfaces	126
Transforming Velocity Vectors	126
Adding Regions	128

Blade Row Example	129
Defining Machine and Zero Theta Axis	130
Adding Blade Rows	131
Creating the Region File	133
Rotational Duplication of Regions	136
Region File Version 1 Format	137

Chapter 4

FieldView Extension Language (FVX).....	142
Introduction	142
FVX Syntax	143
Chunks	143
Lexical Conventions	143
Types	143
Working with Tables	144
Variable Scope	146
Type Casting	146
Operators	146
Statements	147
Control Structures	148
Functions	151
General Function Library	154
Basic Functions	154
String Functions	156
Mathematical Functions	158
Standard I/O Functions	158
System Facilities Functions	162
CFD Open Post-Processing Functions	162
CFD Data I/O	163
Creation and Modification of Post-Processing Objects	175
FVX Show Min Max Annotation	201
.....	202
FVX Legends	203
FVX Support to Return Object Handles	207
Geometric Color and Scalar Colormap Specification	209
Vector Options	218
Annotation	219
Quantify and Query	223
Surface to Surface Sampling for Dataset Comparison	233
Transient Data Handling	236
Graphing	240
GUI Functions	244
Other Functions	246
Dynamic Clipping	249
FVX View Controls	250

FVX Debugger	251
Access to FVX Programs from the Tools Menu	252
Python-Enabled FVX	253
Support for Tkinter	254
FVX Learning Tools	256
FVX Tutorial Scripts	256
FVX Templates	257
Guide FVX saved with Restarts	257

Chapter 5

Restart Files and Script Language.....	260
Restart Files Menu	261
File Naming Convention	262
Automatic Restart	262
Restart Flexibility	262
Restart saved on Exit from FieldView	263
Restart Files Operation	264
Complete Restart	265
Complete, Current Window...	265
Complete Restart, No Data Read	266
Current Dataset Restart	267
Multi-Window Layout...	268
Layout Restart Files	272
Preference Restart	274
Script Restart	274
Formula Restart	274
Data File Input	275
Computational Surface	275
Iso-Surface	275
Streamlines	276
Particle Paths	278
Annotation	278
View (World)	278
Colormap Specification	278
Surface Plot	279
Boundary Surface	279
Vortex Cores / Surface Flows	279
Coordinate Surface	279
2D Plots Restart	279
Point Probe Input	280
Presentation Render	280
Clip Groups	280
FieldView Script Language Commands	280
Sample Scripts	304
Changing the View in an Animation	305

Holding View (pausing)	305
Animating Streamlines During View Interpolation	305
Animating Streaklines For Transient Data	306
Integrating Multiple Surfaces	306
Integrating Multiple Functions	307
Automating the creation of a Sampled Dataset	307

Chapter 6

Animation	308
Introduction	308
Flipbook Animation	309
Building a Flipbook Animation	310
Control and Playback of Animations	311
Output Formats	311
Examples	313
Keyframe Animation	316
Keyframe Actions	317
Keyframe Animation Panel	320
Error Conditions	329
Perspective and Mouse Controls	329
Surface and Region Detach	331

Chapter 7

Printing and Saving Images.....	333
Introduction	333
Printing and Saving Images	334
FieldView Pixel Resolution	337
Possible Problems	339
Error Conditions	340

Chapter 8

Advanced Numerical Functions	341
Vector Quantities	341
Unit Vectors	341
Surface Unit Normals	342
Create a Vector from a Scalar	342
Extract a Component of a Vector	343
Equal Length Vectors for Two Datasets	343
Non-Rotating Velocities using Rotating Quantities	344
Integral Quantities	345
Line Integrals	346
Volume Integrals	346
Integrated Force	348
Built-In CFD Functions	349

Miscellaneous Quantities	350
Curve Lengths in Structured Geometries	350
Second Derivatives	350
Rotating Quantities	351

Chapter 9

Building FieldView Plugins	353
Adding User-Defined Functions	353
Adding User-Defined Data Readers	356
Enabling the Passing of Constants to GUI Buttons	358
Writing a User-Defined Reader	359
Transient User-Defined Reader	379
Support for Cartesian Grids	383
Using User-Defined Plugins with a FieldView Server	384
Frequently Asked Questions	386
Writing and using Parallel User-Defined Data Readers	387
Grid-Parallel Data Readers	387
Partitioned-File Parallel Data Readers	388
Features Unsupported in Parallel Data Readers	389

Appendix A Built-In Functions	391
Geometric Functions	391
Scalar Functions Available with PLOT3D Q Files	391
Vector Functions Available with PLOT3D Q Files	393

Appendix B PLOT3D Formats	394
Introduction	394
Face Data and PLOT3D Format	394
Grid XYZ Files	395
Solution Q Files	397
Function Files	399
Face Data and Function Files	400
PLOT3D	402
PLOT3D Constants	403
Using the PLOT3D Data File Input Panel	403
Grid File Input	406
Using the Grid File Input Panel	407
Function File and Function Name File Input	407
Merge Series File Selection	409

Appendix C Function File Name Format	411
File Naming Convention	411
File Format	411
Face Data and Function Name Files	411

Error Conditions	412
Appendix D Unstructured Grid Format	413
General Remarks on Unstructured Data Format	413
Introductory note	413
Supported Element Types	414
Standard 3D element types	415
Arbitrary Polyhedron Cells	417
Arbitrary Polygon Boundary Faces	420
FieldView Compliance for Unstructured Data	420
Binary Format	422
General Remarks on Binary Format	422
Split Binary Format	422
Grid File in Split Binary Format	422
Results File in Split Binary Format	432
Combined (Grid & Results) Binary Format	437
Unformatted (FORTRAN 77) Format	451
Split Unformatted (FORTRAN 77) Format	451
Grid File in Split Unformatted (FORTRAN 77) Format	451
Results File in Split Unformatted (FORTRAN 77) Format	459
Combined (Grid & Results) Unformatted (FORTRAN 77) Format ...	465
ASCII Format	476
Split ASCII Format	476
Grid File in Split ASCII Format	477
Results File in Split ASCII Format	485
Combined (Grid & Results) ASCII Format	490
Transient Data	503
Creating FV-UNS files with FORTRAN 77 and C for different OS	504
ASCII FV-UNS files	504
Binary FV-UNS files	505
UNFORMATTED FV-UNS files	505
Appendix E Colormap File Format	509
File Naming Convention	509
Limitations:	510
Appendix F FieldView Limits	512
Per Session	512
Per Dataset	512
Per Grid	513
By Object	513
Legends	513
Surfaces	513
Streamlines	514
Annotation	514

2D Plots	514
Arbitrary Polyhedra	514
 Appendix G 2D Plot Format	 515
 Appendix H Structured Boundary Files	 516
Transient PLOT3D	516
Face Data and Surface Normal Information	517
CFX-4	517
NPARC/WIND and WIND US	517
Create Wall Bnd File	518
Create Exterior Bnd File	518
File Format	518
Face Data for PLOT3D Data	522
Face Data and Function Name Files	523
 Appendix I Plain Text Export Format	 524
Computational Surfaces	525
Coordinate Surfaces	525
Iso-Surfaces	526
Boundary Surfaces	527
Streamlines	528
 Appendix J MAT-File Export	 530
 Appendix K CSV Export	 531
 Appendix L Particle Path Formats	 532
ASCII	532
BINARY	536
BINARY PARTICLE SET Format Description	536
BINARY STREAKLINE Format Description	539
 Appendix M FieldView Math Fonts	 541
 Appendix N NPARC/WIND Constants and Formulas	 543

List of Figures

Figure 1:	Function Subset and Time Step Selection Panels	6
Figure 2:	PLOT3D and OVERFLOW-2 Data Input panels	8
Figure 3:	PLOT3D Data Input Error message	9
Figure 4:	Read Boundary Data Only	10
Figure 5:	Arbitrary Element Handling	13
Figure 6:	Dataset comparison for Arbitrary Element cases	14
Figure 7:	Arb Poly interpolation improvements	15
Figure 8:	Streamline improvements resulting from Arb Poly Interpolation	15
Figure 9:	FieldView Parallel Operation	18
Figure 10:	Single File Parallel vs Partitioned File Parallel operation	19
Figure 11:	Data Input pulldown menu	20
Figure 12:	Running Local FieldView Parallel	21
Figure 13:	AcuSolve [Direct Reader] Panel	23
Figure 14:	Multi-phase AcuSolve simulation visualized in FieldView	25
Figure 15:	CGNS Unstructured/Hybrid Data Input Panel	26
Figure 16:	CGNS Unstructured/Hybrid File Browser	27
Figure 17:	ANSYS-Fluent CFF [Direct Reader] Data Input Panel	31
Figure 18:	ANSYS-Fluent CFF [Direct Reader] File Browser for .cas.h5 files	31
Figure 19:	ANSYS-Fluent CFF [Direct Reader] File Browser for .dat.h5 files	32
Figure 20:	FLUENT cas/dat [Direct Reader] Data Input Panel	33
Figure 21:	Fluent cas/dat File Browser for .cas files	34
Figure 22:	Fluent cas/dat File Browser for .dat files	34
Figure 23:	Enight Data Input Panel	38
Figure 24:	Enight File Browser for ENSIGHT Gold files	38
Figure 25:	Tecplot 360 Data Input Panel	40
Figure 26:	Tecplot 360 File Browser	40
Figure 27:	LS-DYNA d3plot [Direct Reader] Data Input Panel	42
Figure 28:	LS-DYNA d3plot [Direct Reader] File Browser	42
Figure 29:	LS-DYNA interpolation issue with Thresholding	43
Figure 30:	OVERFLOW-2 Direct Reader for FieldView	48
Figure 31:	PLOT3D Data Input	57
Figure 32:	SC/Tetra [Direct Reader] Data Input Panel	58
Figure 33:	SC/Tetra [Direct Reader] File Browser	58
Figure 34:	scSTREAM [Direct Reader] Data Input Panel	59
Figure 35:	scSTREAM [Direct Reader] File Browser	60
Figure 36:	STL [Direct Reader] Data Input Panel	61
Figure 37:	STL [Direct Reader] File Browser	62
Figure 38:	UH3D Reader port for WINDOWS	63
Figure 39:	ultraFluidX [Direct Reader] panel	64
Figure 40:	FieldView results read operation	64
Figure 41:	Function Subset Selection panel	65
Figure 42:	Time Step Selection panel	66

Figure 43:	PLOT3D PFPR example for 8 partitions	75
Figure 44:	Reading a PLOT3D PFPR layout file	76
Figure 45:	Partition File Parallel Operation	78
Figure 46:	CONVERGE Internal Combustion Spray Modeling	81
Figure 47:	CONVERGE [FV-UNS Export] Data Input Panel	82
Figure 48:	CONVERGE [FV-UNS Export] File Browser	83
Figure 49:	Transient Set Confirmation	83
Figure 50:	Importing Spray Droplet data	84
Figure 51:	FLUENT Export Pull-down	86
Figure 52:	FLUENT Export Panel	87
Figure 53:	FLUENT Export File Browser	88
Figure 54:	FLUENT Execute Command GUI	89
Figure 55:	Export Particle Data Panel	90
Figure 56:	Export Particle Data Panel	91
Figure 57:	Save File Panel	91
Figure 58:	Function Specification Panel	101
Figure 59:	Function Selection Panel	102
Figure 60:	Function Formula Specification Panel	104
Figure 61:	Function Operations	105
Figure 62:	Region Hierarchy Schematic	113
Figure 63:	Transformed Region Hierarchy Schematic	114
Figure 64:	Region Controls Panel	115
Figure 65:	Region Names Panel	115
Figure 66:	Region Mirror and Rotate Parameters	116
Figure 67:	Transforming Datasets in Cartesian Coordinates	124
Figure 68:	Mirror Offset Problem	124
Figure 69:	Mirror Offset Correction	125
Figure 70:	Transforming Data into Cylindrical Coordinates	125
Figure 71:	Jagged Radial Surface	126
Figure 72:	Smooth Radial Surface	126
Figure 73:	Tangential phase-1_velocity component on R surface	127
Figure 74:	Radial phase-2_velocity component on theta surface	127
Figure 75:	Region Definitions	129
Figure 76:	Blade Row	129
Figure 77:	Machine Axis Defined in X-Direction	130
Figure 78:	Zero Theta Surface	130
Figure 79:	Period Definition	131
Figure 80:	Blade Row Definition	132
Figure 81:	Wheel Speed	132
Figure 82:	Turbo Region Example	134
Figure 83:	Transformed Velocity Magnitude on the Blades	135
Figure 84:	Displaying Omega	135
Figure 85:	Copying Regions	136
Figure 86:	Region Example 2	137
Figure 87:	FVX Example: Working with particle path data	198
Figure 88:	FVX Example: Select by particle diameter	198

Figure 89:	FVX Example: Select only recirculating trajectories	199
Figure 90:	FVX Example: Modify path display type	199
Figure 91:	Geometric Color Specification	210
Figure 92:	FVX Example: Controlling local scalar range	216
Figure 93:	FVX Example: Controlling local scalar range	217
Figure 94:	FVX Example: Controlling local scalar range	218
Figure 95:	Legacy Font illustration	221
Figure 96:	FVX Example: Text placement	222
Figure 97:	FVX Example: Drawing arrows	223
Figure 98:	Surface to Surface Sampling for Dataset Comparison	233
Figure 99:	Python-Enabled FVX Scheme	253
Figure 100:	Custom GUI panel showing some Tkinter widget examples	255
Figure 101:	Save/Open Restart menu options	261
Figure 102:	Restart File Panel	264
Figure 103:	Complete Restarts for some Tutorial Datasets	266
Figure 104:	Two datasets in a single window	269
Figure 105:	Applying a Multi-Window Layout Restart	270
Figure 106:	Layout Replace Warning	270
Figure 107:	Number of Datasets equals Number of Windows	271
Figure 108:	Number of Datasets greater than Number of Windows	271
Figure 109:	Number of Datasets less than Number of Windows	272
Figure 110:	Script View Controls and matching GUI actions	282
Figure 111:	Summary of Image Background SCRIPT command	284
Figure 112:	Interactive Dataset Mirror Copy	286
Figure 113:	Interactive Dataset Translate Copy	287
Figure 114:	Interactive Dataset Rotational Copy	287
Figure 115:	Tools Pull-Down Menu	309
Figure 116:	Flipbook Size Warning Panel	310
Figure 117:	Flipbook Controls Panel	311
Figure 118:	Flipbook File Save Panel	312
Figure 119:	Surface Sweep Extent and Step Control	313
Figure 120:	Streamline Build Control	314
Figure 121:	Transient Data Controls Panel in Flipbook Build Mode	315
Figure 122:	Dataset Control Sweep	315
Figure 123:	Keyframe Animation Panel	320
Figure 124:	Keyframe Animation Panel Create Action	321
Figure 125:	Keyframe Track Selection Panel	323
Figure 126:	Keyframe Value Specification Panel	324
Figure 127:	Keyframe Time Line Track Selection Panel	325
Figure 128:	Keyframe Time Line Display	326
Figure 129:	Keyframe Animation Perspective Warning	327
Figure 130:	Keyframe Delete Track Confirmation	328
Figure 131:	World/Dataset/Region/Surface Multi-Transform	330
Figure 132:	Toggle Multi-Transform Icons	330
Figure 133:	Light Multi-Transform	331
Figure 134:	Light Toggle Multi-Transform	331

Figure 135: Save images to file	334
Figure 136: Saving a Multi-Window Image to File	335
Figure 137: Postscript Options panel	336
Figure 138: Error message when direct printing is not configured	337
Figure 139: Rotating (left) and Non-Rotating (right) Vector Fields	345
Figure 140: Q criterion feature detection	349
Figure 141: PLOT3D Data Input Panel	402
Figure 142: Grid File Input Panel	406
Figure 143: Function Name Input Panel	408
Figure 144: Function Name Warning	408
Figure 145: Function Name Mismatch Panel	409
Figure 146: Merge Series File Selection	410
Figure 147: Face/Node numbering for Tetrahedron Cell type	415
Figure 148: Face/Node numbering for Pyramid Cell type	416
Figure 149: Face/Node numbering for Prism Cell type	416
Figure 150: Face/Node numbering for Hexahedron Cell Type	416
Figure 151: Overview of Arbitrary Polyhedron Cell type	417
Figure 152: Hex Cell with Trimmed Face and Center Value	419
Figure 153: Hex Cell with Hanging Node, No Center Value	419
Figure 154: Surface Normal Clockness	421
Figure 155: Example Unstructured Dataset	502
Figure 156: Unstructured Data Input Panel	503
Figure 157: Right-handed IJK system	521

Chapter 1

Data Files



Datasets can be read into **FieldView** in the following ways:

1. Using a built-in reader.
2. Exporting from a solver program to a **FieldView** compatible format such as PLOT3D or the **Field-View** Unstructured File Format, and then using one of the built-in **FieldView** readers.
3. Using a stand-alone translator, supplied by **Tecplot Inc.** or another commercial solver company, to convert data files to a **FieldView** compatible format.
4. Using a User Defined **FieldView** Plugin Reader.



Note: Information on commercial solvers, known problems and changes or updates to Plugin Toolkit Readers will be kept up to date on the Customer Support Section of the **Tecplot Inc.** web page. See www.tecplot.com/en/products/fieldview-solver-interfaces for the latest information on all changes and up to date information on reading data into **FieldView**.

Solver	Company	Method
AcuSolve	Altair	Export to FV-UNS
AcuSolve Direct Reader	Altair	Direct Plugin Toolkit Reader
AVUS	Government Version of COBALT ₆₀	Standalone Translator
BANFF	Reaction Engineering	Export to PLOT3D
CFD++	Metacomp Technology	Standalone Translator to FV-UNS
CFD-ACE	ESI Group	Export to PLOT3D
CFX	ANSYS	Export to FV-UNS

CFX-TASCflow	ANSYS	Standalone Translator to PLOT3D
CGNS Unstructured/Hybrid Reader		Direct Plugin Toolkit Reader
COBALT	COBALT CFD	Export to FV-UNS
COBALT60	Air Force Research Lab	Standalone Translator to FV-UNS
CONVERGETM	Convergent Science, Inc.	Export to FV-UNS
DROP3D	ANSYS	Export to FV-UNS
FENSAP	ANSYS	Export to FV-UNS
FIDAP	ANSYS	Direct Reader
Fine/Turbo	Numeca	Export to PLOT3D
FIRE	AVL	Standalone Translator to FV-UNS
FLOW-3D® Animation Data	Flow Science	Direct Plugin Toolkit Reader
FLOW-3D® Restart Data	Flow Science	Direct Plugin Toolkit Reader
FLOW-3D®	Flow Science	Legacy Direct Reader
ANSYS-Fluent CFF [Direct Reader] (current version)	ANSYS	Direct Plugin Toolkit Reader
FLUENT cas/dat Direct Reader (current version)	ANSYS	Direct Plugin Toolkit Reader
FLUENT (current version)	ANSYS	Export to FV-UNS
FLUENT Universal (version 4.2 and below)	ANSYS	Universal Files
FLUENT/UNS (and RAMPANT)	ANSYS	FieldView Case and Data files
FrontFLOW		Export to FV-UNS
FUN3D	NASA	Export to FV-UNS, VTK, CGNS or Tecplot Binary format
GASP	Aerosoft	Export to PLOT3D

GLACIER	Reaction Engineering	Standalone Translator to PLOT3D
HAVOC	Corvid Technologies	Direct Reader
LS-DYNA	LSTC	Direct Plugin Toolkit Reader
NPARC/WIND	NPARC Alliance	Direct Reader
OpenFOAM	ESI	Direct Reader
OVERFLOW-2	NASA	Direct Reader
PHOENICS - BFC Data	CHAM	Direct Reader
PHOENICS - non-BFC Data	CHAM	Direct Reader
PLOT3D	NASA	Direct Reader
POLYFLOW	ANSYS	Export to FV-UNS
PowerFlow	EXA	Export to FV-UNS
Pratt	Pratt Whitney	Direct Reader
RavenCFD	Corvid Technologies	Export to FV-UNS
SC/Tetra	Software Cradle Co., Ltd.	Direct Reader
scFLOW	Software Cradle Co., Ltd.	Direct Reader
SCRYU	Software Cradle Co., Ltd.	Direct Reader
scSTREAM	Software Cradle Co., Ltd.	Direct Reader
STAR-CCM+	CD-adapco	Export to FV-UNS
STL	Stereolithographic CAD	Direct Reader
Tetrex	Tetra Research	Export to FV-UNS
ThermoAnalytics	ThermoAnalytics	Export to FV-UNS
UH3D	Mindware	Direct Reader
ultraFluidX	Altair	Direct Reader
USM3D	NASA	Export to FV-UNS
VECTIS	Ricardo	Export to FV-UNS
VTK	Kitware	Direct Reader
WIND US	NPARC Alliance	Direct Reader
XDB Import	Tecplot Inc.	Export from FV

Overview on Reading Data in FieldView

GRID PROCESSING

The GRID PROCESSING section on the Data Input panels allows users to interactively balance performance versus read time and memory. Settings can be easily changed, depending on usage, to maximize productivity.

When **FieldView** reads data, it performs grid processing after reading the grid. Grid processing increases **FieldView**'s speed in such operations as creating and sweeping Coordinate Surfaces, computing accurate streamlines, and Dataset Sampling. The costs of grid processing are read time and memory. Previous versions of **FieldView** required setting environment variables to balance performance versus read time and memory. For a full list of operations benefiting from this initial grid processing, please refer to [page 25](#) of the **FieldView User's Guide**.

Grid Processing setting guidelines:



- If you don't know what setting to use, try "Balanced" (the default value)
- If your data takes a long time to read or if you're going to read many time steps, try "Less"
- If some operations are too slow (see the list of operations impacted by grid processing on [page 25](#) of the **FieldView User's Guide**), try "More"
- The recommended Grid Processing option for performing Surface Sampling is "Less"

GRID PROCESSING is shown on the PLOT3D and OVERFLOW-2 panels in [Figure 2](#) and the FV-UNS panel in [Figure 4](#). The slider defaults to a "Balanced" position between faster data input using less memory and faster performance using more memory. If you wish to post-process your data using many Coordinate Surfaces or streamlines, or if you're going to use Dataset Sampling for comparing results, setting the slider to "More" will improve **FieldView**'s performance. If performance of these features is less critical, setting the slider to "Less" will minimize read time and memory.

This Grid Processing setting is saved in, and read from, the Data Input restart file or **FVX**.

Exceptions

The presence of the following environment variables will disable Grid Processing controls on the Data Input Panel, and override the "grid_processing" settings in Data Input (.dat) restarts and FVX scripts for the entire FieldView session:

```
FV_PROBE_PERFORMANCE  
FV_PROBE_SAVE_MEM
```

These environment variables (maintained for backward compatibility) offer another way of setting the Grid Processing level, but for the entire FieldView session. For more on these environment variables, please refer to the [Balance Between Memory Usage and Performance](#) section of the **FieldView User's Guide**.

When any one of these environment variables is set, the Data Input Grid Processing GUI will display a field explaining why it is disabled, for example:

"Grid Processing controls disabled because `FV_PROBE_PERFORMANCE` is set".

This message will also be printed to the console.

If `FV_PROBE_SAVE_MEM` is set, Grid Processing is set to Less, as the goal of this environment variable is to use as little memory as possible during the read phase.

If `FV_PROBE_PERFORMANCE` is set, Grid Processing is set to Balanced on the **FieldView** interface, but this setting is actually ignored, with precedence given to `FV_PROBE_PERFORMANCE`.

There are certain conditions detected during the read such that Grid Processing will be ignored:

A Grid **DataGuide™** file (`.fvpre`) was found and used.

In this case, the grid setting used is the same as the one used at the time of the generation of the **DataGuide™** file.

The message printed to the console is:

The Grid Processing setting was ignored because a DataGuide file was found and used.

The dataset being read is Cartesian.

In this case, grid processing is not needed.

The message printed to the console is:

The dataset is Cartesian.

Grid Processing is not needed for Cartesian.

The grid file is a duplicate of a file read earlier.

In this case, the grid setting used is the same as the one used before.

The message printed to the console is:

The grid file is a duplicate of an earlier grid file.

The Grid Processing setting was inherited from the earlier grid file.

Function Selection

Reading results data in Replace or Append mode using most readers will automatically bring up the Function Subset Selection panel containing a list of the variable names from the results file.

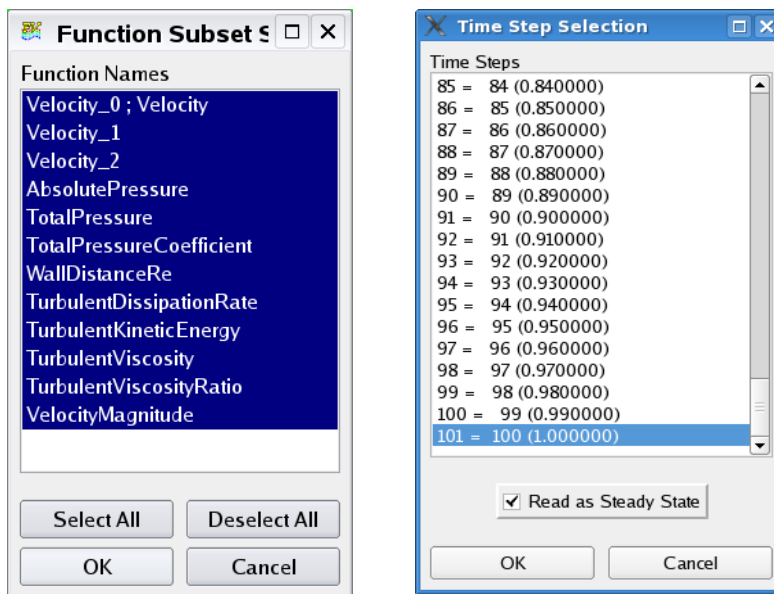


Figure 1 Function Subset and Time Step Selection Panels

All Function Names will be initially selected (highlighted). Clicking on a selected Function Name will deselect (unhighlight) it and vice versa.

Vector Function Selection

Vector functions are indicated on the Function Subset Selection panel by a triplet of scalar variable names, the first of which has a “;”, followed by the vector name (“Velocity” in the panel shown in **Figure 1**). Selecting any of the three consecutive variables that define a vector function will automatically select the other two variables. Similarly, deselecting a variable that belongs to a vector function will deselect all three lines that belong to the vector function. In the example panel shown in **Figure 1**, deselecting “Velocity_1” will also deselect “Velocity_0 ; “Velocity” and “Velocity_2”, since they form a vector triplet.

Transient Data

For a transient dataset given as a series of files, **FieldView** will prompt you to read the whole series of files and treat them as a transient dataset.

A dataset will automatically be recognized as transient as long as a particular file naming convention is used. The file naming convention embeds the integer *time step* number (but *not* the *solution time*) in the filename. This is represented as ##### below. Note that you should use as many digits as required to represent your time steps. There is no requirement that five be used. **FieldView** looks for the embedded time step value (#####) to the left of the first ‘dot’ (.), if there is one in the file name, otherwise it will search from the right. The required format can be one of either:

```
prefix####.extension  
prefix.####
```

Example: pipe04020.extension, pipe04021.extension, ..., pipe04073.extension
The actual extension used is not important.



Note: Only single byte numbers in filenames will be recognized as time steps in a transient series.

For the readers that produce a single file containing transient data (AcuSolve, OpenFOAM, FLOW-3D® and FIDAP), **FieldView** will present you with the Time Step Selection panel (see **Figure 1**, right) when the data is read. This will show you the time steps found in the file(s) and allow you to choose which one you wish to initially read into memory. By default, **FieldView** selects the *last* time step to read in. You will have to explicitly choose a different time step if that is desired.

When turned on, the 'Read as Steady State' toggle button allows the selection of a specific time step of a multiple time step per file dataset.

This capability is particularly of interest for quickly animating particles computed for a transient dataset but with a fixed geometry over time. This can be done by:

- Reading the dataset with the “Read as Steady State” option ON.
- Importing a STREAKLINE (not PARTICLE SET) format Particle Path file, with the Import button from the Particle Paths panel.
- Setting the Display Type to one that is compatible with animations, such as Growing, Spheres, Dots or Polyspheres.
- Using the Animate button on the Particle Paths panel.

Since only particles will be updated, it will be many times faster than a full transient animation, in which the volume data would be read and updated at each step.

The transient panel will be disabled and transient restrictions on particle path display options will be lifted. The Particle Path Animate button will also be enabled in this mode.

Note that in order for time step or solution time information to be available in a **FieldView** session (for example, to be included in an annotation using the **escape sequence** %%T or %%N), the dataset needs to be read as transient with the 'Read as Steady State' option turned off.

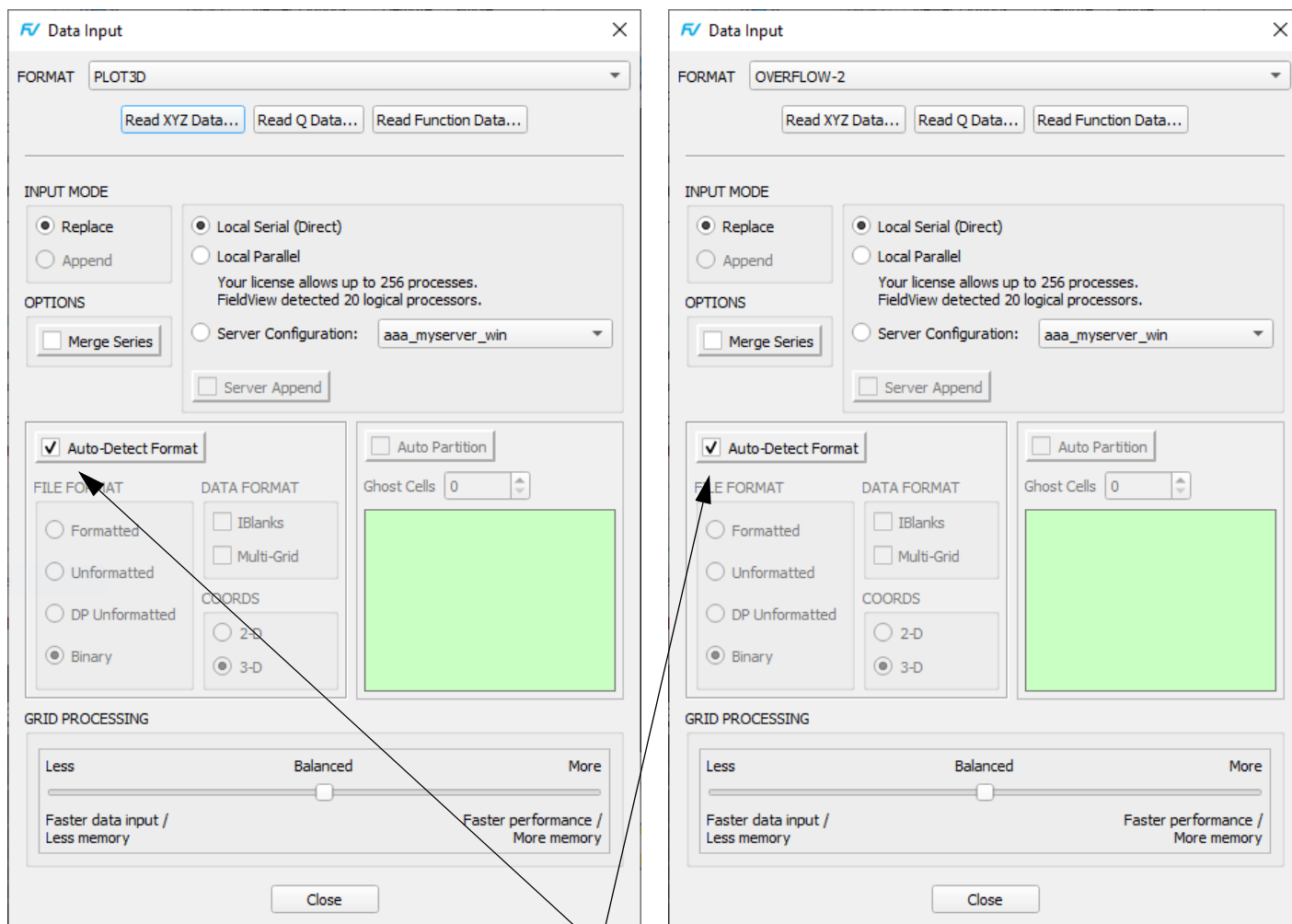
For *all* transient data, *any* time step can be accessed through the Transient Data Controls panel (see **Chapter 14** of **Working with FieldView** for more information).

Changing of time step for a transient dataset will not cause all grid and results files to be re-read. Only the file or files containing the results of the current dataset will be re-read when the time step number is changed. The transient sweep reading of grid and results files (PLOT3D, FV-UNS-split) is optimized

so that only the files that need to be changed for the new time step will be read. An invariant grid for the current dataset and all files for other datasets will remain in memory.

PLOT3D & OVERFLOW-2 Auto-Detect Format

PLOT3D and OVERFLOW-2 data files can be saved using different file formats and attributes. A previous limitation of the **FieldView** data input panels for both types of data was that the file formats and attributes needed to be explicitly and correctly set before the file could be read. By default, **FieldView** will attempt to automatically detect the values for the **FILE FORMAT**, **COORDS** and **DATA FORMAT**. You still have the option to disable this default behavior by turning the **Auto-Detect Format** option OFF - in this case, explicit setting of the file format and attributes will be required.



Auto-Detect Format is on by default

Figure 2 PLOT3D and OVERFLOW-2 Data Input panels

If **FieldView** is unable to correctly detect the file format, the **Auto-Detect Format** button will be turned off, and the following pop-up message will appear:

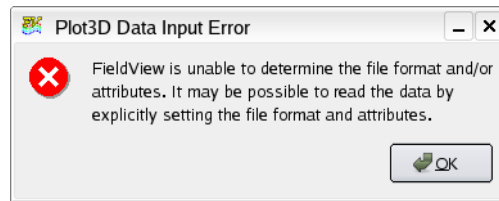


Figure 3 PLOT3D Data Input Error message

Note that if **Auto-Detect Format** switch has been turned OFF, it will remain OFF unless you turn it back ON - it does not get reset back to the original default during the session.

Auto-Detect can only be used interactively or with **FVX** Data Input. The syntax for the **FVX** `read_dataset` command is:

```
auto_detect = 'on'
```

or

```
auto_detect = 'off'
```

Auto-Detect is NOT read or written to the data file restart.

Auto-Detect is enabled whether using Direct or Server, and supports non-parallel servers, parallel servers, and partitioned-file parallel (PFPR).

Read Boundary Data Only

Often it is useful to review only the boundary surfaces for a dataset. By limiting the dataset read operation to read in just the boundary data, a significant reduction in the time required to read can be realized. Reading of boundary data only is available for all unstructured data readers, including user-defined (toolkit plugin) unstructured data readers.

If an unstructured data reader always has boundary data without any volume data, such as XDB Import, or STL [Direct Reader], then Read Boundary Data Only is always enabled and the check button is grayed out.

The time savings associated with Read Boundary Data Only will depend on the ratio of boundary to volume data, and will typically be in the range of 15x to 20x faster than reading the entire volume data for a given dataset.

Read Boundary Data Only is enabled by turning on the check button in the Data Input panel (see [Figure 4](#)). This setting is off by default. Read Boundary Data Only is fully compatible with **FieldView** Parallel. Partitioned File Parallel formats are also fully supported. This feature will also work correctly to examine extruded 2D boundary data, created using the `FV_2D_TO_3D` environment variable.

Recognize that all surface and rake types which rely on volume data cannot be created following a read boundary only data option.

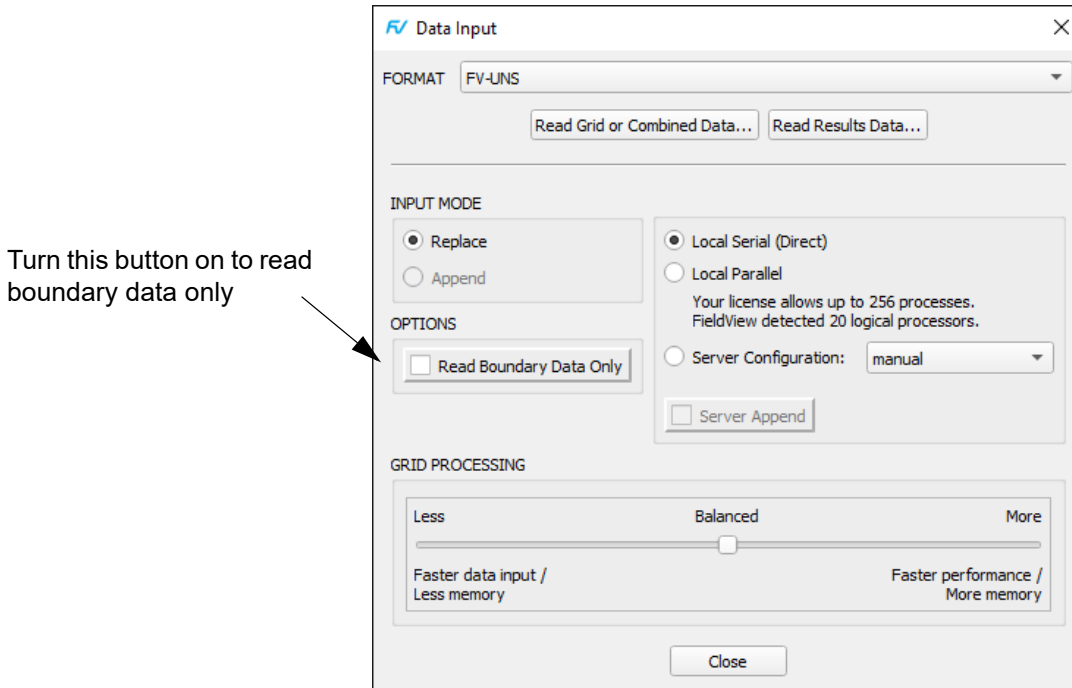


Figure 4 Read Boundary Data Only

Complete support has been provided for RESTARTS. However, it is important to keep in mind that it will not be possible to create any rakes or surfaces which need volume data. So, consider the case where a Complete RESTART has been saved for a full volume data read. If we attempt to apply this RESTART (ALL, NO_DATA_READ) following a Boundary Data only read, any surfaces based on volume data will not be present.

Regarding transient streaklines however, It is possible however to do the following:

1. Read the volume data and compute streaklines from a transient sweep. This can be automated to run in batch.
2. Read the Boundary Data only, read in the saved streakline data (particle path) and create an animation from a transient sweep.

In the above scenario, the time savings by not having to read in the volume data at each time will be considerable. In general, the time required to read Boundary Data only is dramatically reduced relative to the time needed to read the volume data for the same case.

This feature is currently limited to work with unstructured datasets only. Writing **DataGuide™** files is disabled when the read boundary only switch is on.

Reading more than one Dataset at a time

You can read multiple datasets into **FieldView**. This is done with the Append button that appears on all of the data input panels. This will allow you to read in, display and visualize several datasets at the same time.



Note: The restarts from the current dataset can be applied to the second dataset. Implementing this feature can save you a considerable amount of time. For more information see "[Current Dataset Restart](#)" of the **Reference Manual** or [Chapter 14](#) of the **User's Guide**.

The datasets that you read into memory do not have to be of the same type or format or have the same number of variables. One can be a structured dataset and another can be an unstructured dataset. However, a Results file Append will always try to read results for the highest-numbered dataset. If it succeeds, the *current* dataset will be changed to the highest-numbered dataset. A Results file Replace will (as in previous versions of **FieldView**) always try to read results for the *first* dataset. If it succeeds, the current dataset will be set to 1. Therefore, reading grid files and results files in the following order: Grid1, Grid2, Result1, Result2 will *not* yield the desired result in **FieldView** memory. The order Grid1, Result1, Grid2, Result2 will need to be used.

Switching between datasets can be accomplished on the Dataset Controls panel or the Main Toolbar (see [Chapter 14](#) of **Working with FieldView** for more information). You can also switch between datasets by "quick-picking" (double-click) on any surface, rake, legend, etc. that belongs to that dataset.

Each dataset loaded into memory will have a separate set of functions (results) which will be shown when the Functions button is pressed. Variables of one dataset *cannot* be used in formulas for a different dataset (with the exception of Dataset Comparison mode).

For PLOT3D and split **FieldView** Unstructured files the grid will be automatically re-used when appending results that are based on the same grid file. We expect users to save significant disk space, and to see a substantial improvement in the handling of datasets where the grid is invariant with time, or across several datasets.

All surfaces and rakes created will belong to a specific dataset. If you have two datasets in memory and each dataset has a computational surface created in it, then both computational surfaces will be surface number 1, specific to the given dataset. A surface cannot be switched to a different dataset. Control of the *current* dataset is through the Dataset Controls panel or the Main Toolbar.

Dataset Comparison for Multiple Datasets

FieldView will let you make numerical comparisons between datasets. If the datasets are based on the same underlying grid, Dataset Comparison is directly possible and formulas spanning datasets can be created using the Function Specification Panel (see [Function Specification Panel](#)). If the underly-

ing meshes differ, or if the datasets are based on different file formats, then the Dataset Sampling Tool can be used to create a sampled dataset which can then be used for numerical comparisons (see [Dataset Sampling](#)). Note that both Dataset Comparison and Dataset Sampling requires that when your input data is not read serially with the FieldView Client and remote or local servers are used, the Append Server button *File.. Data Input.. Server Append* option must be checked ON. This is the default. Note also that Dataset Sampling is not supported on parallel servers.

Merged Transient Datasets

If more than one transient dataset is read into **FieldView**, it is possible to animate all time steps using merged solution times for all datasets during the transient sweep. This feature simplifies the task of creating an animation of multiple transient cases (see [Use Merged Times](#)).

Appending Datasets to the same Server Process

This feature permits you to use the same SERVER process to read more than one dataset. To turn this feature on, first read a dataset into **FieldView** using the Client-Server feature. Then, to read another dataset to that same SERVER process, in the File menu Data Input pulldown, turn the Server Append button ON. With a minimum of two datasets read into **FieldView** in this way, the Dataset Sampling and Dataset Comparison features will be fully enabled, except that Dataset Sampling is not supported when using Parallel Servers. These features will be discussed in more detail in sections describing the Client-Server operation of **FieldView**.

Data Written On Different Systems

It is known that files created on LINUX or Windows systems have different byte ordering compared to their UNIX counterparts. It is also often the case that datasets are created on one system, and post-processed on another. The **FieldView** Unstructured (FV-UNS) and PLOT3D readers permit reading of non-native byte ordered FV-UNS (all types) and PLOT3D (all types) files.

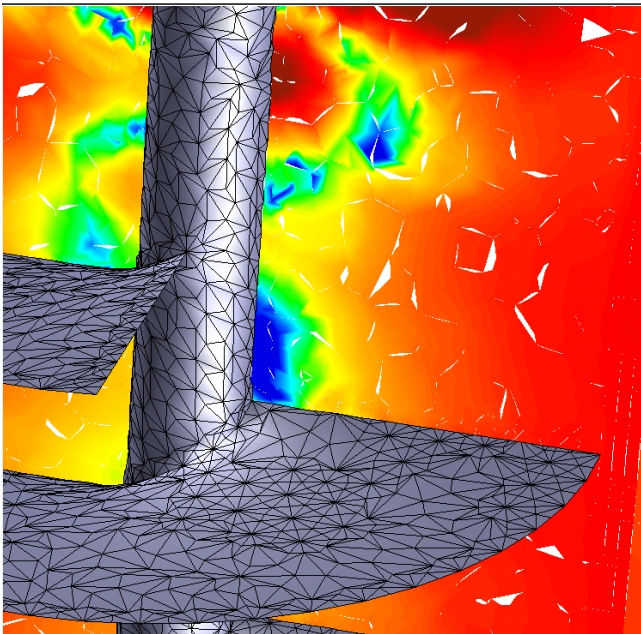
Support for Arbitrary Elements

Arbitrary elements are now widely used by commercial CFD solvers. A common discretization strategy is to mesh the internal volume with regular hexahedral cells, the near-surface volume with extruded cell layers, and generate arbitrary polyhedra to fill the gap in between the two. By default, FieldView employs interpolation and gradient calculations natively for arbitrary polyhedra, consistent with other commercial solvers such as STAR, STAR-CCM+ and FLUENT. Tetrahedralization on-the-fly is only used in the following two cases:

- The cell has a cell-center node
- The cell has face-centered nodes

The importance of native handling for arbitrary polyhedra is highlighted in the illustration below:

Tetrahedralization



Native Arbitrary Element

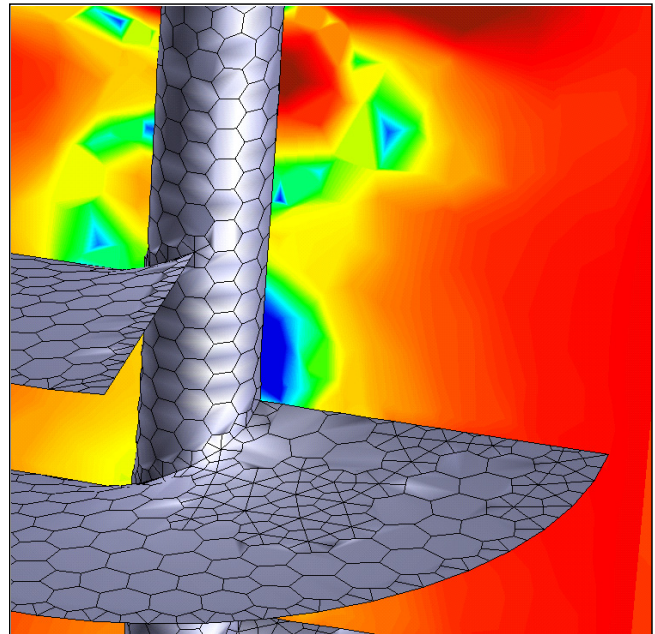


Figure 5 Arbitrary Element Handling

Holes in the tetrahedralized mesh (on the left) are clearly seen.

Special handling is implemented in **FieldView** to calculate derivatives (gradient, divergence and curl) for the case of split planar faces. These types of faces are commonly encountered in hexcore style meshes, where mesh resolution within the core regions change from coarser to finer cells.

Several key features are supported for datasets containing arbitrary elements:

- Visual Dataset Comparison can be used to create side-by-side illustrations
- Numerical Dataset Comparison with full support for Dataset Sampling
- Current Dataset Restarts can be created on tet or hex mesh cases and applied directly to arbitrary element cases
- Surface based streamlines are supported
- Separation and re-attachment lines are supported.

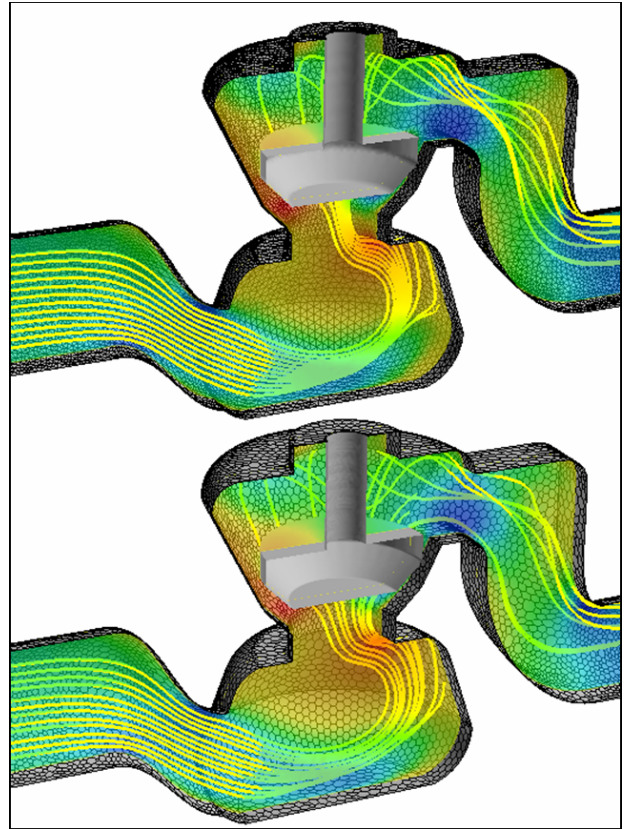


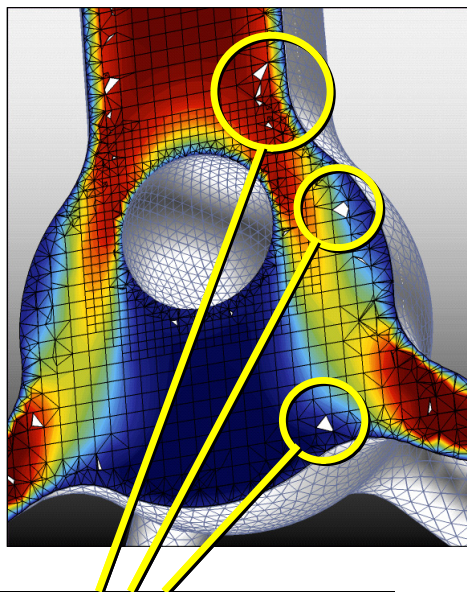
Figure 6 Dataset comparison for Arbitrary Element cases

Considerations when working with datasets containing Arbitrary Polyhedra

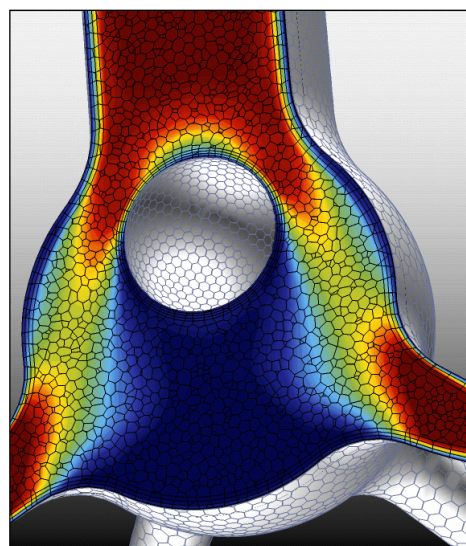
An individual element must have less than 256 vertices on a single face. And there can be no more than 256 faces for a single element. Typically arbitrary elements will have 10 to 20 faces, with each face having up to 10 edges. These values lie well below the limits specified.

The vortex core detection schemes and shock surface feature detection are currently not supported for arbitrary elements.

If holes are seen in surfaces, it is because there is face mismatching between adjacent cells which are intersected by the cutting planes of interest.



Holes are present due to face mismatching between adjacent cells.



No holes should be present for arb poly meshes where adjacent face matching is correctly handled.

Figure 7 Arb Poly interpolation improvements

Note that face mismatching can be tolerated by some commercial solver codes to obtain a solution. In most cases however commercial solvers can run checks to find and fix holes in meshes.



streamlines stop short on tetrahedralized mesh



streamlines continue to outlet with native arb poly handling

Figure 8 Streamline improvements resulting from Arb Poly Interpolation

Owing to the native handling of arbitrary polyhedra, streamlines should not stop at arbitrary polyhedral cell split faces. Since these cell types are commonly encountered in hexcore meshes, it is again important to use native arbitrary polyhedral handling. Also, the native interpolation methodology being applied provides for exact streamline seed placement. Streamline calculation times are slower for arbitrary polyhedra relative to tetrahedral meshes.

In general, streamline calculations are expected to be 1.3x to 2.0x slower. These factors are derived from test cases with comparable node counts. If you conduct timing tests on a polyhedral mesh which has been derived from a tetrahedral mesh, the node count for the former will be much greater. Since streamline calculation times are generally proportional to the number of nodes, timing comparisons need to take this into account.

With some data, **FieldView** may print messages similar to the following during data input, and data will be missing:

```
FV-UNS: Some arbitrary polyhedral cells were skipped (not read):
```

```
2 cells were skipped for the following reason:
```

```
Not enough nodes for an arbitrary polyhedron cell face.
```

```
12 cells were skipped for the following reason:
```

```
Arbitrary polyhedron cell is not valid. Faces did not connect to each other, or a face had a duplicate edge.
```

If you run with `FV_DEBUG` set, you will see details prior to the above report, for example:

```
Invalid connection between cell faces at node: 2992.82 412.086 17.5403
Invalid connection between cell faces at node: 3314.28 721.914 48.1584
Invalid connection between cell faces at node: 3315.8 720.69 22.0393
Invalid connection between cell faces at node: 3316.11 720.564 28.4177
Invalid connection between cell faces at node: 3314.56 721.618 17.5472
Invalid connection between cell faces at node: 3317.15 718.894 48.3413
Invalid connection between cell faces at node: 3317.06 719.917 42.6953
Invalid connection between cell faces at node: 3316.03 720.999 31.8735
Invalid connection between cell faces at node: 3316.27 719.838 48.3413
Invalid connection between cell faces at node: 3316.73 720.35 48.1584
Less than 3 face vertices at node: 2993.55 411.326 17.5155
Invalid connection between cell faces at node: 2993.08 411.82 17.5282
Less than 3 face vertices at node: 2993.44 411.442 48.1495
Invalid connection between cell faces at node: 2998.09 407.904 46.0248
```

In addition, the following may print (with or without `FV_DEBUG`) due to new handling as of **FieldView 17**:

```
Some arbitrary polyhedral cells in grid N are not valid.
```

```
Attempting repair of invalid cells. This will take extra time.
```

If you still have problems, set the following environment variable to any value to revert to the more extensive checking and repair done in **FieldView 16.1** and earlier:

```
FV_ARB_POLY_FULL_CHECK
```

If you are writing code to export FV-UNS files, setting `FV_ARB_POLY_FULL_CHECK` while testing your export is recommended. If `FV_ARB_POLY_FULL_CHECK` is set, a console warning will be printed the first time any data with arbitrary polyhedron cells is read.

To revert to the previous behavior for reading arbitrary polyhedron cells, set the environment variable:

```
FV_PRE17_ARB_POLYS
```

If `FV_PRE17_ARB_POLYS` is set, a console warning will be printed the first time any data with arbitrary polyhedron cells is read.

```
Some arbitrary polyhedral cells were skipped (not read):
1404 cells were skipped for the following reason:
Arbitrary polyhedron cell faces do not have consistent clockness. All
the faces of a cell must have the same clockness.
```

For cases such as the above, it may be desirable to tetrahedralize a dataset containing arbitrary polyhedra. A quick tetrahedralization scheme can be implemented during a data read operation using the following environment variable:

```
FV_TET_CONV = 1
```

Another environment variable can be used to control how **FieldView** tetrahedralizes native arbitrary polyhedral cells:

```
FV_ARB_POLY = 1
```

If `FV_TET_CONV` is set, then setting `FV_ARB_POLY` will change the tetrahedralization scheme to a slower, more robust scheme. If failures in tetrahedralization occur, errors are sent to the console.

The feature extraction toolkit is currently based on tetrahedral cells. So, it is possible as a work-around to read an arbitrary element dataset using the `FV_TET_CONV` environment variable, calculate and export Vortex Cores, and then re-import them as particle paths to a dataset read using native arbitrary elements.

Note that this environment variable should not be used if **FieldView** instead prints a message similar to the following:

```
Some arbitrary polyhedral cells were skipped (not read):
NNNN cells were skipped for the following reason:
```

Arbitrary polyhedron cell is not valid. Faces did not connect to each other, or a face had a duplicate edge.

The environment variable can not be used to repair a grid with invalid arbitrary polyhedra, and errors can be expected in processing results in this way.

FieldView Parallel Datasets

FieldView Parallel uses several worker processes, the total number being specified by the user, to read datasets or create surfaces and/or rakes. The process of how the **FieldView** Client is used to pass commands to the controller/worker servers, and receive information back from them is illustrated in the figure below. At this time, the only "grid-parallel" data formats supported are either multigrid FV-UNS, FLOW-3D, OpenFOAM, OVERFLOW-2, PLOT3D datasets, or those datasets which are read using plugin readers written for parallel multi-grid file support. Note that all readers are supported for "partitioned-parallel" (i.e., PFPR).

The Controller Server handles communications between the Worker Servers and the **FieldView** Client. (Note that the **FieldView** Client is the part of **FieldView** that you normally see when you run **FieldView**). An outline of the data flow when running **FieldView** Parallel is shown in the following figure:

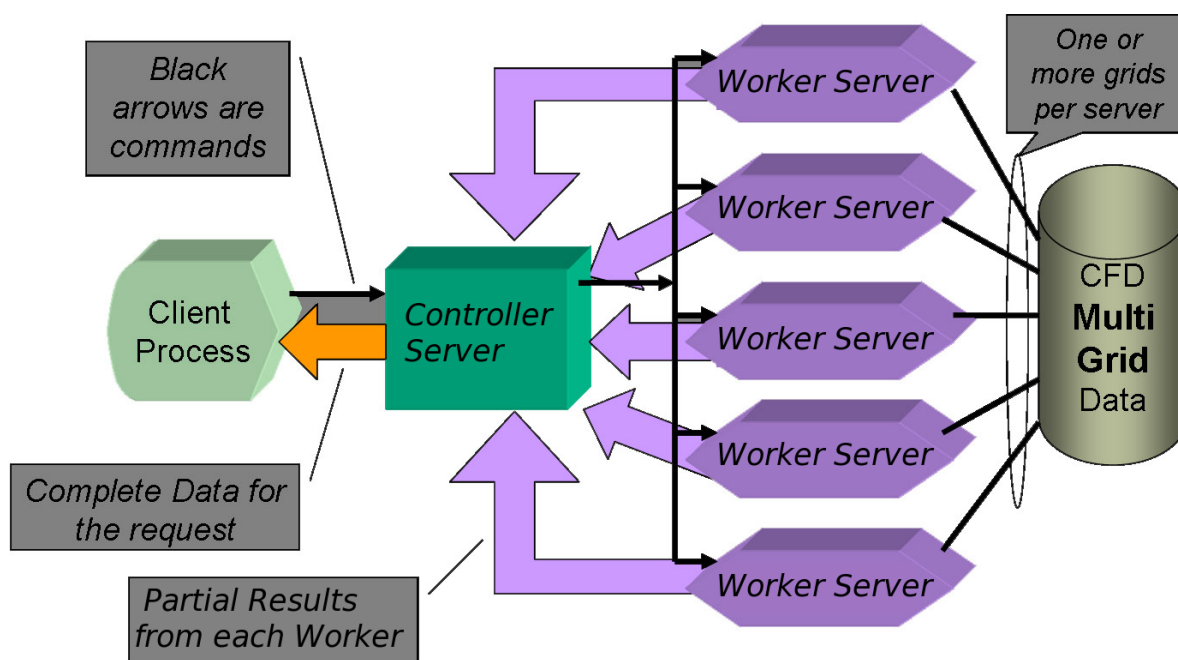


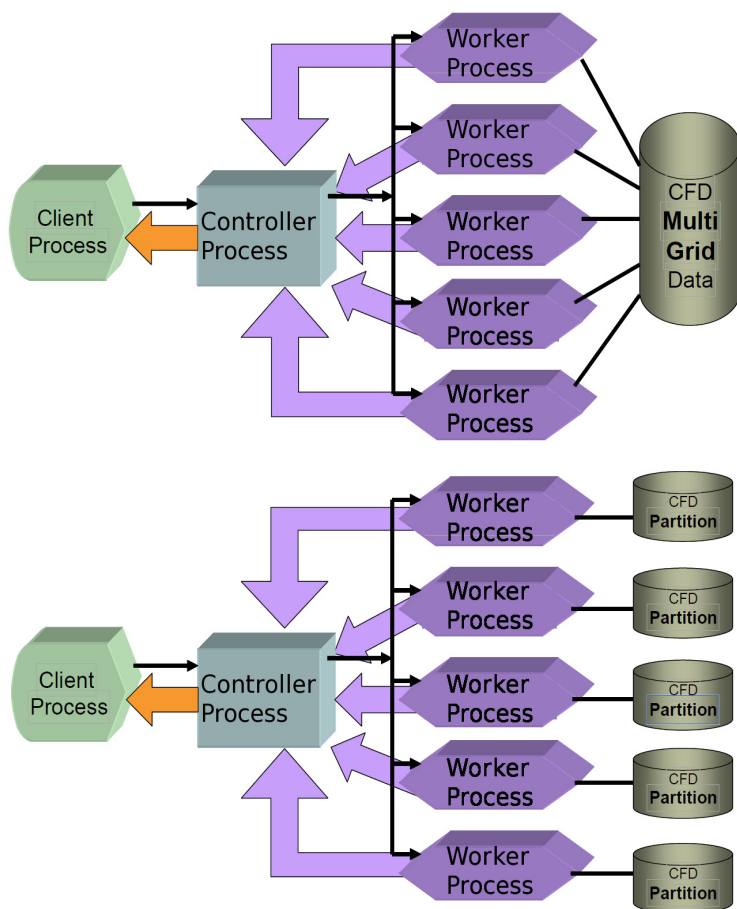
Figure 9 FieldView Parallel Operation

The **FieldView** Client first initiates the process of reading data. Grids from a dataset are distributed onto the Worker Servers, as directed by the Controller Server. To create surfaces or rakes, the client sends a request to the Controller Server. The Controller Server, in turn, calls the Workers in a loop.

As each Worker completes its task, the Controller collects information. Once all Worker tasks are complete, the Controller delivers the information back to the Client for visualization.

Q. What is the difference between a single multigrid vs parallel partitioned file?

The fundamental requirement of a dataset in order for it to show any scaling performance with **FieldView** Parallel is that it must be made up of multiple grids (unless the **Auto Partitioner** is used). At present the most commonly available type of multiple grid dataset is the one in which all of the grids are stored in a single multigrid file. However, it is possible to break up multiple grid datasets into multiple files, where each of the files may themselves contain multiple grids. We refer to this type of dataset as being “partitioned”. Whether a file has been partitioned or not will have no effect on the Client side operation of **FieldView**, as is illustrated below. Partitioned data can be written using any of the supported **FieldView** data formats, and is read into **FieldView** using a layout file. Please refer to the section entitled **Partitioned File Parallel Reader (PFPR)** for more details.



Parallel FieldView

A dataset containing multiple grids is read using several worker server processes. Grids are load balanced across the available worker processes.

Partitioned File Parallel FieldView

A dataset, split into several different partitions. Each partition can contain one or more grids, and the partitions are likely to be located on separate file systems.

Figure 10 Single File Parallel vs Partitioned File Parallel operation

In the example above for the Partitioned File Parallel **FieldView** case, 5 worker processors, each associated with its own file system, are used in parallel. The partitioning of the single dataset into several pieces is usually performed by the CFD solver code as it sets up to run the job in parallel.

Direct Readers in FieldView

The Data Input pulldown menu offers some or all of these panel selections (depending on licensing):

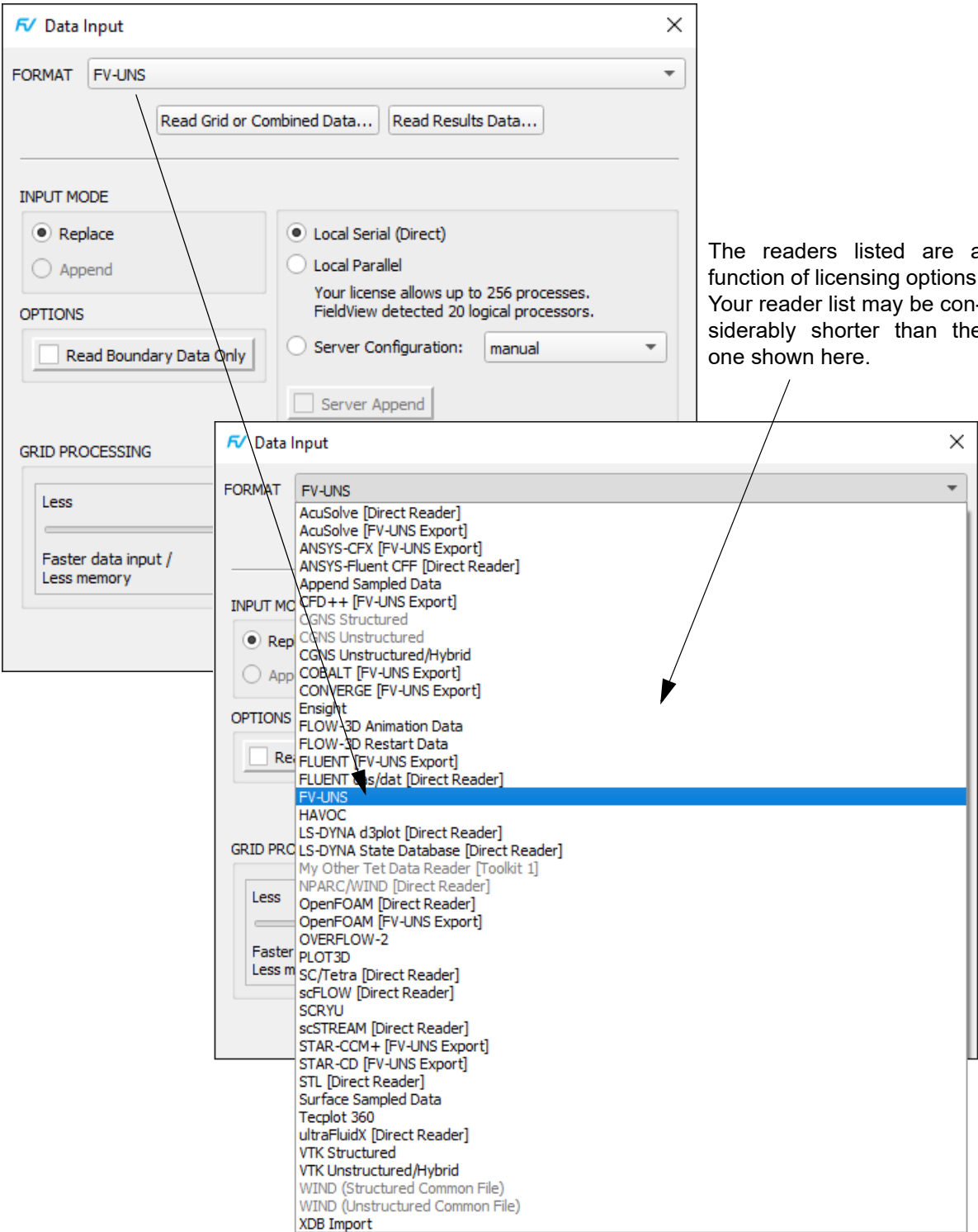


Figure 11 Data Input pulldown menu

Working with the Data Input Menu

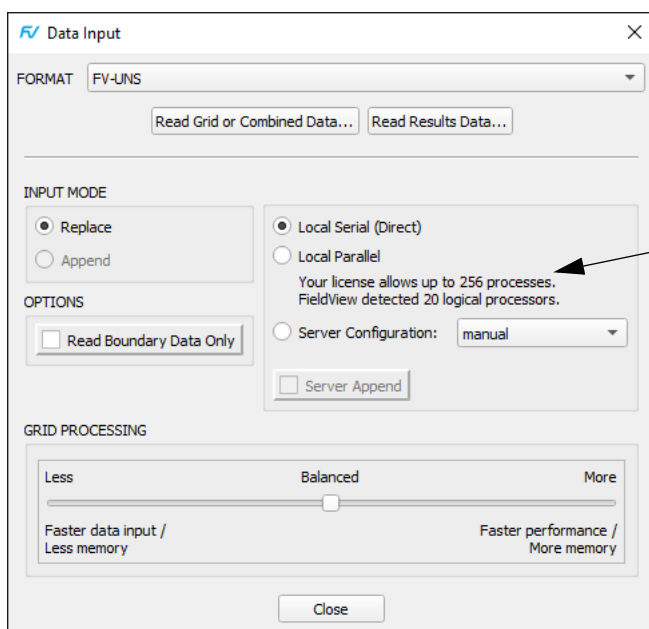
The Data Input menu (shown in **Figure 11**) is made up of different sections. For several commercial CFD solvers, there are either translators or export tools available to convert their output into a **FieldView** compatible format. Getting data from these sources into **FieldView** is indicated in the menu with the phrase [FV-UNS Export] - this implies that you will need to do something to the solver output in order to be able to read the data into **FieldView**. So, as an example, in order to read data from the commercial solver AcuSolve which has been translated into the **FieldView** Unstructured File format, you would select the top entry, AcuSolve [FV-UNS Export], from this menu pulldown.

Other entries on the menu permit you to read data in its native format directly to **FieldView**. Examples above include CGNS, FLOW-3D, HAVOC, PATRAN, SCRYU and UH3D, to name a few. There are some cases where you can either read your dataset directly, or read the translated or exported dataset instead. For the case of commercial solver FLUENT, you can either read the FLUENT native files directly using one of the FLUENT [Direct Reader] selections from the Data Input menu, OR, you can read an exported FV-UNS form of the data from FLUENT using the FLUENT [FV-UNS Export] option.

Note that legacy readers found in earlier versions of FieldView on a section called 'More Readers' has been removed, but FieldView still support restarts and scripts which used those readers in order to maintain backward compatibility with those readers.

Reading Data Interactively with FieldView Parallel

When **FieldView** is started it performs the following three checks: 1. That **FieldView** Parallel is supported for this operating system, 2. That more than 1 processor (CPU) is present, and 3. Whether **FieldView** is licensed for parallel operation. The text shown under the *Local Parallel* option on your Data Input menu will indicate what FieldView has found in these respects, on systems which currently support local parallel operation in **Figure 12** below:



Runs least of either number of local CPUs or licensed processes [shared memory]

Figure 12 Running Local FieldView Parallel

Local parallel

All FieldView Users are licensed for parallel operation. (Evaluation licenses are limited to 3 processes.) Local parallel lets you run **FieldView** Parallel using whatever level of licensing you have. For best performance, **FieldView** Parallel is designed to run only 2 parallel processes (-np 3) on systems with only 2 CPUs (i.e. dual-core systems), where you can expect a 2X factor improvement in performance for most operations when running Local parallel. (If a system has only 1 CPU, the 'Local parallel' option will be grayed out.)

On systems with a greater number of logical cores, you will be able to run up to your licensed number of cores. All supported Customers are licensed with at least 8 cores. By default, Local Parallel will be limited to lesser of either the number of parallel licenses or the number of locally available processors (or CPUs). This is done in order to avoid the situation where someone with a 32 processor parallel license (pfv32) would try to start all 32 processes on a local system that only has 8 processors available.

Additional Comments and Limitations

Full support is provided for RESTARTS and FVX using parallel processing. This includes restarts written with previous versions of FieldView which used differently named parallel options.

FVX support is also provided; the `server_config` argument for this is "Local parallel". The syntax for the **FVX** `read_dataset` command is:

```
server_config = 'Local parallel'
```

Local **FieldView** Parallel operation is one possible way to run **FieldView** Parallel. It is also possible to create a custom server configuration file to match your working environment. Additional details on how to do this are documented in the **User's Guide**, and in the **Installation Guide**.

AcuSolve Direct Reader

(www.altair.com)

This direct reader reads AcuSolve™ .Log files. The source code uses the latest libraries provided by Altair, enabling full support of all features. In order to maintain backward compatibility, **FieldView** restarts based on the AcuSolve Direct Reader prior to the AcuSolve Version 1.8B release will use the former naming convention for the `eddy_frequency scalar, dissipation rate`.

This is a Plugin Toolkit Reader; the necessary files are installed as part of the normal **FieldView** installation on the following supported platforms: Windows 64-bit and Linux 64-bit. The standard location for the reader source and the supporting libraries will be at `FV_HOME/bin/plugins` and `FV_HOME/bin/plugins/lib` respectively. Alternately, these files can be placed in a different directory, as specified using the environment variable `FV_PLUGINS`.

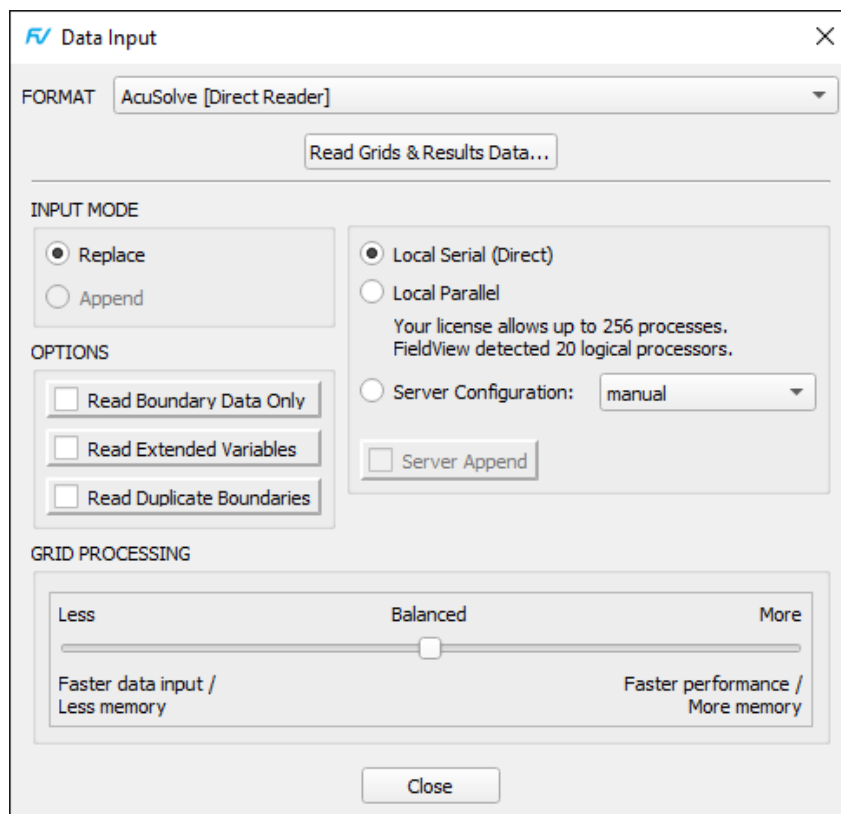


Figure 13 AcuSolve [Direct Reader] Panel

The AcuSolve direct reader permits the reading of additional nodal output, including the extended variables which may be stored during AcuSolve solver runs.

Extended variables from AcuSolve include gradients of almost all scalar and vector variables. Consequently, it will take longer to read all of the additional supported variables. In order to avoid compromising performance of simple postprocessing sessions, reading of the gradients will be enabled by setting the environment variable `FV_ACUSOLVE_GRAD`.

The reader also provides the option of reading duplicate boundaries. When building a model for AcuSolve, many different types of commands utilize surfaces as input. For example, the application of boundary conditions is done on a surface, a request for computing forces is done on a surface, identification of radiation parameters for view factor calculation is done on a surface, etc. Each of these inputs creates a different set of output that gets stored in the AcuSolve database. However, these surfaces are often times the same. In that case, it is possible to eliminate the duplication of surfaces at read time by checking for uniqueness. If a given surface is unique, it is imported by **FieldView**. If it is not unique, then only a single instance of the surface is imported by **FieldView** and the duplicates are ignored. Both toggles are off by default. With these changes, and also thanks to other optimizations in the AcuSolve Direct Reader, the time needed to read an AcuSolve case has been divided by 4.

The reader also provides support for **FieldView** regions. This allows you to be able to separately visualize and control the fluid and solid regions which AcuSolve is capable of creating. Creating separate regions for fluid and solid will make old restarts fail since **FieldView** does allow the region count to be different from the one in the restart. In order to get older restarts to work with the reader, support for regions can be disabled using the environment variable, `FVREG_ACUSIM_OFF`.

The AcuSolve Direct Reader supports AcuSolve's latest multiphase capabilities, giving access to a number of new variables, for visualizing phases or species. **Figure 14** shows a multiphase simulation from AcuSolve, using a VOF model. The Coordinate Surface is being colored by the volume fraction of water at each point.

The AcuSolve Direct Reader supports mid-step mesh displacement, using the following environment variable:

```
FV_ACUSOLVE_PREFER_MIDSTEP
```

If this environment variable is set to any value prior to the **FieldView** startup, then the reader looks for mid-step mesh displacement. If this is found, it is used to displace the mesh, and also returned as a variable called `"mesh_displacement"`. The following message is printed to the console:

```
Displacing mesh coordinates using mid step mesh displacement field.
```

If this environment variable is set to any value prior to the **FieldView** startup, and mid-step mesh displacement is not found, then the reader looks for end step mesh displacement. If found, this is used to displace the mesh, and also returned as a variable called `"mesh_displacement"`. The following message is printed to the console:

```
Mid step mesh displacement field not available, displacing  
coordinates using end step mesh displacement field.
```

If the environment variable is not present (or unset), then the reader only looks for end step mesh displacement. If found, this is used to displace the mesh, and also returned as a variable called `"mesh_displacement"`. The following message is printed to the console:

```
Displacing coordinates using end step mesh displacement field.
```

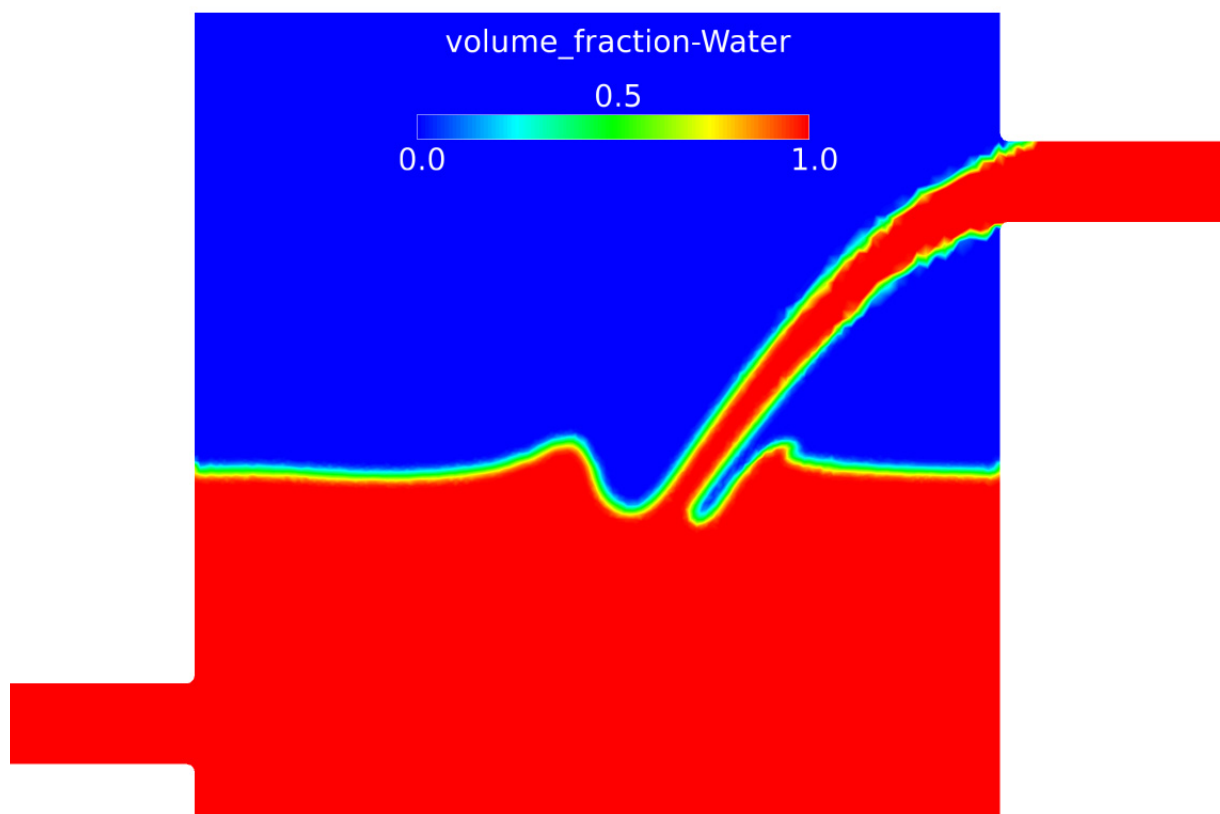


Figure 14 Multi-phase AcuSolve simulation visualized in FieldView

Changes made interactively to the "Read Extended Variables" and "Read Duplicate Boundaries" toggle buttons on the AcuSolve Direct Reader panel (see [Figure 13](#)) are retained within the **FieldView** session and saved as well as preferences, which means **FieldView** will remember them.

Known limitations

There is no direct reader for Mac OS X or 32-bit Linux or 32-bit Windows. It is possible to use the direct reader in client-server mode using the supported server platforms (64-bit Linux and 64-bit Windows).

CGNS

(cgns.github.io)

A plugin Reader has been added for CGNS to read both the structured and unstructured formats.

CGNS Unstructured/Hybrid Reader

(cgns.github.io)

We recommend that this reader be used to read CGNS Unstructured datasets, replacing the pre-existing CGNS Unstructured Reader. This reader can also be used to read CGNS Structured datasets, subject to certain limitations. A major distinction of this reader is that it supports both ADF and HDF5 file formats and libraries. Another key distinction is that streamline wall marking, which prevents streamlines from passing through wall boundaries, is implemented with this reader.

The CGNS Unstructured/Hybrid Reader for **FieldView** has been updated with CGNS library version 3.3.1.

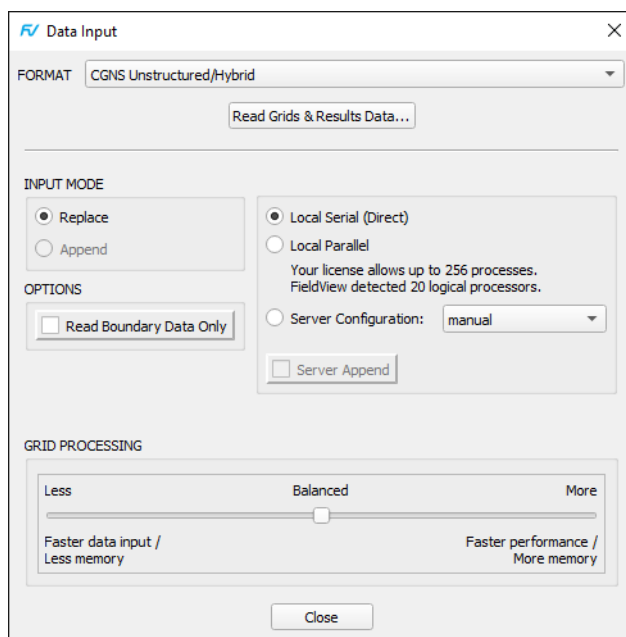


Figure 15 CGNS Unstructured/Hybrid Data Input Panel

To read a CGNS Unstructured file, start by selecting the CGNS Unstructured/Hybrid entry on the Data Input pulldown menu. On the CGNS Unstructured/Hybrid panel, click Read Grids & Results Data... When you do this, you will see a file browser which will let you navigate to the location of the `.cgns` file that you wish to read.

Note: Prior to the introduction of the CGNS Unstructured/Hybrid Reader described in the previous section, FieldView had two separate readers for unstructured and structured CGNS. These readers are only compatible with older versions of CGNS and we recommend that the more recent CGNS Unstructured/Hybrid reader be used instead

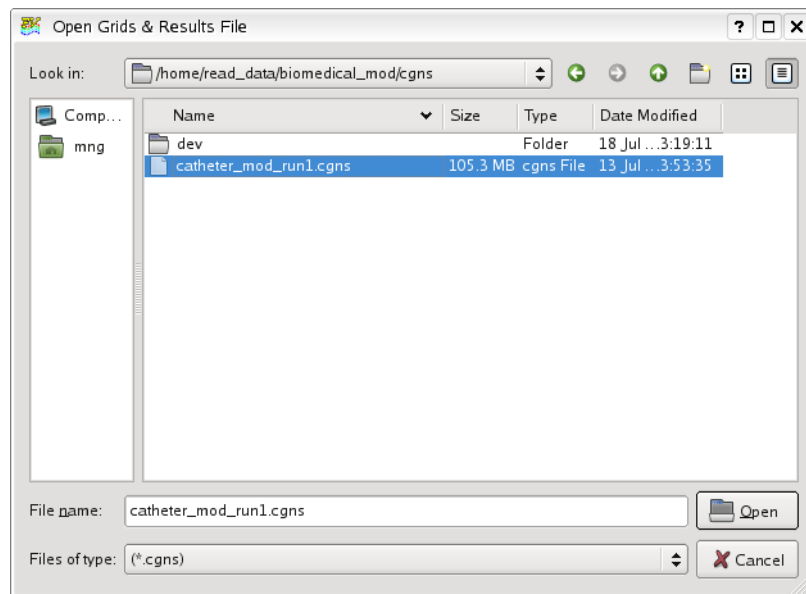


Figure 16 CGNS Unstructured/Hybrid File Browser

The local file browser will automatically apply a filter, *.cgns, to more easily locate files of this format type.

CGNS files written using either nodal or cell-centered formats can be read. For the case of cell-centered data, **FieldView** does nodal interpolation at the time that the dataset is read. Also, full support is available to read standard and arbitrary polyhedral cells.

Known limitation

- This reader can be used to read CGNS Structured datasets. However, it will not be able to provide and structured data context - it will not be possible to create computational surfaces, probe for IJK values. If structured boundaries are defined using a .fvbnd file, they will not be available.

FIDAP

(www.ansys.com)

This data reader will read in files from the FIDAP program from FLUENT. **FieldView** reads the FDNEUT files from Version 7 or later of FIDAP. The FDNEUT file needs to be exported from FIDAP for input into **FieldView** using the FICONV command.

FIDAP transient NOMOMENTUM FDNEUT files can now be read. The velocity field for the first time step will be applied to all time steps.

When 2D FIDAP data is read into **FieldView**, it is 'extruded' by a small amount that is a function of the XY extent of the data, but at least 1.0e-5 units. This extrusion will be just enough for **FieldView** to properly display the data, but will not affect its 2D appearance.

This reader is currently accessed from the fly-out menu in the Data Input pulldown menu, when you select the More Readers... option.

FLOW-3D® Animation Data

(www.flow3d.com)

This direct reader can read Animation Data, when present, in a `flsgrf.dat` file. This is a Plugin Toolkit Reader; the necessary files will be installed as part of the normal **FieldView** installation procedure on the following supported platforms: WINDOWS 64bit, LINUX 32 and 64bit. The standard location for the reader source and the supporting libraries will be at `FV_HOME/bin/plugins` and `FV_HOME/bin/plugins/lib` respectively. Alternately, these files can be placed in a different directory, as specified using the environment variable, `FV_PLUGINS`.

FLOW-3D® Restart Data

(www.flow3d.com)

This direct reader can read Restart Data, when present, in a `flsgrf.dat` file. This is a Plugin Toolkit Reader; the necessary files will be installed as part of the normal **FieldView** installation procedure on the following supported platforms: WINDOWS 64bit, LINUX 32 and 64bit. The standard location for the reader source and the supporting libraries will be at `FV_HOME/bin/plugins` and `FV_HOME/bin/plugins/lib` respectively. Alternately, these files can be placed in a different directory, as specified using the environment variable, `FV_PLUGINS`.

FLOW-3D®

(www.flow3d.com)

There are two ways to read the standard FLOW-3D® `flsgrf.dat` files into **FieldView**. The direct readers used to read either the FLOW-3D® Animation Data or the FLOW-3D® Restart Data have been previously described.

The legacy FLOW-3D® Reader, which only reads RESTART data, is accessed from the fly-out menu in the Data Input pulldown menu, when you select the More Readers... option. This data reader will read in native `flsgrf.dat` files from the FLOW-3D® program from Flow Science (Los Alamos, NM) for Version 6 or later. Note however that there are known problems with FLOW-3D® VERSION 9.1, so it is recommended that `flsgrf.dat` files generated using the latest FLOW-3D® release be read using the Plugin Toolkit reader described above.

This data reader may use the `FLSINP` file used when running FLOW-3D®. The presence of the `FLSINP` file is not required and may not provide meaningful information for newer files. Support is provided mainly for backward compatibility. The `FLSINP` file will be checked for in the same directory as the `FLSGRF` file. This file is used to control how the data from the `flsgrf.dat` file is read. The following information is read from the `FLSINP` file:

- IGRP2 - If equal to 1, read in spatial and solidification data. Only solidification data at the end of the calculation is loaded
- IBFF - If equal to 0, iblanks will not be used

If equal to 1, iblanks will be based on the volume fraction
If equal to 2, iblanks will be based on the volume and fluid fractions
If equal to 3, iblanks will be based on the complement on the volume fraction (used for plotting wall quantities)

BVALVF - blank out this node when $VF(IJK) < BVALVF$

BVALF - blank out this node when $F(IJK) < BVALF$

The defaults are as follows:

IGRP = 1, IBFF = 0, BVALVF = 0.5, BVALF = 0.5

If the FLSINP file is not found, a warning message will be presented, and your data will be scaled by the default values.

The I, J, and K boundaries of the grid are treated as walls.



Note: The legacy FLOW-3D® reader will work with FLOW-3D® Version 9.0 files. It should handle multiple grids or blocks.

Note: FLOW-3D® Animation data is supported with the direct Plugin reader ONLY.

See [Free Surface Flow Tutorial](#) in [Chapter 4](#) of the **User's Guide** for more information on working with FLOW-3D® data.

ANSYS-Fluent CFF [Direct Reader]

(www.ansys.com)

This direct reader can read CFF files from ANSYS-Fluent. These files can be recognized to their .cas.h5/.dat.h5 extensions. Legacy .cas/.dat Fluent files can still be read with [FLUENT cas/dat Direct Reader](#), but that format has been superseded by the new HDF5 based CFF format.

The ANSYS-Fluent CFF reader is based on a Software Development Kit (SDK) provided and maintained by ANSYS through a partnership between ANSYS, Inc. and Tecplot, Inc. Unfortunately, this SDK doesn't yet support MPI parallel read operations. Therefore, the new ANSYS Fluent CFF reader is limited to Local Serial and Client-Server Serial read operations.



Note: All CFF compression levels are supported. Files will be uncompressed on the fly at read time.

Variables names are obtained from a dictionary file (ANSYS_FLUENT.xml) provided with the FieldView installation. If instead you prefer to have FieldView use a different file, such as the one provided with a specific version of ANSYS-Fluent, set the environment variable CFF_VARIABLE_CONFIG_DIR to have it point to the directory where the alternate ANSYS_FLUENT.xml file is located.

Note that variable names will differ slightly from the ones obtained with the legacy cas/dat reader. They will now match exactly variable names as seen in the ANSYS-Fluent interface.

This reader is perfect for checking simulations from ANSYS-Fluent. But if you need the best performance or know you're going to load this simulation in FieldView multiple times, we encourage you to use the ANSYS-Fluent export to the FieldView Unstructured format (FVUNS).

Known Limitations

- Only 3D results are supported.
- Particles computed with the DPM model do not get imported as Particle Paths. Use the FieldView Particle Paths (FVP) export from ANSYS-Fluent instead.
- No parallel (only Local Serial or Client/Server Serial). The ANSYS SDK does not provide support for MPI parallel read

To read FLUENT `.cas.h5` and `.dat.h5` files, start by selecting the ANSYS-Fluent CFF [Direct Reader] entry on the Data Input pulldown menu. Then on the panel, click Read Grid Data... When you do this, you will see a file browser which will let you navigate to the location of the `.cas.h5` file that you wish to read, to be followed by the `.dat.h5` file.

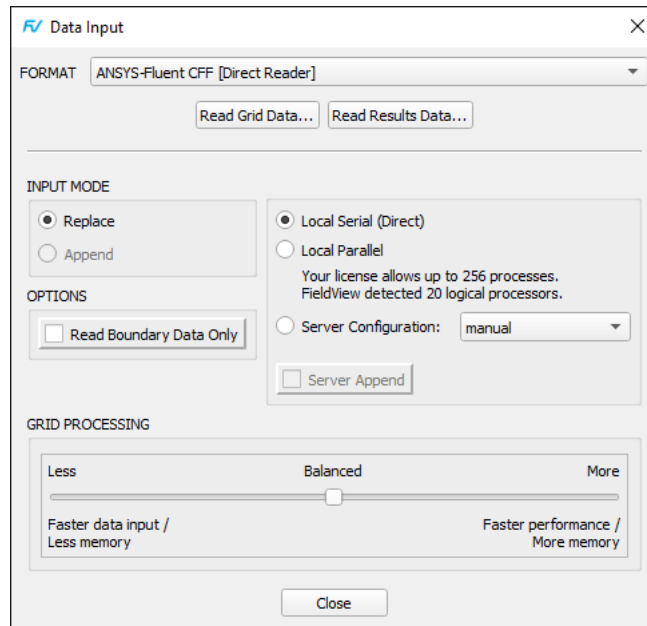


Figure 17 ANSYS-Fluent CFF [Direct Reader] Data Input Panel

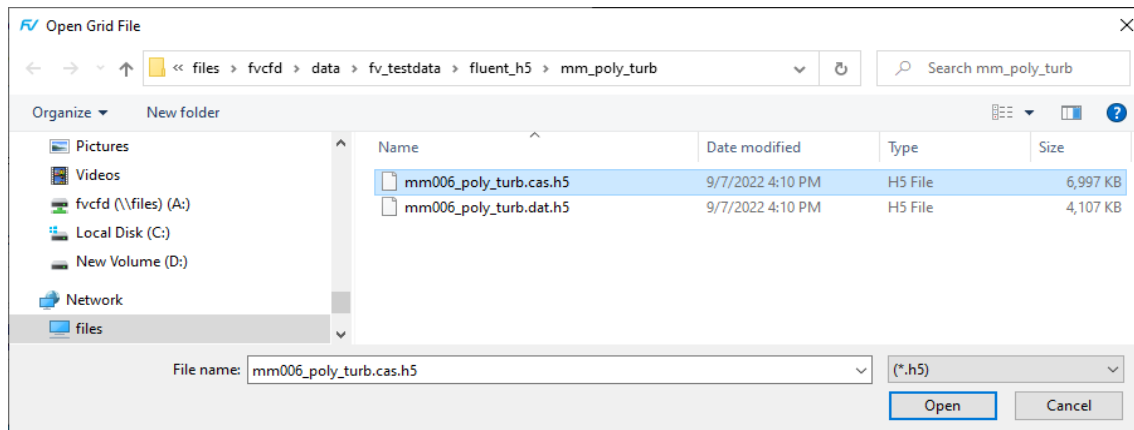


Figure 18 ANSYS-Fluent CFF [Direct Reader] File Browser for .cas.h5 files

After you select the file you want to read, click the Open button. FieldView will prompt you if multiple .cas.h5 files exist, asking if you would like to read the data as transient. (Note that ANSYS-Fluent exports transient data in compliance with FieldView's transient naming convention as described in the Transient Data section of our Reference Manual [PDF]. After the prompt is answered, a second file browser is launched to allow you to read the .dat.h5 file(s.) Select the corresponding results file, and click on the Open button to complete reading of the dataset.

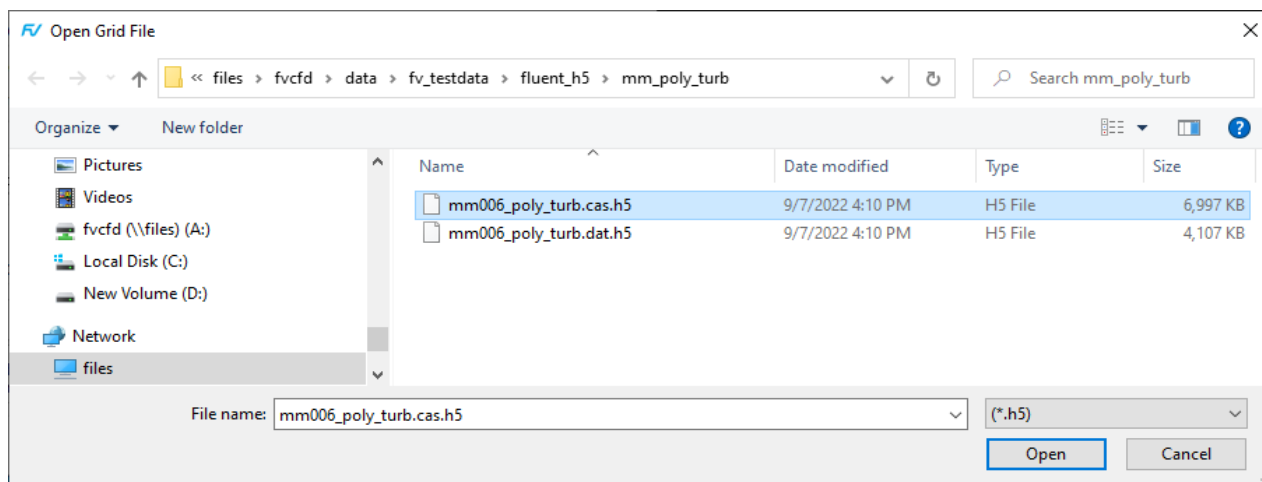


Figure 19 ANSYS-Fluent CFF [Direct Reader] File Browser for .dat.h5 files

It is also possible to select a different .dat.h5 file (of a name not matching the grid file name) or read a steady grid, with transient results. At this point, if you cancel the file browser, you will return to the Input panel shown above in [Figure 17](#).

FLUENT cas/dat Direct Reader

(www.ansys.com)

This legacy Direct reader FLUENT .cas and .dat files remains an input option, but is deprecated to the recommended **ANSYS-Fluent CFF [Direct Reader]**. It features many improvements compared to the FLUENT Direct Reader including faster read times, support for arbitrary polyhedra, the ability to read .cas and .dat files with different names and more consistent variable name matching with FLUENT. All currently known data input issues are resolved with this reader. Also, restarts created on FLUENT datasets based on the pre-existing FLUENT Direct Reader can be applied after reading data with the FLUENT cas/dat Direct Reader using the Complete Restart, No Data Read option.

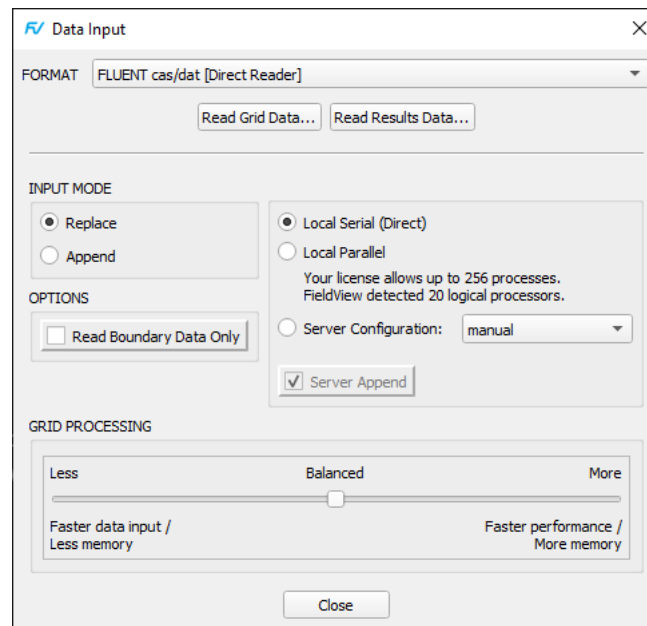


Figure 20 FLUENT cas/dat [Direct Reader] Data Input Panel

To read FLUENT .cas and .dat files, start by selecting the FLUENT cas/dat [Direct Reader] entry on the Data Input pulldown menu. On the FLUENT cas/dat [Direct Reader] panel, click Read Grid Data... When you do this, you will see a file browser which will let you navigate to the location of the .cas file that you wish to read.

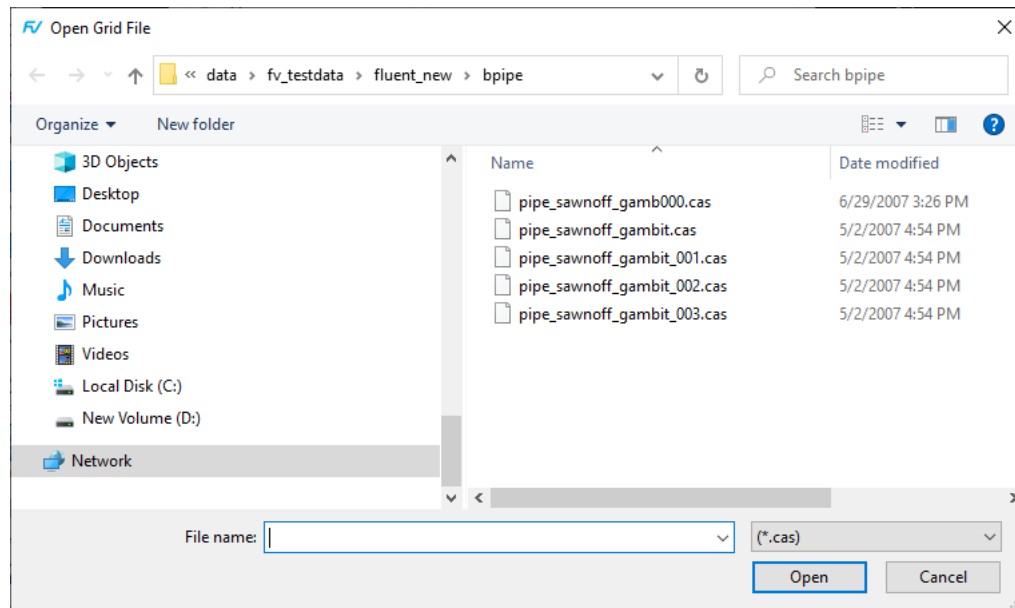


Figure 21 Fluent cas/dat File Browser for .cas files

After you select the file you want to read, click the Open button. When this is done, a second file browser is launched to allow you to read the .dat file. **FieldView** automatically attempts to locate the matching .dat file for the .cas file which has already been read. If a matching file is found, it will show up in the Filename section of the file browser. You can read this data file by clicking the Open button.

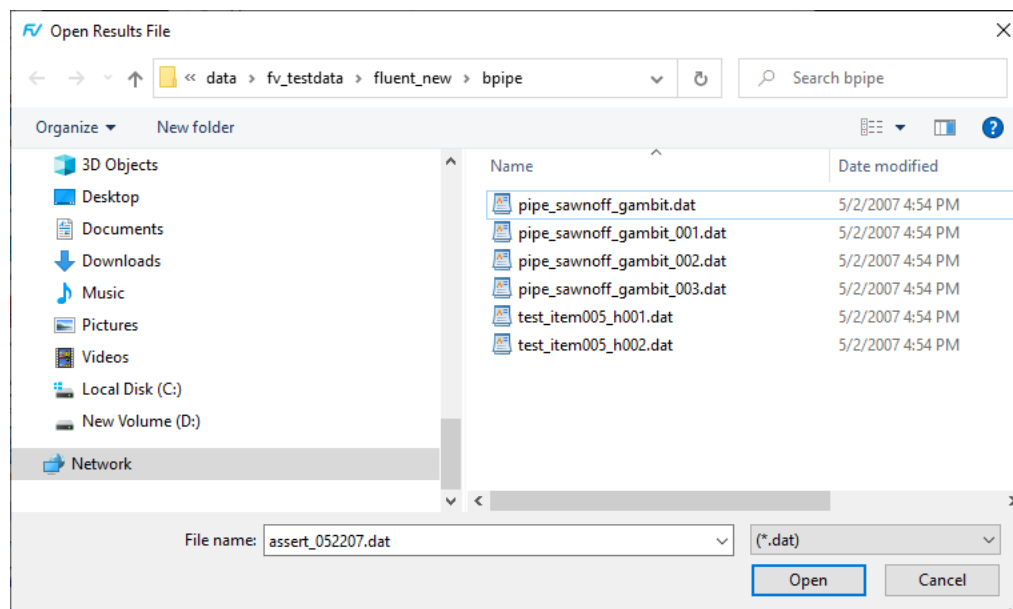


Figure 22 Fluent cas/dat File Browser for .dat files

It is also possible to read a different `.dat` file - the former restriction on being able to read a matched `.cas/.dat` pair has been removed. You can use the file browser to locate a `.dat` file and read it at this point. If you cancel the file browser, you will return to the FLUENT cas/dat Data Input panel (see [Figure 20 FLUENT cas/dat \[Direct Reader\] Data Input Panel](#)).

This reader can also be used to read FLUENT mesh (`.msh`) files, which are very similar to FLUENT `.cas` files, but only contain the mesh information. Therefore, no functions will be available after the dataset is read.

To read a FLUENT `.msh` file, start by selecting the **FLUENT cas/dat [Direct Reader]** entry on the Files > Data Input pulldown menu. On the FLUENT cas/dat [Direct Reader] panel, click Read Grid Data... When you do this, you will see the Open Grid File browser. The default File of type: filter is set to "`(* .cas)`"; change this to "`All Files (*)`" and navigate to the location of the `.msh` file that you wish to read. After you select the file you want to read, click the Open button. After this is done and the Open Results File browser is launched, click Cancel.



Note: The legacy FLUENT cas/dat Direct Reader does not support compressed files; in order to be read, files must be uncompressed.

We expect that it is possible to read a restart which was based on the pre-existing FLUENT direct reader onto a FLUENT dataset which was read using the FLUENT cas/dat direct reader. Restarts of the type Complete, No Data Read and Current Dataset will work in this way. Variable name mapping from the older names to the newer ones, thereby preserving the restart contexts, is done automatically.

Known changes in behavior

The classification of which variables are considered boundary-only, volume variables or both has changed. If you use the new reader with a restart created with the old reader, we will automatically convert boundary-only to volume and vice versa as needed. (The exception is if the old restart has a boundary-only variable such as heat flux on a non-boundary surface.)

FLUENT Direct Reader

(www.ansys.com)

This legacy Direct reader FLUENT `.cas` and `.dat` files remains an input option, but is deprecated to the recommended **ANSYS-Fluent CFF [Direct Reader]**. The legacy reader is not available for the macOS platform. Known Limitations

Although FLUENT can read `.gz` compressed files, this reader lacks this capability. It is necessary to unzip compressed files in order for them to be read correctly. The base names for the `.cas` and the `.dat` file must match. Note that the export from FLUENT to the **FieldView** Unstructured format is fully supported. Because some variables are derived by FLUENT, they may only be available in the exported files and not the direct reader.

FLUENT Universal

(www.ansys.com)

This data reader will read in the Universal files from the FLUENT program (Version 4.2 and below) from Fluent, Inc. Cell types are preserved and all of the dependent variables listed in the Solution Data Section of the Universal file are stored in **FieldView**. In addition, wall cells and porous cells are marked during input. This file format has been superseded by the FLUENT export to **FieldView** Unstructured File Format. Except for legacy datasets, we recommend that you do not use this format as later FLUENT features are not supported.



Note: This is not a native FLUENT dataset reader. FLUENT datasets currently need to be exported from FLUENT in order to be read into **FieldView**. The FLUENT UNIVERSAL file format is a very old format, which is no longer in use with current FLUENT products.

This reader is currently accessed from the fly-out menu in the Data Input pulldown menu, when you select the More Readers... option.

FLUENT/UNS (and RAMPANT)

(www.ansys.com)

This reader is *only* used to read in the **FieldView** Case and Data files from the FLUENT/UNS program from Fluent, Inc (Version 5.2 or later) or the RAMPANT program (Version 2.1 or later). The **FieldView** Case + Data format is one of the available FLUENT export options, and has been the default export for versions of FLUENT prior to FLUENT 6.1.

2D FLUENT/UNS data *must* be exported from FLUENT (for versions before FLUENT 6.2) using the **FieldView** Case + Data option. The **FieldView** Unstructured Export option from FLUENT supports 2D data with the release of FLUENT 6.2. The RAMPANT-FLUENT/UNS reader 'extrudes' the 2D data by a small amount that is a function of the XY extent of the data, but at least 1.0e-5 units. This extrusion will be just enough for **FieldView** to properly display the data, but will not affect its 2D appearance. **FieldView** will recognize a correctly named sequence of 2D **FieldView** Case + Data files as a transient case.



Note: This is not a native FLUENT data reader. FLUENT datasets currently need to be exported from FLUENT in order to be read into **FieldView**.

This reader is currently accessed from the fly-out menu in the Data Input pulldown menu, when you select the More Readers... option.

FV-UNS Data Input (Native FieldView Unstructured Format)

The native **FieldView** Unstructured (FV-UNS) format is widely used to read unstructured data into **FieldView**. Several of the major commercial solvers support exporting or translating their data into the FV-UNS format. These solvers will be covered after general comments about the FV-UNS format.

Three types of FV-UNS formats are available: Binary, Unformatted and ASCII. All three formats are described in [Appendix D](#) of this **Reference Manual**.



Cylindrical Note: When cylindrical coordinates are specified using an FVREG file (Region definition), then XYZ information is presented to you in **FieldView** as RTX/Z (Radius, Theta, X/Z) data. That is, Coordinate surface sliders, the 2D plotting system, Point Probing, Streamline seeding, Exporting, etc. will be in RTX/Z coordinates instead of the usual XYZ coordinates. See [Chapter 3](#) of this **Reference Manual** for more information. Note that the Grid information in the FV-UNS file itself is still required to be in XYZ coordinates.

*Transient **FieldView** Unstructured Data*

FieldView supports transient FV-UNS datasets. Transient FV-UNS data requires one file per time step. Hence, four time steps require four FV-UNS files.

This is different from other solvers (CFX-4 .dmp file, for example), where *all* data resides in a single file. One advantage of this technique is that it can be far quicker to access a specific time step during visualization.

A dataset will automatically be recognized as transient if each file has a time step number embedded in its name using a convention described in [Transient Data](#). The actual extension used is not important, but “*.uns” or “*.fvuns” is recommended, and typically used.

If any one of the files of the series is chosen, **FieldView** will find other files with the same file naming convention in the directory and present you with the option of treating the set as transient. If agreed to, the chosen time step will be loaded into memory, and the remaining filenames stored for reference. Other time steps can be accessed through the Transient Data Controls panel (see [Chapter 14](#) of **Working with FieldView** for more details).

See [Vortex Shedding Tutorial](#) in [Chapter 8](#) of the **User’s Guide** for more information on working with transient FV-UNS data.

*Face Data for **FieldView** Unstructured Data*

FieldView supports face data on boundary faces of FV-UNS data. See [Appendix D](#) of this **Reference Manual** for complete information about the FV-UNS file formats.

Ensignt Reader

The Ensignt reader has been optimized for transient datasets where grids are not moving, making rebuilding of the grid data unnecessary when changing timesteps. This reader supports cases with moving grids, as well as a changing number of elements and nodes over time.

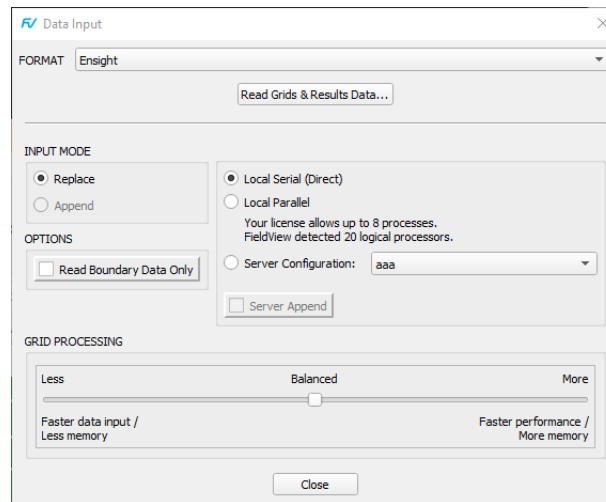


Figure 23 Ensignt Data Input Panel

To read data of this format, select the Ensignt entry from the Data Input pulldown menu and click Read Grids & Results Data... When you do this, you will see a file browser (using the *.case filter) as illustrated below.

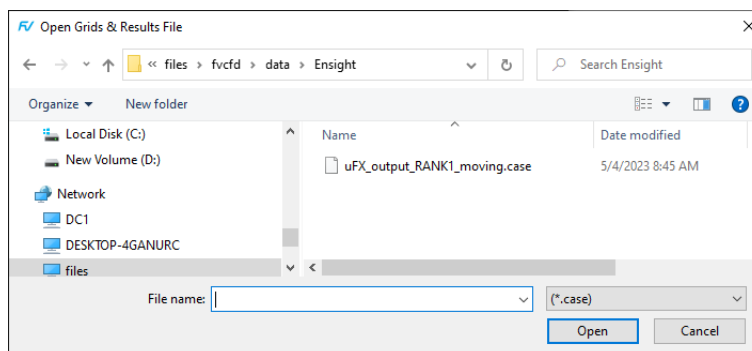


Figure 24 Ensignt File Browser for ENSIGHT Gold files

The local file browser has two filters, *.encas and *.case, to more easily locate files of ENSIGHT Gold format types. The Reader can handle ENSIGHT Gold files for both nodal based data (scalar per node) and cell centered data (scalar per element). For cell centered data, nodal interpolation is done at read in time by **FieldView** using interpolation methods which are consistent with those used by commercial solver codes, such as FLUENT (Ansys) or STAR-CCM+ (CD-adapco), for standard and arbitrary polyhedral cell types.

Known limitations

For transient cases, ENSIGHT Gold files must be written in the format where a set of files have the time step embedded in the name for each time step, for each of the scalars and vectors written. This is how the exports are typically written from commercial solver codes such as AcuSolve, FLUENT and STAR-CCM+.

Automation for the Ensignt completely supported with Restarts, **FVX** and Python scripting.

Tecplot 360 Reader

The Tecplot reader has been optimized for transient datasets where grids are not moving, making rebuilding of the grid data unnecessary when changing timesteps. This reader supports cases with moving grids, as well as a changing number of elements and nodes over time.

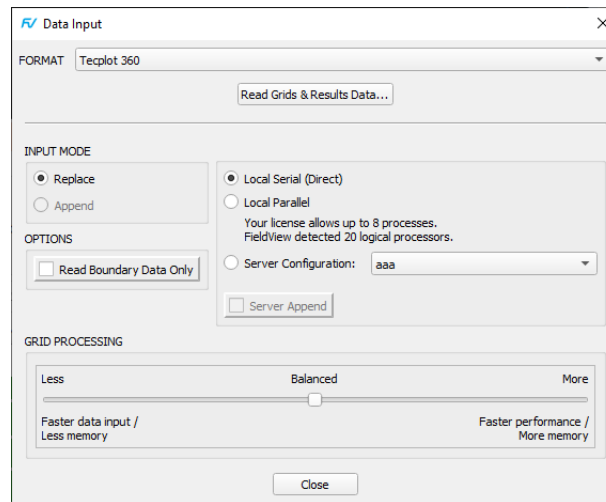


Figure 25 Tecplot 360 Data Input Panel

To read a Tecplot 360 file, select the Tecplot 360 entry from the Data Input pulldown menu, and click Read Grids & Results Data... When you do this, you will see a file browser, filtering for *.plt, files as as illustrated below.

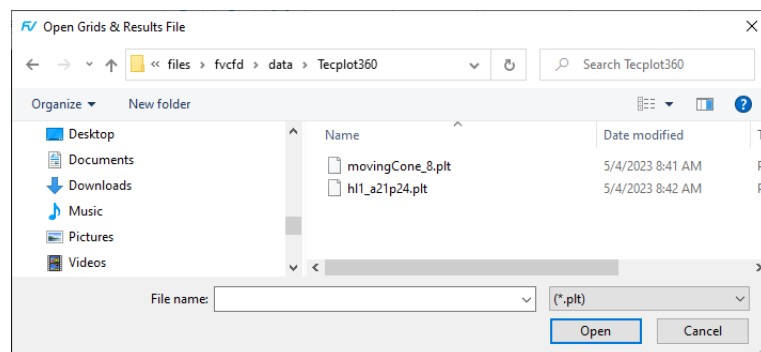


Figure 26 Tecplot 360 File Browser

Known changes in behavior and limitations

- Solution times for transient data files cannot be read from Tecplot files. To create streaklines or to use the solution time **escape sequence** `%T` for annotation, it is possible to set a solution time value with the Use Delta Time function on the Transient Controls panel.
- Grid coordinates in Tecplot files do not need to be written using standard names or locations. Special handling has been included to try to correctly interpret variations on how this data is stored.

Automation for the Tecplot 360 Reader is completely supported with Restarts, **FVX** and Python scripting.

HAVOC

(www.corvidtec.com)

This reader is based on a proprietary format, and is intended to read results from a CFD hydrocode called CTH. This code, which is available from Sandia National Laboratories, is a multi-material, large-deformation, strong shock wave, solid mechanics code. It is capable of handling multi-phase, elastic-viscoplastic, porous and explosive materials. CTH has several material models appropriate for strong shock, large deformation calculations. The reader is accessible in **FieldView** from the standard pull-down list for reading data and is entitled HAVOC.

LS-DYNA d3plot Direct Reader

(www.lstc.com)

The LS-DYNA d3plot Direct Reader is the recommended reader for LS-DYNA datasets. It features many improvements compared to the pre-existing LS-DYNA State Database Direct Reader and resolves all known bugs, including some interpolation problems present in earlier versions of third party routines this reader was based on.

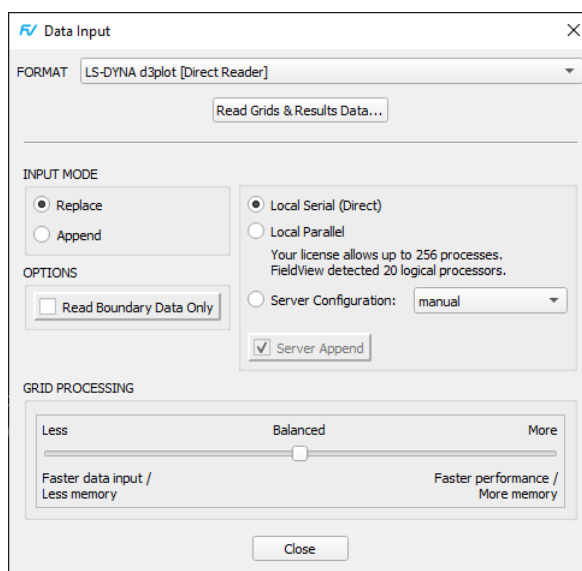


Figure 27 LS-DYNA d3plot [Direct Reader] Data Input Panel

To read an LS-DYNA d3plot file, select the LS-DYNA d3plot [Direct Reader] entry from the Data Input pulldown menu. On the LS-DYNA d3plot [Direct Reader] panel, click Read Grid & Results Data... When you do this, you will see a file browser, as illustrated below.

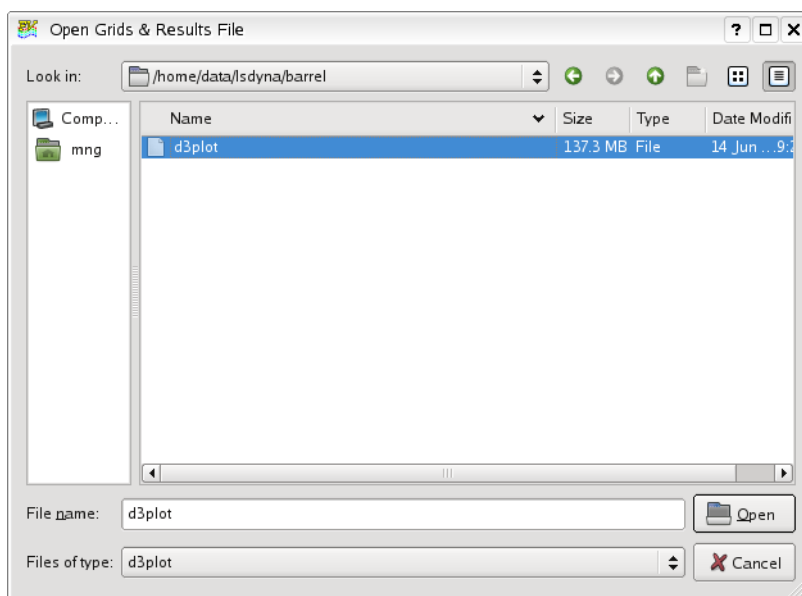


Figure 28 LS-DYNA d3plot [Direct Reader] File Browser

The local file browser has a filter, `d3plot`, to more easily locate files of LS-DYNA d3plot format.

Known changes in behavior

Strain energy is now interpreted as a boundary-only variable. For older restarts, the conversion from volume to boundary only data is automatically done.

In applying an old restart with thresholding turned on, it may be possible that surfaces will show up with holes. The holes are actually a result of a very small difference resulting from changes to interpolation. These holes (see [Figure 29 LS-DYNA interpolation issue with Thresholding](#)) can be removed by interactively pushing the Threshold Function slider to the range extents.

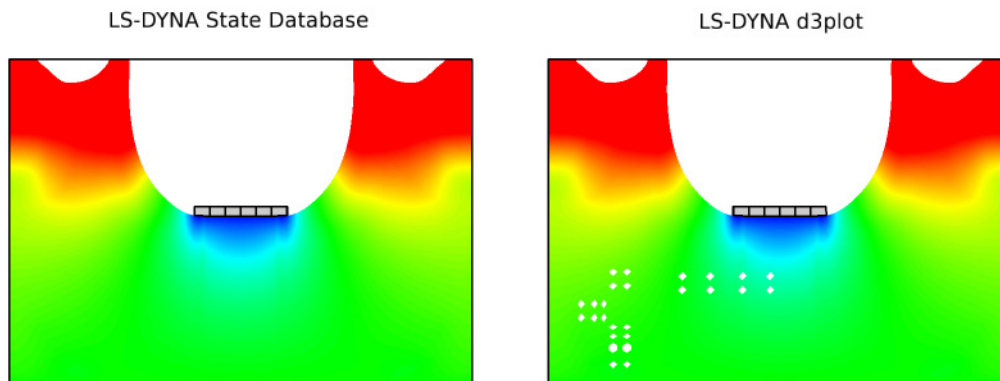


Figure 29 LS-DYNA interpolation issue with Thresholding

LS-DYNA

(www.lstc.com)

A direct reader is now available to read LS-DYNA d3plot files. This is a Plugin Toolkit Reader; the necessary files will be installed as part of the normal **FieldView** installation procedure on all supported platforms with the exception of MAC OS X and LINUX IA64. The standard location for the reader source and the supporting libraries will be at `FV_HOME/bin/plugins` and `FV_HOME/bin/plugins/lib` respectively. Alternately, these files can be placed in a different directory, as specified using the environment variable, `FV_PLUGINS`.

The supported element types for this reader include triangles and quads used for thin shells, and tets, pyramids and hexes used for volumes and thick shells.

A **FieldView** Formula Restart file has been provided to automatically generate all tensor invariant representations for the shear and stress tensor data. This is available on the **FieldView** DVD (see `/misc/lsdyna/lsdyna_formulas.frm`).

Known Limitations

There are some element types which can be used in LS-DYNA simulations which are not currently supported. They include: beams, trusses, springs, lumped masses and dampers. It is our expectation that we will be able to read datasets which contain these types of elements - however, they will not be available for postprocessing once the data is in **FieldView**.

We will not be able to read datasets containing smooth particle hydrodynamics (SPH) data.

There is currently no support to read time history plot (.d3dhd.t) files.

NPARC/WIND

(www.grc.nasa.gov/WWW/wind/)

The NPARC/WIND reader within **FieldView** is implemented as a standard plugin. If you have your own NPARC WIND or WIND Unstructured reader source, or if you have obtained a plugin reader from the NPARC Alliance, you can exchange it with the reader source distributed with **FieldView**. This infrastructure change also enables the use of WIND US (unstructured) reader plugin supplied by the NPARC Alliance, resolving internal issues. Please contact **Tecplot Inc.** or the NPARC Alliance for additional details concerning the WIND US (unstructured) reader.

The NPARC/WIND reader will accept input of structured data from grid (*.cgd) and data (*.cfl) files or from combined (*.cgf) files. Combined files are supported if you read them first as a grid file and then again as a results file. **FieldView** does not support unstructured NPARC/WIND data. This reader also supports grid subset selection in the case of multi-grid files and grid point increment (where, for example, every other point in the selected grids are read in). If an unstructured grid is selected to be read in, a warning will be issued in the xterm window where **FieldView** is running and the unstructured grid data will be otherwise ignored. For unsupported, cell centered results, a warning message is written to the xterm where **FieldView** is running. In addition, there is no support for transient data at this time.

Note: The **FieldView** NPARC/WIND reader supports user-specified variables in the results file.

Structured Boundary File

The NPARC/WIND Structured Boundary file is automatically created by the reader. Details about this file in particular and the Structured Boundary file in general can be found in [Appendix H](#) of this **Reference Manual**.

Solver variable conversion

The NPARC/WIND reader converts all of the solver variables from the SI (MKS) system of units to the English (FSS) system of units, when applicable. However, note that all grids are in units of inches. Any variables that the reader is unable to convert will print a message such as:

```
No conversion of anut for zone      1
```

in the console window in which **FieldView** is being run.

Constants and Formulas

A complete description of the handling of NPARC/WIND constants by the reader and the availability of formulas through the use of a Formula Restart file can be found in [Appendix N](#) of this **Reference Manual**. The formula restart file, `wind.frm`, is installed in the `/fvx_and_restarts` directory of the **FieldView** installation.

OpenFOAM

(www.openfoam.com)

A direct reader is available for OpenFOAM datasets created using the free, open source CFD software package produced by OpenCFD Ltd. Meshes generated from snappyHexMesh as well as other third party tools are supported. The capabilities of this reader include:

- the ability to read either reconstructed or decomposed (partitioned) data in either ASCII, binary or gzip compressed formats,
- the ability to read more than one partition per processor,
- the ability to list all time steps & select a starting time step,
- the automatic recognition of all boundary names, with the option to either read or skip partition-to-partition interfaces,
- the automatic recognition of all scalars (including tensor components) and vectors,
- streamline wall marking,
- grid subsetting,
- ability to read moving mesh cases,
- optimization for fast & accurate interpolation of standard cell types (hex, tet, pyramid & prism),
- full **FVX** & RESTART support,
- support for conjugate heat transfer problems and multi-region datasets (fluid, solid).

To read an OpenFOAM case, you can choose one of the following three methods:

- (a) select the system/controlDict file for the case, or
- (b) select any file in the top-level case directory (such as the fake files created by paraFOAM, or a log file), or
- (c) select the system/decomposeParDict file for the case

Selecting (a) or (b) will allow both partitioned and non-partitioned cases, but will read the non-partitioned representation if both are present. In general, **FieldView** will look first for either a single grid or recombined solution. If found, we read this version of the dataset. If the dataset is neither single grid nor recombined, then in cases (a) or (b), we look for a partitioned case.

It is possible to have only a grid file at the top level for case (a) or (b). In this case, if you only have results for partitioned data, this is what will get read.

To look explicitly for a partitioned case always, use (c).

To read an OpenFOAM case using **FVX**, the `data_format` option is set to `'openfoam'`.

When an OpenFOAM dataset is read, all time steps are available for selection instead of just the time steps that have solver results. In time step selection panel, there is an extra time step at the very beginning, usually time=0. This "initial" time step typically contains the initial values of many of the variables (these are often all zero, with the exception of boundaries). Since OpenFOAM only assigns

step numbers to solver time steps, we assign a step number to the initial time step, handling cases having either positive, or positive and negative, time steps.

The inclusion of the initial time step affects Restarts older than **FieldView** 13.2, since the number of time steps increases by 1. To use a Complete Restart saved from a previous version of **FieldView**, the initial time step can be ignored by setting an environment variable:

```
FV_OF_NO_INIT_COND = 1
```

It is possible to save read time and memory by setting the environment variable

```
FV_OPENFOAM_NO_PROC_BND
```

to any value, which will have **FieldView** avoid converting OpenFOAM processor boundaries to **FieldView** boundaries, when loading results with the "OpenFOAM [Direct Reader]".

The default behavior (in the absence of `FV_OPENFOAM_NO_PROC_BND`) is unchanged - the processor boundaries are converted into **FieldView** boundaries.

OVERFLOW-2

The NASA CFD code OVERFLOW-2 is used widely by the rotorcraft, spacecraft and fixed-wing communities for design and performance analysis. OVERFLOW-2 is an evolutionary code combining established structured-grid CFD algorithms with structural dynamics, overset grid methods, RANS/Hybrid-LES turbulence modeling, high-temperature gas dynamics and chemical/contaminant transport modeling. OVERFLOW-2 employs a modified PLOT3D solution file output format which will be described in detail below.

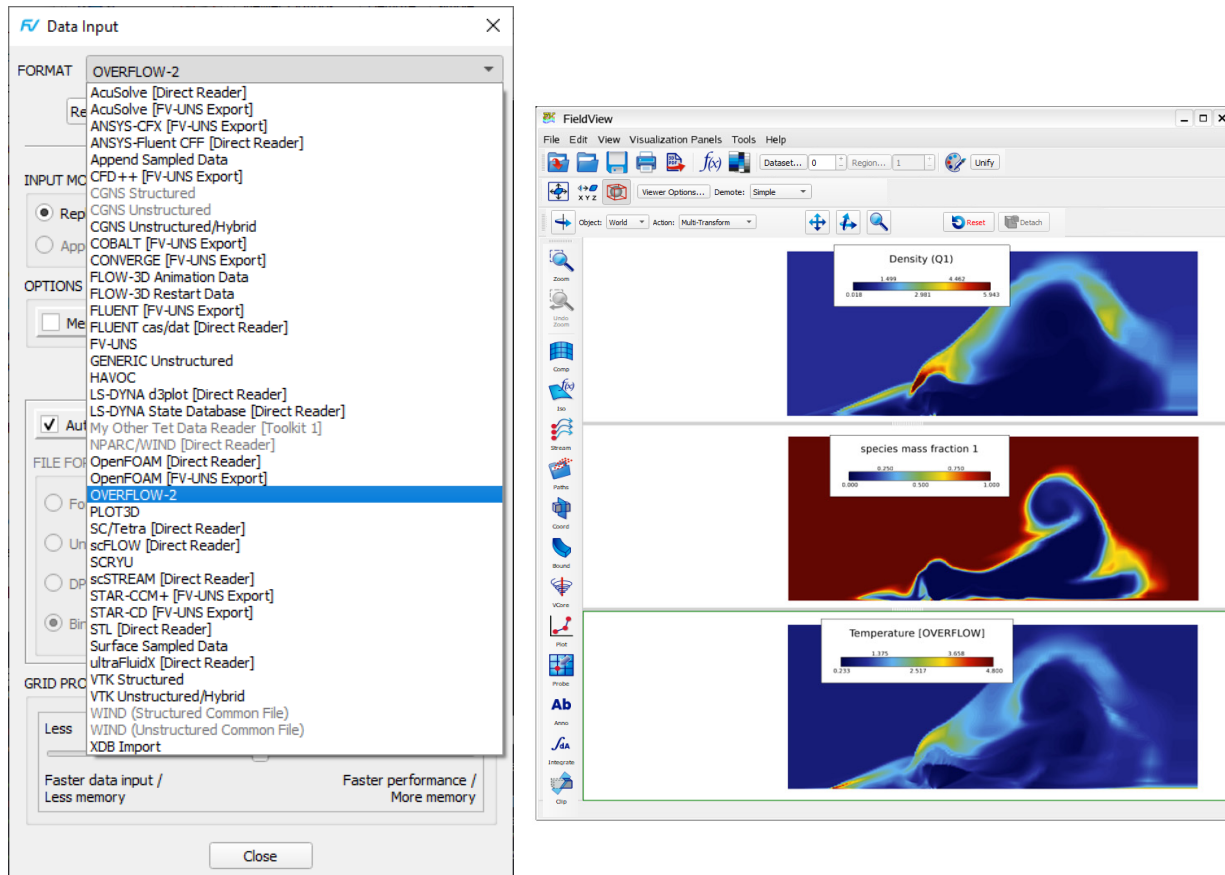


Figure 30 OVERFLOW-2 Direct Reader for FieldView

Reading Grid Files

Grid files match the standard PLOT3D file (Unformatted, single and double precision, big- or little-endian) and data (single and multi-grid, IBLANK) formats. PLOT3D grid files may be supplied by the user and will typically have the standard name `grid.in`. OVERFLOW-2 can also generate grids, or bricks, and these (`x.*`) files are always multigrid PLOT3D files with IBLANK information. Concerning grids, full **FieldView** compatibility has been included for

1. Boundary surfaces defined with `.fvbnd` files
2. Regions, defined with `.fvreg` files
3. Create Exterior Bnd File (from the Tools pulldown menu)
4. Create Wall Bnd File (from the Tools pulldown menu)

Reading Q Files

A primary difference between the standard PLOT3D solution files and the OVERFLOW-2 solution files is the presence of one or more additional Q variables. The sixth Q variable is always the specific heat ratio, gamma, and is always present, even if it is constant. The direct reader reads the OVERFLOW-specific parameters NQ, NQC and NQT which dictate the total number of Q variables, the number of chemical components and the *(implicit) number of turbulence modeling variables present in a solution file, respectively. These additional parameters are printed to the console window. A typical output, following a successful read of an OVERFLOW-2 dataset might look like:

```
NQ = 9   NQC = 2   NQT = 1
Q variable names:
  Density (Q1)
  x-momentum (Q2)
  y-momentum (Q3)
  z-momentum (Q4)
  Stag. energy (Q5)
  gamma (Q6)
  species density 1 (Q7)
  species density 2 (Q8)
  turbulence model quantity 1 (Q9)
```

Another specific difference between the standard PLOT3D format and the OVERFLOW-2 format is an expanded section of constants which are used to derive advanced built-in functions. These constants are:

GAMINF	freestream specific heat ratio (gamma-inf)
BETA	sideslip angle
TINF	freestream dimensional temperature (Rankine)
IGAM	specifies the thermodynamic model;
HTINF	freestream stagnation enthalpy (H-inf)
HT1, HT2	the lower and upper stagnation enthalpy limits for IGAM=2
RGAS[NQC]	an array of specific gas constants;
FSMACH	freestream Mach number;
TVREF	simulation time (t)
DTVREF	timestep size (delta-t).

The above constants are available for use in the **FieldView** function calculator, and may be used to create formulas. The correct syntax for use in the function calculator is to add `OVERFLOW` as a prefix to the constant, for instance, `REFMACH` becomes `OVERFLOW_FSMACH`.

Several specific modifications have been made to correctly derive many built-in functions that are dependant upon the specific heat ratio, gamma, and/or the thermal gas constant, R. These newly derived functions are included along with the PLOT3D functions - specific OVERFLOW functions will have the string `[OVERFLOW]` appended to their names.

To avoid RESTART and formula name conflicts, the internal variable `FSMACH` will be assigned the value of `REFMACH_OVERFLOW` when reading OVERFLOW-2 datasets. This assignment also correctly handles the divide-by-zero problem which could potentially occur when `FSMACH = 0` (as in the case of hover flight for rotorcraft applications).

In converting `[PLOT3D]` to `[OVERFLOW]` functions and formulas, `GAMINF` (freestream GAMMA) is used instead of `GAMMA` (previously a constant, fixed value). The scalar variable `gamma(Q6)` is read from the OVERFLOW q-file and is used in all formulas involving GAMMA.

Special handling is also required for the case of the freestream gas constant, `R`. The appropriate choice for `R` will depend upon the thermodynamic model as follows:

- A. If `NQC < 2` and `IGAM = 0` or `1`, `R = 1` and is constant everywhere.
- B. If `NQC < 2` and `IGAM = 2`, `R` is a function of the stagnation enthalpy and `RGAS[]`.
- C. If `NQC > 1`, `IGAM` is ignored, `R` is a function of the species mass fraction (`yk`) and species gas constants (`RGAS[]`).

For case A., `RGAS[]` is not defined and should not be used. Later versions of OVERFLOW have initialized `RGAS[]` more precisely but this is not sufficient for proper user support. For cases B. and C., since a derived function for `R` is provided, it is recommended that functions use this instead. This will be consistent for all OVERFLOW-2 functions.

The complete list of derived scalars and vectors for the OVERFLOW-2 reader are:

```

Stagnation density [OVERFLOW]
Norm. stag. density [OVERFLOW]
Pressure [OVERFLOW]
Norm. pressure [OVERFLOW]
Stagnation press. [OVERFLOW]
Norm. stag. press. [OVERFLOW]
Cp [OVERFLOW]
Stagnation Cp [OVERFLOW]
Pitot pressure [OVERFLOW]
Pitot press. ratio [OVERFLOW]
Log(norm. pressure) [OVERFLOW]
Temperature [OVERFLOW]
Norm. temperature [OVERFLOW]
Stag. temperature [OVERFLOW]
Norm. stag. temp. [OVERFLOW]
Log(norm. temp.) [OVERFLOW]
Enthalpy [OVERFLOW]
Norm. enthalpy [OVERFLOW]
Stag. enthalpy [OVERFLOW]
Norm.stag.enthalpy [OVERFLOW]

```

```

Norm. int. energy [OVERFLOW]
Norm. stag. energy [OVERFLOW]
Mach number [OVERFLOW]
Speed of sound [OVERFLOW]
Entropy [OVERFLOW]
Entropy measure s1 [OVERFLOW]
Shock function [OVERFLOW]
Filter. shock func. [OVERFLOW]
Press.gradient mag. [OVERFLOW]
Shock Finder [OVERFLOW]
Shock Filter [OVERFLOW]
Shock Finder [OVERFLOW] [Isentropic Transient]
Shock Finder [OVERFLOW] [Transient]
Shock Filter [OVERFLOW] [Isentropic Transient]
Shock Filter [OVERFLOW] [Transient]
OVERFLOW_REFMACH

```

and

```

Press.grad. Vectors [OVERFLOW]

```

Transient Data File Naming Convention

FieldView will recognize a transient sequence using OVERFLOW-2 file naming conventions (as described in [Transient Data](#)), with .fvbnd files specified as either `prefix.#####.fvbnd` or `prefix.fvbnd` (this latter option provides for the use of a single .fvbnd file over an entire transient sequence).

Complete support has been provided for RESTARTS. Also, **FVX** can be used to read an OVERFLOW dataset - please refer to [Chapter 4](#) for an example.

DataGuide™ Support

Complete support has also been provided for **DataGuide™**. This feature offers the benefit of significantly reduced read-in times, and lowers the memory requirements needed to work with data. Note however that if the `-p1` switch is used when generating **DataGuide™** files, we will not process any of the additional Q variables or any of the newly derived functions.

Support for brk (brick) files

We have provided limited support for brick files. If the `brkset.restart` file exists in the current working directory, this file is used by **FieldView** to define the Cartesian topology of the "off-body" grids (or bricks). This file contains information which defines the topology of both "near-body" (curvilinear) and brick grids. The full PLOT3D x-files contain the same information in complete nodal format. During the course of a "normal" OVERFLOW-2 simulation, the brick grids are static. Therefore, **FieldView** is able to only read the xyz coordinates of the near-body grids and use the Cartesian topology to construct the bricks. This applies for both steady and transient datasets. Note that although the brick topologies are static, their associated IBLANK arrays can change. Therefore, the full x-file must still be read thru to update the IBLANK array properly. If the `brkset.restart` file is not present, the full (unsteady) x-files must be used instead.

For the case of transient, the brk files follow a naming convention for which the time step number is specified as a numeric string suffix, i.e. `brkset.###`. **FieldView** handles this as follows. If the grid file name ends in a number, we will look for a matching brkset file, and not use the `brkset.restart` file.

Transient OVERFLOW-2, Adaptive (Changing) Grids

Data in which grids or node count change during a transient sweep is permitted only for OVERFLOW-2 (and PLOT3D) files. Other types of structured data input (including plugin readers) are not supported in this way. Note that earlier versions of FieldView required that you have a FieldView Boundary (.fvbnd) file for each time step for cases like this.

In this scenario, FieldView will prohibit use of the Computational Surface Panel. To work with I, J, and K visualization objects, you can create FieldView Boundary files (.fvbnd) for each time step to represent I, J, and K ranges as Boundary Types. For instance, consider a case where your grid file has the name, `grid.001`. Then, if your boundary surface in `grid.001` is composed of:

grid 1, K = 1, I = 20 thru 30, and J = 10 thru Max
grid 2, K = 1, I = 10 thru 15, and J = 15 thru Max

The Boundary file `grid.001.fvbnd` would look like:

```
FVBND 1 4
My Boundary Name
BOUNDARIES
1 1 20 30 10 $1 1 F 0
1 2 10 15 15 $1 1 F 0
```

Further detail concerning the .fvbnd format can be found in [Appendix H](#)

This mode of operation is fully supported with RESTARTS. **FVX** also supports this mode of operation implicitly. There are no special **FVX** commands needed to perform a transient sweep on a dataset with a changing grid count.

The creation of **DataGuide™** files are fully supported.

Grid subsetting is not supported for transient changing grid counts.

FieldView Parallel Support

The OVERFLOW-2 reader is completely compatible with **FieldView** Parallel. Also, full support has been provided for Partitioned File Parallel. Note that PFPR has *not* been enabled for brick grids.

Additional Comments and Limitations

This reader is limited to read only the OVERFLOW-2 format. Support for the older OVERFLOW format is not covered with this reader.

Correct surface grid integration for overset grid handling has not been included with this reader. This shortcoming will be addressed in a later **FieldView** release.

Command line values for the specific heat ratio, `-gamma`, and the ideal gas constant, `-gasconstant`, will be ignored when an OVERFLOW-2 dataset is read into **FieldView**.

Dataset sampling does not support the additional Q variables present in OVERFLOW-2 files.

The number of species that this reader can support is currently limited to 20.

PHOENICS - BFC Data

(www.cham.co.uk)

This data reader will read in files from the PHOENICS program from CHAM (Version 1.6.5 to Version 2.2.1). Both a grid file and a restart file (PHI file) are required. Please note that these files may not be in direct access format. 2D data is fully supported. Note: Non-Cartesian files are not supported by this reader.

This reader is currently accessed from the fly-out menu in the Data Input pulldown menu, when you select the More Readers... option.

Translator

A Windows NT utility program, `phoplt3d.exe` is available that converts PHOENICS PHI(DA), XYZ(DA) and, optionally, EARDAT files, into PLOT3D file formats which can be read into **FieldView**. The utility program `phoplt3d.exe` is a binary executable that will only run on a Microsoft Windows NT 4 operating system (SP 3 or higher). There is no UNIX version of this utility. This executable and accompanying documentation (`FV_PHOENICSwininstructions.pdf`) are available in the `/PHOENICS` directory of the `/translators` directory on the **FieldView** Installation Disc. These files will need to be manually copied from the **FieldView** Installation Disc as the `/translators` directory is *not* automatically installed.

PHOENICS - non-BFC Data

(www.cham.co.uk)

This data reader will read in files from the PHOENICS program from CHAM (Version 1.6.5 to Version 2.2.1). Only the restart file (PHI file) is required. The data reader will use this file as both the grid and results file. Please note that this file may not be in direct access format. 2D data is fully supported. See above for translator information.

This reader is currently accessed from the fly-out menu in the Data Input pulldown menu, when you select the More Readers... option.

PLOT3D

The PLOT3D format is a standard file format for structured grids developed by NASA. A complete description of this format and various aspects of using it in **FieldView** can be found in [Appendix B](#) and [Appendix C](#) of this **Reference Manual**.



Note: When cylindrical coordinates are specified using an FVREG file (Region definition), then XYZ information is presented to you in **FieldView** as RTX/Z (Radius, Theta, X/Z) data. That is, Coordinate surface sliders, the 2D plotting system, Point Probing, Streamline seeding, Exporting (except for Streamline export, which still uses XYZ coordinates), etc. will be in RTX/Z coordinates instead of the usual XYZ coordinates. See [Chapter 3](#) of this **Reference Manual** for more information. Note that the Grid file itself is still required to be in XYZ coordinates.

Transient PLOT3D Data

FieldView supports transient PLOT3D datasets. Transient PLOT3D data requires one file per time step for *at least* the results (Q or Function files). Multiple Grid (XYZ) files are only required if the grid is moving or otherwise changing. This is different from other solvers (FLOW-3D®, for example), where all data resides in a single file. One advantage of this technique is that it can be far quicker to access a specific time step during visualization.

A dataset will automatically be recognized as transient if each file has a time step number embedded in its name using a convention described in [Transient Data](#). A transient dataset can consist of a *single* Grid (XYZ) file and *multiple* Q or Function files (or both, at one each per time step), or, for moving grid/ changing geometry configurations, *multiple* Grid (XYZ) files and *multiple* Q or Function files (one per time step). The actual extension (*.bin, *.dat, *.xyz) used is not important and is not required. Multiple extensions are permitted.

Example: (moving/changing grid)

Grid file: grid0010.bin, grid0020.bin, grid0030.bin, ..., grid4030.bin
Q file: q0010.bin, q0020.bin, q0030.bin, ..., q4030.bin

For this example, when any one of the grid files is chosen during read-in, **FieldView** will find similarly named files in the same directory and allow the option of treating this case as a *transient* case, listing the number of files found similarly named. If you do *not* decide to treat it as a transient set, then the dataset will be treated as non-transient. If you *do* decide to treat it as transient, **FieldView** will store the names of all of the grid files, but only read-in the one chosen. It will, however, allow access to the other time steps through use of the Transient Data Controls panel (see [Chapter 14](#) of **Working with Fieldview** for more details). If a *different* Q file (different time step) is chosen during the next phase of the data read-in, then the appropriate Grid file will be accessed as well.



Note: If you have multiple grid files, you must read in the same result file (time step) as the grid file chosen. (If you choose a different results time step, FieldView will issue a message and force loading of the correct time step.) Once data is read in, using the Transient Data Controls panel to select any given time step will always result in proper matching files.

Merge Series

For Merge Series, a series of 2D files (or 3D files with one of the dimensions equal to 1) may be read in as time steps of a solution. The files will be appended together in the K dimension (or in the dimension that is equal to 1, for 3D solutions).

Example: (non-moving/non-changing grid)

Grid file: case4.xyz

Q file: case4_1020.q, case4_1021.q, case4_1022.q, ..., case4_1335.q

For this example, when the Grid file is read in, there will be no other similarly named *.xyz files. Hence, **FieldView** will not present any transient pop-ups. However, when a Q file is chosen, **FieldView** will recognize the series and present the options. If you decide to treat the series as transient, the time step picked will be the data immediately available in **FieldView**, but all time steps accessible through the Transient Data Controls panel (see [Chapter 14](#) of **Working with Fieldview** for more details).

See the [Basic Aerospace Tutorial](#), the [Basic Combustion Tutorial](#) and the [Basic Turbomachinery Tutorial](#) in the **User's Guide** for more information on working with PLOT3D data.

Transient PLOT3D, Adaptive (Changing) Grids

Data in which grids or node count change during a transient sweep is permitted only for PLOT3D (and OVERFLOW-2) files. Other types of structured data input (including plugin readers) are not supported in this way. Note that earlier versions of FieldView required that you have a FieldView Boundary (.fvbnd) file for each time step for cases like this.

In this scenario, FieldView will prohibit use of the Computational Surface Panel. To work with I, J, and K visualization objects, you can create FieldView Boundary files (.fvbnd) for each time step to represent I, J, and K ranges as Boundary Types. For instance, consider a case where your grid file has the name, grid.001. Then, if your boundary surface in grid.001 is composed of:

grid 1, K = 1, I = 20 thru 30, and J = 10 thru Max
grid 2, K = 1, I = 10 thru 15, and J = 15 thru Max

The Boundary file grid.001.fvbnd would look like:

```
FVBND 1 4
My Boundary Name
BOUNDARIES
1 1 20 30 10 $1 1 F 0
1 2 10 15 15 $1 1 F 0
```

Further detail concerning the .fvbnd format can be found in [Appendix H](#).

This mode of operation is fully supported with RESTARTS. **FVX** also supports this mode of operation implicitly. There are no special **FVX** commands needed to perform a transient sweep on a dataset with a changing grid count.

The creation of **DataGuide™** files are fully supported.

Grid subsetting is not supported for transient changing grid counts.

Double Precision PLOT3D Data

FieldView supports the Double Precision (DP) Unformatted format (in addition to Formatted, Unformatted and Binary PLOT3D formats). DP Unformatted files store floating point results data in double (64 bit) precision. This is true for XYZ files, Q files, Function Files, and Surface Based results files. All integer data for DP Unformatted files is stored in single (32 bit) precision.

Use DP Unformatted toggle button in the FILE FORMAT section of the PLOT3D Data Input panel to read DP Unformatted files.

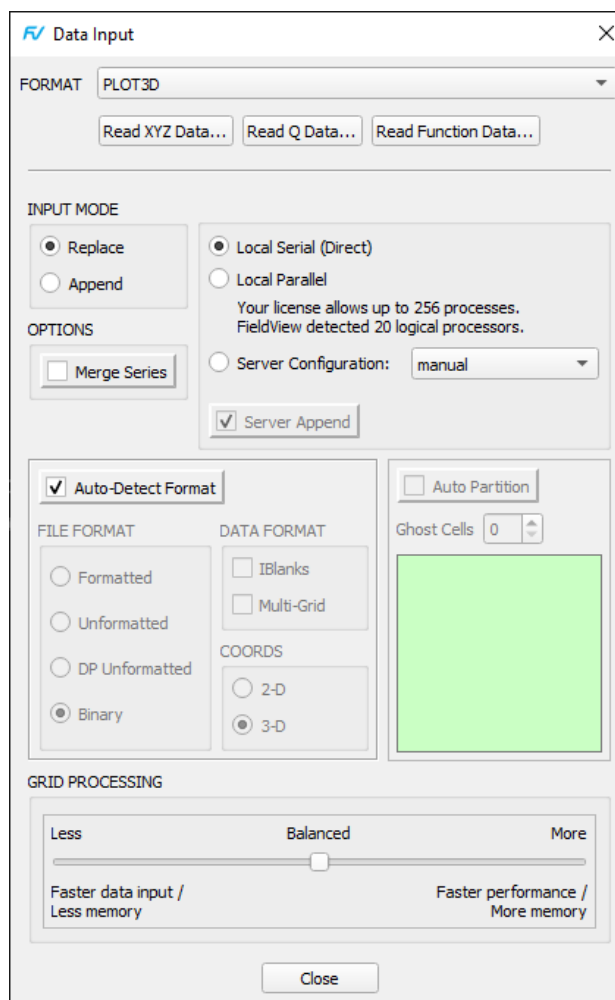


Figure 31 PLOT3D Data Input

Face Data for PLOT3D Data

FieldView supports face-based results on boundary surfaces of PLOT3D data. In order to provide face results for a 3D dataset, *three* additional files will need to be created. The first is a special form of the Structured Boundary file (*.fvbnd) which communicates the boundary surface definitions, the surface normal directions, and whether there are face results for each of the boundary surfaces. The format is described in [Appendix H](#) of this **Reference Manual**. The second is a 2D Function File, which contains the face results for those boundary surfaces that have them and the third is a Function Name file which communicates the names of the face result variables to **FieldView**. See the Face Results sections of [Appendix H](#) of this **Reference Manual**.

SC/Tetra

We recommend that this direct reader be used to read SC/Tetra datasets from Software Cradle Co., Ltd. Exporting results to the native **FieldView** unstructured file format is not required.

To read a SC/Tetra dataset, start by selecting the Read Grid & Results Data... button on the Data Input panel (see **Figure 32**). You will be presented with a file browser which will let you navigate to the location of the dataset that you wish to read.

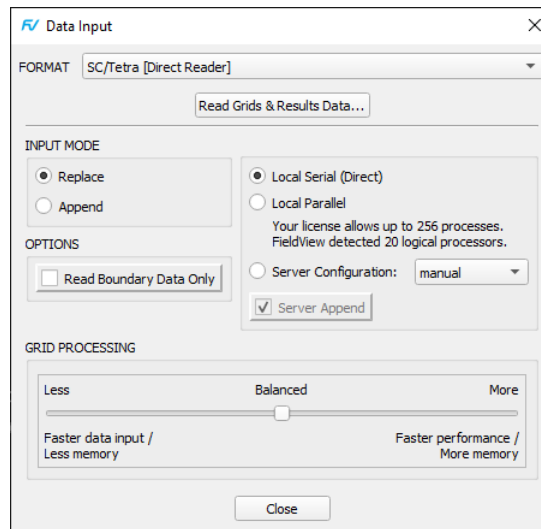


Figure 32 SC/Tetra [Direct Reader] Data Input Panel

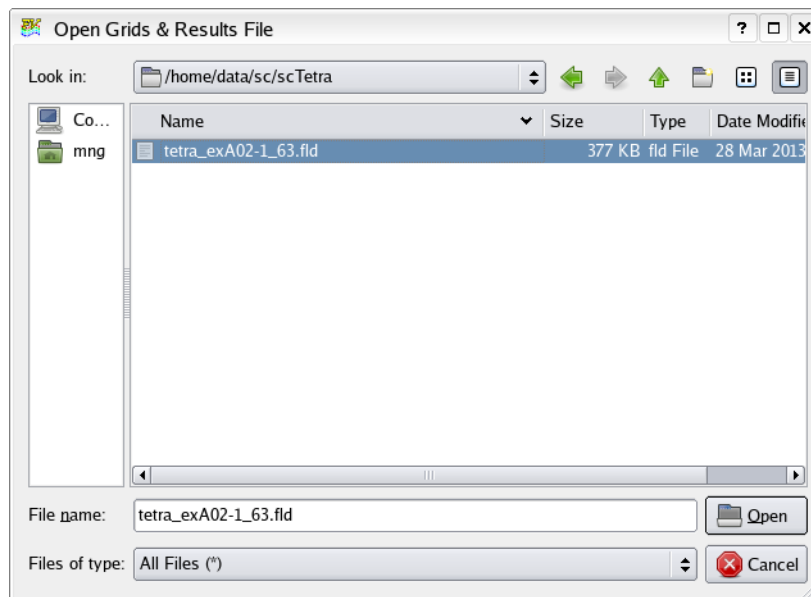


Figure 33 SC/Tetra [Direct Reader] File Browser

Automation for the SC/Tetra direct reader is completely supported with Restarts, **FVX** and Python scripting.

Known Limitations

- Parallel operation is not currently supported for this direct reader.
- Transformed grid transient cases will read correctly, however the grid will appear to be stationary instead of moving.

scFLOW

This direct reader may be used to read scFLOW datasets from Software Cradle Co., Ltd. This reader has native support for arbitrary polyhedra, moving grids, overset grids, nodal and face based results.

Automation for the scFLOW direct reader is completely supported with Restarts, **FVX** and Python scripting.

Known Limitations

- Parallel operation is not currently supported for this direct reader.

SCRYU

(www.cradle.co.jp)

This program (available in Japan, and now more recently in North America) has an export to a file format which is directly read-able by the SCRYU Reader.

scSTREAM

We recommend that this direct reader be used to read sc STREAM datasets from Software Cradle Co., Ltd. Exporting results to the native **FieldView** unstructured file format is not required.

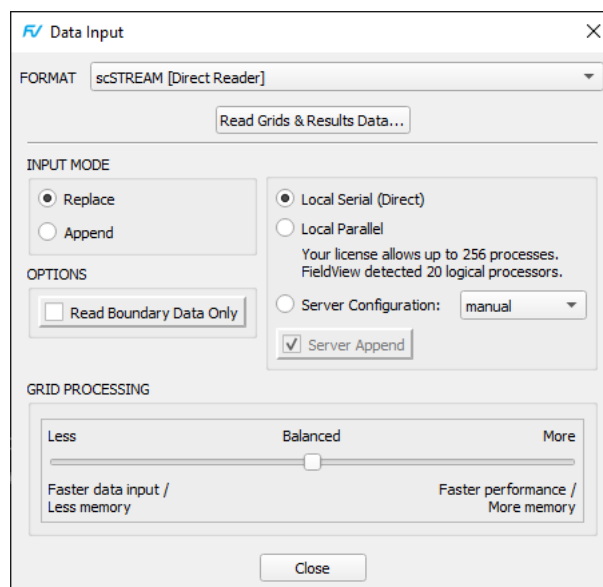


Figure 34 scSTREAM [Direct Reader] Data Input Panel

To read a sc STREAM dataset, start by selecting the Read Grid & Results Data... button on the Data Input panel (see **Figure 34**). You will be presented with a file browser which will let you navigate to the location of the dataset that you wish to read.

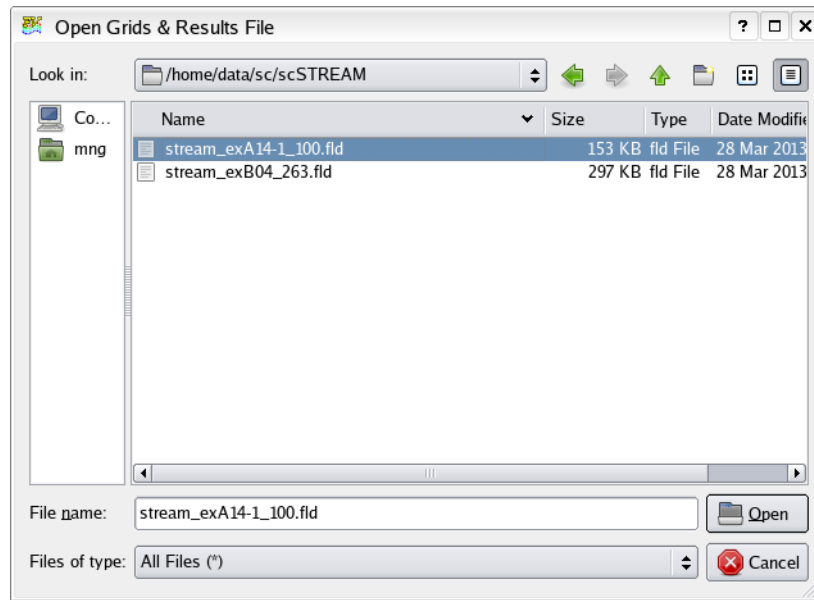


Figure 35 scSTREAM [Direct Reader] File Browser

Automation for the scSTREAM direct reader is completely supported with Restarts, **FVX** and Python scripting.

Known Limitations

- Parallel operation is not currently supported for this direct reader.

Surface Sampled Data

This reader is used to read FV-UNS files written by surface sampling. For information, see [Surface to Surface Sampling for Dataset Comparison](#) in the **FieldView Reference Manual**.

STL

This reader lets you read in STL (Stereolithographic) CAD data, allowing you to show the original CAD representation of your model. Independent scaling and translation of any dataset allows you to show your STL CAD data alongside your CFD simulation results in the same postprocessing session.

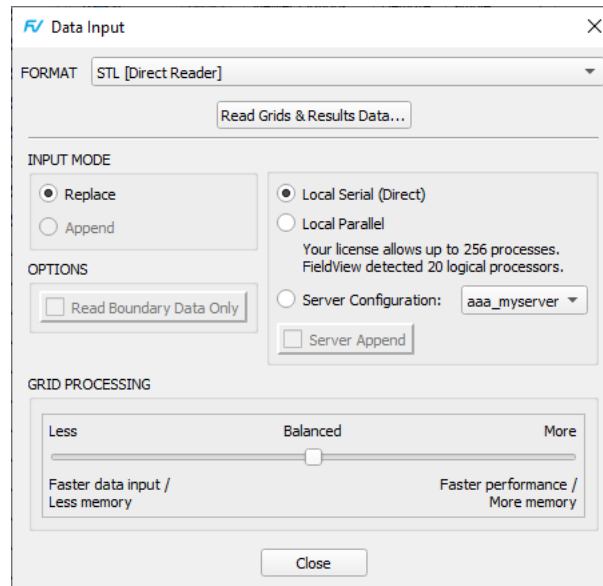


Figure 36 STL [Direct Reader] Data Input Panel

To read an STL file, select the STL [Direct Reader] entry from the Data Input pulldown menu. On the STL [Direct Reader] panel, click Read Grids & Results Data... When you do this, you will see a file browser, as illustrated below.

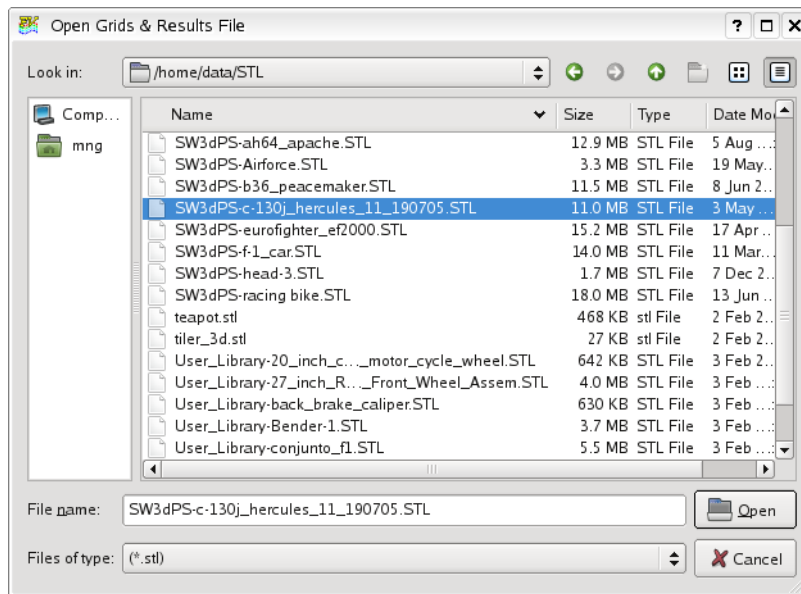


Figure 37 STL [Direct Reader] File Browser

NOTE: The STL file browser is case-insensitive. However the STL [Direct Reader] in FieldView expects that extensions .stl and .STL are two different formats ("stl" and "stlbin", respectively), and if the incorrect extension is used for the given file type, the file will fail to read, and FieldView will issue *Input File Error* message.

UH3D

This solver is designed for vehicle front end cooling analyses. The code was originally developed by Core Technologies in the Ford Motor Company, with commercialization and development supported by MindWare Engineering. Reader modules are available for grid and results volume data and for the UH3D geometry data. To enable the reader, launch **FieldView** with the "-uh3d" command line option. Contact **Tecplot Inc.** directly for additional details on this reader.

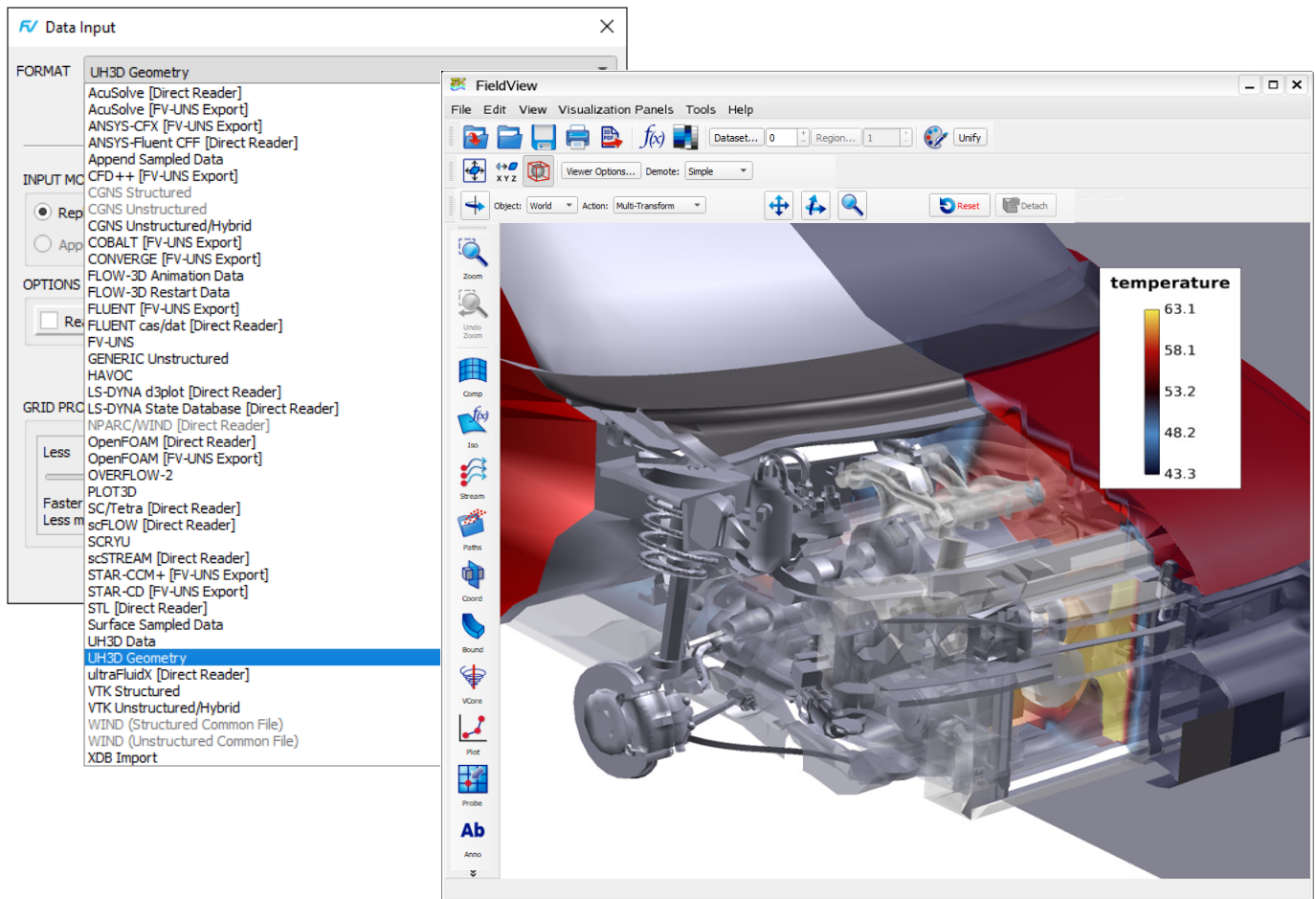


Figure 38 UH3D Reader port for WINDOWS

ultraFluidX

FieldView supports reading results from the Lattice Boltzmann solver ultraFluidX. This reader has been optimized for transient datasets where grids are not moving, making rebuilding of the grid data unnecessary when changing timesteps. Also, starting with FieldView 20, this reader supports cases with moving grids, as well as a changing number of elements and nodes over time.

The Data Input menu entry "ultraFluidX [Direct Reader]..." (see [Figure 11](#)) will bring up the panel below.

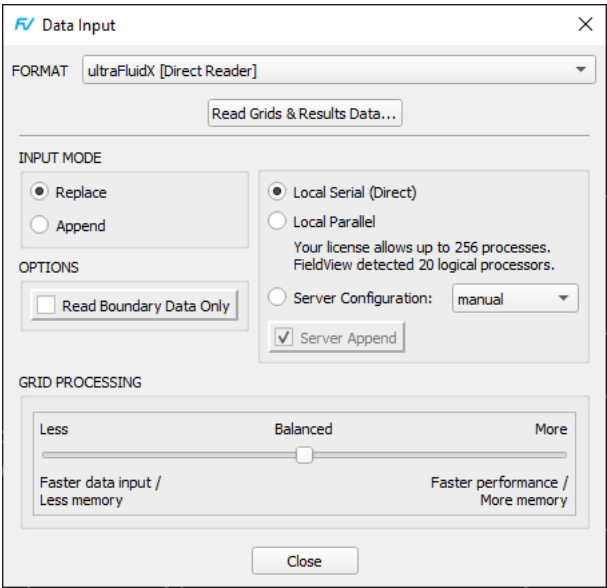


Figure 39 ultraFluidX [Direct Reader] panel

To read ultraFluidX results, use a parallel data input mode (such as "Local Parallel") to load the `uFX_output.layout` file generated by ultraFluidX, as seen in [Figure 40](#).

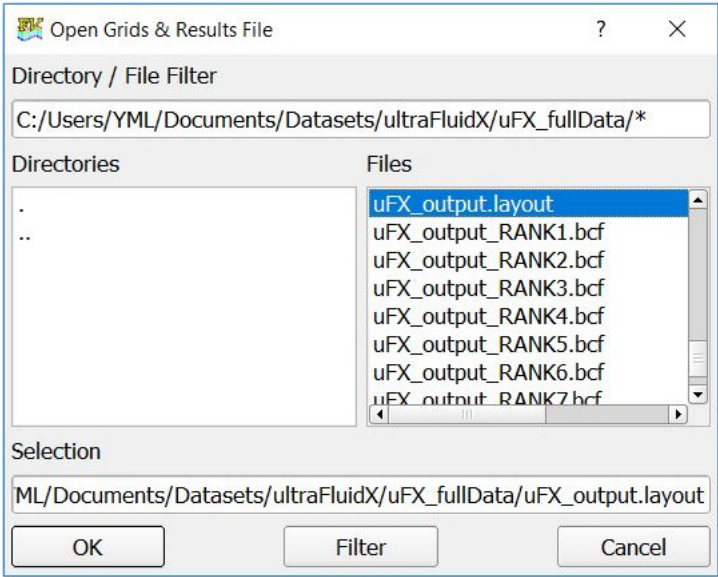


Figure 40 FieldView results read operation

Next, the Function Subset Selection panel will be displayed. This allows reading only a subset of the variables stored by ultraFluidX.

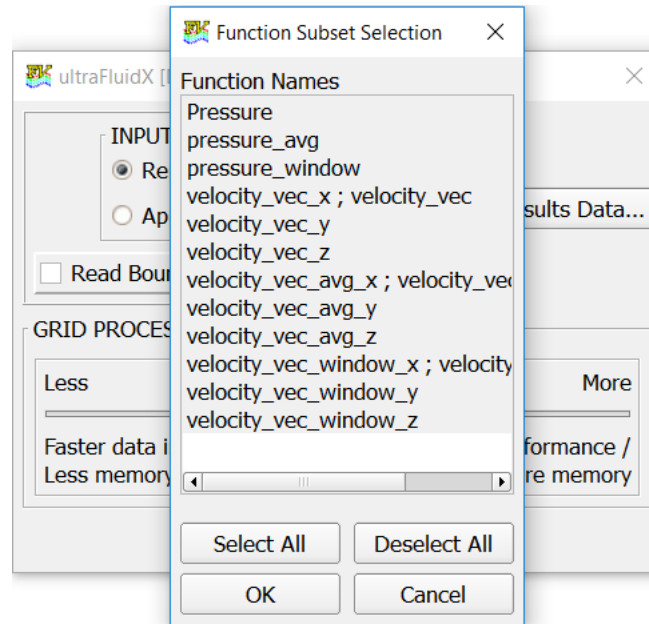


Figure 41 Function Subset Selection panel

Finally, the Time Step Selection panel allows selecting a particular time step to be the first one to be read. All time steps will later be accessible through the Tools > Transient Data... menu.

Alternatively, the dataset can be read as a steady state case by selecting "Read as Steady State". In that case, only the selected time step will be available.

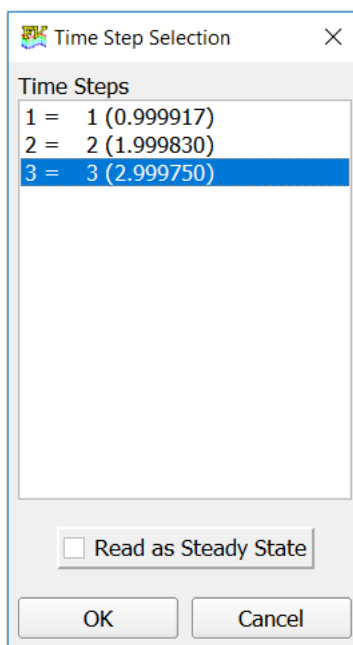


Figure 42 Time Step Selection panel

How to read distributed results from ultraFluidX? (Recommended approach)

ultraFluidX outputs volume results in a distributed manner per rank, so one *.bcf file per rank (uFX_output_RANK#.bcf, with # being the number of the rank) will be created in the subfolder uFX_fullData, together with the mesh and result files for all the saved time steps. To allow reading all these *.bcf files in a single operation, ultraFluidX also writes a so-called **FieldView** Layout file (uFX_output.layout) that groups the *.bcf files. In order to read this layout file, the “Local Licensed Parallel” option (from the File > Data Input menu) or a parallel Server Configuration file must be selected. **FieldView** will fail to read the layout file if the default File > Data Input > Direct mode is selected. The benefit of reading distributed results in parallel is that it will be faster than reading the same case in a merged form, and many **FieldView** operations will benefit from parallelization in the same way. Please note that there is no need to match the number of **FieldView** worker processes with the number of ranks used by ultraFluidX. For more on running **FieldView** in parallel, please see [FieldView Parallelization with Multithreading and MPI](#) in [Chapter 1](#) of the **User’s Guide**.

ultraFluidX also outputs distributed surface results in the subfolder uFX_surfaceData, with the respective *.bcf and layout files. These results can either be read separately or appended to the volume results in a single **FieldView** session by selecting the “Append” option seen on [Figure 39](#). The read procedure described above for volume results applies to surface results.

How to read merged results from ultraFluidX?

Optionally, ultraFluidX can write a copy of the volume and surface results with all the ranks merged into single files. When that is the case, one *.bcf file will be written for the volume results, and one for

the surface results. These results can only be read in serial mode, with File > Data Input > Local Serial (Direct), or a with serial Server Configuration being selected. **FieldView** cannot read them in parallel.

ultraFluidX also has a number of optional exports that can be read in **FieldView** with the same procedure as described above for merged results, by selecting the corresponding *.bcf file.

- Section cuts will be exported to a separate subfolder named `uFX_sectionCuts`
- Partial surfaces will be exported to `uFX_surfaceData`, in the subfolder `uFX_partialSurface_name`
- Partial volumes will be exported to `uFX_volumeData`, in the subfolder `uFX_partialVolume_name`

VTK

As VTK is supported by many CFD solvers, this reader will provide **FieldView** users with an additional interface option. VTK also has the advantage of including formats for 2D surfaces and solutions, as well as 3D meshes, making it an alternative to XDB files for extract based workflows. With this reader, users who have configured their solvers for writing VTK extracts, sometimes using in situ, will have the option to analyze their results in **FieldView**.

The **FieldView** Data Input menu ([Figure 11](#)) includes two entries for reading VTK files:

VTK Structured...
VTK Unstructured/Hybrid...

The Structured reader maintains the structured nature of the dataset, giving access to structured specific capabilities in **FieldView**, such as Computational Surfaces, structured surface plots, etc. The Unstructured/Hybrid reader is capable of reading unstructured cell data, face or polygon data, and structured data, which is converted to unstructured cells. It can thus read a wider range of VTK files, but doesn't preserve the nature of structured grids.

Both the structured and the unstructured/hybrid readers are capable of reading `VTK_STRUCTURED_GRID` and `VTK_RECTILINEAR_GRID` geometries, but only the unstructured/hybrid reader is capable of reading `VTK_UNSTRUCTURED_GRID` and `VTK_POLY_DATA` geometries.

VTK is made of a wide variety of formats, most of which are supported by **FieldView**. More specifically, **FieldView** can read:

- Structured files ending in "s", "r" or "i" (`.vts`, `.vtr`, `.vti`)
- Unstructured files ending in "u" or "p" (`.vtu`, `.vtp`)
- Structured or unstructured legacy files with the extension `.vtk`
- Structured, unstructured or hybrid (both) multi-block legacy files with the extension `.vtm`
- Parallel partition files with extensions that start with "p" (e.g., `.pvtu`)

A `.vtm` file is a metadata file, containing a set of files of types already mentioned. The Structured reader will read only the structured grids in a `.vtm` file. The Unstructured/Hybrid reader will read all grids, converting structured to unstructured.

Like a `.vtm` file, a VTK parallel partition file is a metadata file, with a separate grid in each file. Metadata files, including `.vtm` and parallel partition files, can be read as multi-grid parallel, which will reduce read time but is not a requirement.

Multi-file transient is enabled. For information on multi-file transient naming conventions, please refer to the [Transient Data](#) section.

Both the structured and the unstructured/hybrid readers support IBlanking information from the `vtk-GhostType` scalar if it exists.

An attempt to read data not supported by one of the VTK readers will produce a console message such as:

```
VTK reader: data type vtkUnstructuredGrid is unstructured
and is not supported by the structured reader.
```

There is nothing in the format that presents itself as boundary types, so all 2-D elements are collected under a single boundary "default".

VTK may specify results at nodes or at cell centers. Any results at cell centers are interpolated to nodes for **FieldView**, except in the case of face-based boundary results, where the results will be present in both face-centered and nodal variations. The face-centered results for boundary surfaces will have a variable name suffix "[BNDRY]".

Limitations

Point set geometries are not supported.

Vertex and line element types are not supported. `VTK_PENTAGONAL_PRISM` and `VTK_HEXAGONAL_PRISM` element types are not supported. When any of these are encountered, a console message will be issued to that effect and the unsupported elements are otherwise skipped.

WIND US

The WIND US plugin reader has been provided as a courtesy of the NPARC Alliance. A version of this plugin has been included in **FieldView** and is supported for the LINUX64 platform only. To learn more about the capabilities of this reader, documentation is available at <http://www.grc.nasa.gov/WWW/winddocs/>. To obtain the latest version of this plugin reader, please contact the NPARC Alliance directly at (nparc-support@arnold.af.mil <<mailto:nparc-support@arnold.af.mil>>).

The WIND US plugin contains two readers, one for Structured Common Files and the other for Unstructured Common Files. Full support for RESTARTS and **FVX** is provided.

To read WIND .cgd (grid) and .cfl files, start by selecting either the WIND (Structured Common File) or WIND (Unstructured Common File) entry on the Data Input pulldown menu. On the data input panel, click Read Grid Data... When you do this, you will see a file browser which will let you navigate to the location of the .cgd file that you wish to read. After you select the file you want to read, click the Open button. When this is done, a second file browser is launched to allow you to read the .cfl file. **FieldView** automatically attempts to locate the matching .cfl file for the .cgd file which has already been read. If a matching file is found, it will show up in the Filename section of the file browser. You can read this data file by clicking the Open button. It is also possible to read a different .cfl file by browsing to the results file of interest. Although file filters have been implemented to make locating .cgd and .cfl files easier, they can be easily over-ridden to select datasets saved with different suffixes.

A common file can handle both structured and unstructured grids in the same file (or simulation). If a Common File of this type is read using the Structured reader, the unstructured zones are skipped. If it is read using the Unstructured reader, the structured grid cells are converted to unstructured cells and both grid and solution are read. When the Unstructured reader is used to read structured data, computational surfaces are not available. Also, structured grid boundaries will not be available. Consequently, when reading a hybrid dataset, it may be beneficial to read it twice; once using the Structured reader to display the boundaries, and again using the Unstructured reader to calculate streamlines and/or display iso-surfaces.

It is a requirement that you have write permissions to the directory that contains the dataset(s) you are attempting to read. This is because **FieldView** will automatically create a structured boundary file (.fvbnd). If your dataset is called my.cgd, then the corresponding name of the structured boundary file will be my.cgd.fvbnd. If you do not have write permissions, **FieldView** will issue the following message:

```
Failed to write structured boundary file
```

and the .fvbnd file will not be generated. In addition, if an older .fvbnd file (with the same name) already exists and the user does not have file permission to overwrite it, the boundary information may be incorrect and will not be read in if the grid information found in this older .fvbnd file is different than that for the current grid.

If you want to disable the writing of the structured boundary file, the environment variable FV_NO_BOUNDRY_FILE needs to be set (to any value).

Transient results, stored as one file per time step, are automatically recognized, based on the standard **FieldView** naming conventions.

Multi-grid datasets can be read on the LINUX64 platforms using Local Parallel, or using your own server configuration file. Since this is a fully integrated plugin, it is possible to read WIND data from any **FieldView** Client using a server configuration file for a LINUX64 system.

An update to the formula restart file, wind.frm, is installed in the /fvx_and_restarts directory of your **FieldView** installation. We recommended that this formula restart gets read after reading any WIND data since it provides many additional scalar and vector functions to examine your data.

Several environment variables are available to provide additional control over how WIND data is read and displayed in **FieldView**.

Variable	Definition
FV_UNSTR_CGD_SURFONLY	Set this variable to read only unstructured boundaries for faster loading. When this is set, it will not be possible to create surfaces or rakes within the dataset volume.
FV_UNSTR_CGD_BC	Define boundary type names based on their respective boundary conditions rather than the surface IDs. Boundary Surface restarts will be affected by this setting.
FV_UNSTR_COUPLED SURFS	Omit coupled surfaces from returned boundary surfaces. Boundary Surface restarts will be affected by this setting.
FV_UNSTR_CGD_NOPRISMLAYER	By default, an internal boundary surface is created which contains all triangular faces having a prism on one side and a non-prism on the other. Set this to omit defining the edge of your prism layer as a boundary surface.
FV_UNSTR_CGD_STRUCTCONVERT	Convert structured zones to unstructured format (Note applies to volume information only, no boundaries are defined).
FV_UNSTR_CGD_METRIC	The specification of consistent units needs to be done when setting up the solver. Set this variable to provide grid and solution variables in MKS rather than FSS units.
FV_UNSTR_CGD_UNITS	Provide grid units using native units created by the grid generator and based on the underlying problem geometry. Flow variables are displayed using FSS units. This may introduce an inconsistency in displaying results. The computation of streaklines for transient cases may be affected by this setting.
FV_UNSTR_NOGROUP	It is possible to create a named collection of structured or unstructured surfaces. Surfaces can be still be manipulated using their Boundary Type names. Set this to ignore surface group definitions in boundary surfaces. Boundary Surface restarts will be affected by this setting.

XDB Import

FieldView is able to import as well as write XDB files. There are essentially four types of XDB files:

1. Steady state, which is a snapshot of all surfaces and rakes

2. Surface sweep, containing a sweep of a coordinate, computational or iso-surface (and possibly other surfaces)
3. Transient sweep, saving each time step to an individual file to create a series of files.
4. Transient sweep, which is a sweep saving all timesteps to a single file. This requires that the environment variable `FV_USE_FV13_XDB_FORMAT` be set.

An XDB file is imported from the Data Input menu. XDB imports can be read directly or remotely using **FieldView** Client Server. XDB files can not be imported using parallel servers unless read via PFPR **Partitioned File Parallel Reader (PFPR)** layout files. However XDB files tend to be small and can be efficiently read without the need for parallel. This menu is also where you can select *Local Parallel* or your own custom server configuration as explained on [page 19](#) in the Installation Guide.

The XDB reader has been extended in **FieldView 19** to support the new XDB format introduced with XDBLib 2.0 (only supported on Linux64). XDBLib 2.0 is **FieldView's** solution for in situ post-processing. It's a library that allows solvers with native in situ post-processing and the ones instrumented for in situ thanks to libsim, which writes directly to XDB extracts, without having to write full 3D datasets. This is particularly interesting for large and transient simulations, for which full 3D mesh and results are too large and take too long to process. If you're interested into getting access to the XDBLib 2.0 library or in getting help in instrumenting your solver for in situ post-processing, please contact **Tecplot Inc.**

Once an XDB file has been read, it can be managed just like any other dataset in **FieldView**. XDB datasets can be manipulated in the following ways:

1. Datasets can be scaled, rotated, translated, mirrored or rotationally duplicated
2. Multiple copies of a surface can be created; each surface can be independently transformed, scaled and/or rotated
3. 2D plots can be created on surfaces; surfaces can be probed
4. XDB datasets can be appended and Dataset Comparison can be used
5. Linked Surface sweeps and Merged Transient sweeps can be performed
6. Geometric functions (such as X, Y and Z) no longer need to be saved when creating XDB files. They are automatically available when the dataset is read, and can be used to threshold surfaces.
7. Vector components no longer need to be saved when creating XDB files. If a vector quantity is saved, all components of the vector will be available to be used for scalar coloring and thresholding.
8. New XDB extracts can be created from XDB datasets (useful for limiting time ranges for example)

Although information is available within the XDB files to describe what they were originally, all surfaces are stored as boundary types. A boundary type naming convention has been established to create some correspondence between the original surface type and its name within the XDB file. In general, the first part of the boundary type name corresponds to the type of surface it was originally derived from. The remaining part of the name attempts to provide additional context for that surface up to an 80 character limit.

For further information on XDB Imports, see [XDB Workflows for CFD](#) in **Working with FieldView**.

Partitioned File Parallel Reader (PFPR)

The Partitioned File Parallel Reader can read all supported **FieldView** formats except PW Common File. As a result, you can create a **FieldView** layout file to read multiple XDB files, for example.

Please note that the **DataGuide™** feature is also supported. **DataGuide™** files can be created for each individual FV-UNS or PLOT3D file, specified in the layout file.

In addition to interactive reading, **FieldView** Restarts and **FVX** are fully supported.

Important points and limitations

You must be running a **FieldView** parallel shared memory (shmem) or cluster (p4) server.

The layout file must be visible to the controller **FieldView** server process on the target system.

The name of the layout file is arbitrary, except for the usual rules for transient filenames (see below). It is recommended that a file extension, *.layout, be used to help you more easily identify layout files, but this is not a requirement at this time.

Parallel data readers, including PFPR, do not support the following **FieldView** features:

- Dataset Sampling
- Create Wall Boundaries
- Create Exterior Boundaries

Description of Layout File Format

Layout files are text files with the following general format:

```
FIELDVIEW LAYOUT 1
Base filename of first partition file
Hostname
Directory of first partition file on this machine
Base filename of second partition file
Hostname
Directory of second partition file on this machine
```

A drawback of this format is that it is not flexible enough in working with job schedulers. To overcome this limitation, wildcards are allowed to be used for the Hostname specification. A layout file may look like this:

```
FIELDVIEW LAYOUT 1
Base filename of first partition file
*
Directory of first partition file on this machine
Base filename of second partition file
*
Directory of second partition file on this machine
```

It is possible to provide a mix of wildcards and hostnames if desired.

The "Directory" entry for a partition can be relative to the directory containing the layout file. For example, if the Directory entry is ".", the partition is assumed to be in the same directory as the layout file. If the Directory entry is "GridFiles", the partition is assumed to be in a directory called "GridFiles", which is a sub-directory of the layout file's directory.

Relative pathnames in layout files make them more portable. You can move the layout file and the relative partitions to a different parent directory, without having to edit the layout file.

No comment lines are allowed.

No leading spaces (blanks) are allowed.

If the unstructured data has separate grid files and results files, then there must be a layout file for the grid partitions and a separate layout file for the results partitions.

For PLOT3D data, there must be a layout file for the grid files, the Q files, and the function files if they are present.

The hostnames in the layout file must all belong to the current MPI group, but these names can be wildcarded using "*". The same hostname can occur multiple times in the layout file. Having a greater

or lesser number of partitions than available server processes is allowed. Having more partitions than server processes is referred to as "overload".

Grid numbering is as follows after reading partitioned data: All of the grids from the first partition file come before all of the grids from the second partition file, and so on.

For transient partitioned data, there must be separate layout files for each time step. The layout file names must follow the standard **FieldView** naming conventions for time step files:

basenameNNN.extension

- or -

basenameNNN

If there are separate grid and results layout files, the results layout files can be transient while the grid layout file is not, as with non-partitioned data.

Simple Layout File example

For a dataset made up of two partitions, where each partition is located on a different file system, the layout file is:

```
FIELDVIEW LAYOUT 1
cyl.A.UNS
gecko
/nfs/tmp/mydata/cyl_vib
cyl.B.UNS
lizard
/nfs/tmp/my_other_data/cyl_vib
```

Limitations:

- FieldView requires that all partitions of each dataset have identical lists of solution variables and surface based (boundary) variables.
- The partition count across a transient set of PFPR layout files can not vary. If they vary, you will see the following error:
Time Series Rejected: Mismatch in the number of partitions.
All Layout files in a potential time series must specify the same number of partitions.
- The number of partitions per FieldView "worker" process may not exceed 1342. If exceeded, you will see the following error:
Partitioned File Error:
The number of partitions in the Layout file exceeds the maximum (1342) that can be read per process. Try running the server with more processes.
The data file has not been read: <path>file.layout

Detailed PLOT3D PFPR example

For this example, we will provide detail on the layout files needed to read a PLOT3D double precision unformatted grid and Q file which has been saved using 8 separate partitions. This PLOT3D dataset also has boundary surfaces associated with it, and, these boundary surfaces also have face based results (one value per element) stored on them.

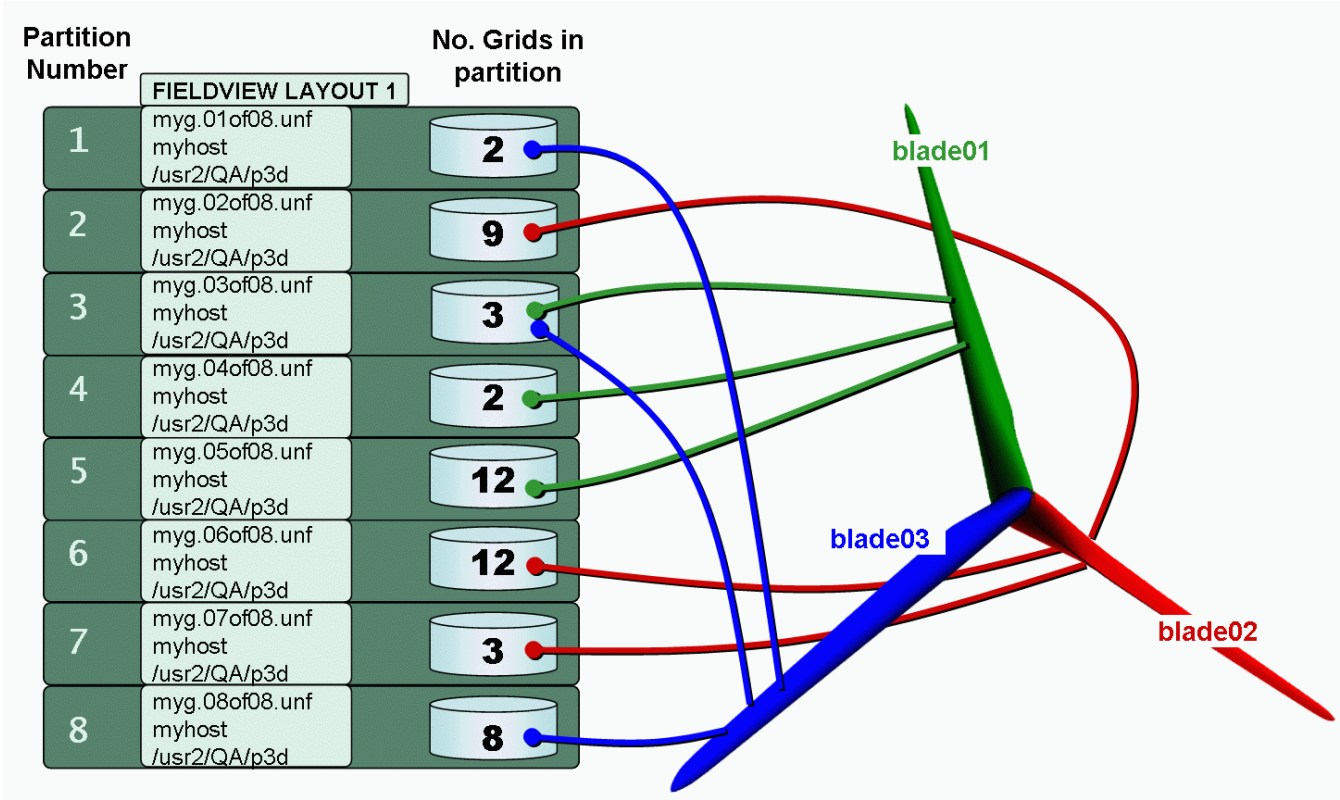


Figure 43 PLOT3D PFPR example for 8 partitions

In the figure above, the relationship between the layout file, the partitions, the grids, and the boundary surfaces is shown. Note that there are a varying number of grids stored in each partition. As noted previously, this particular problem also has several boundary types identified, namely blade01, blade02 and blade03.

- Boundary type blade01 has computational surfaces in partitions 3, 4 and 5,
- Boundary type blade02 has computational surfaces in partitions 2, 6 and 7, and,
- Boundary type blade03 has computational surfaces in partitions 1, 3 and 8.

Note that boundary types can span different partitions - **FieldView** is capable of correctly merging the computational surfaces together. Also note that one partition, partition 3, has computational surfaces which belong to two different boundary types.

The layout file used to read this partitioned case into **FieldView** is named `ovr_grid.layout`. The `fvbnd` file used to define the boundary surfaces and to specify the presence of surface based results is named `ovr_grid.layout.fvbnd`, listed below:

```
FVBND 1 4
blade01
blade02
blade03
BOUNDARIES
1 1 1 $ 1 $ 1 1 T 1
1 14 1 $ 1 $ 1 1 T 1
1 44 1 $ 1 $ 1 1 T 1
2 15 1 $ 1 $ 1 1 T 1
2 17 1 $ 1 $ 1 1 T 1
2 13 1 $ 1 $ 1 1 T 1
3 3 1 $ 1 $ 1 1 T 1
3 41 1 $ 1 $ 1 1 T 1
3 29 1 $ 1 $ 1 1 T 1
```

Before you attempt to read this dataset into **FieldView**, make sure that you have started a parallel server.

The general recommendation for best performance for reading any partitioned dataset is that the number of server processes needed is equal to one plus the number of partitions. So, in this case, one plus eight partitions means that `np` must be set to 9 in your parallel server config file. However, it is possible to read a partitioned dataset with fewer processes than partitions. Overloading the Partition File Parallel reader is discussed in further detail below.

Once you have started your parallel server, you can proceed to read the `ovr_grid.layout` layout file into **FieldView** as follows. From the PLOT3D Data Input panel, select DP unformatted as the file format, and turn on Multi-Grid and Iblanks in the Data Format section. Next, browse to select this file and hit OK.

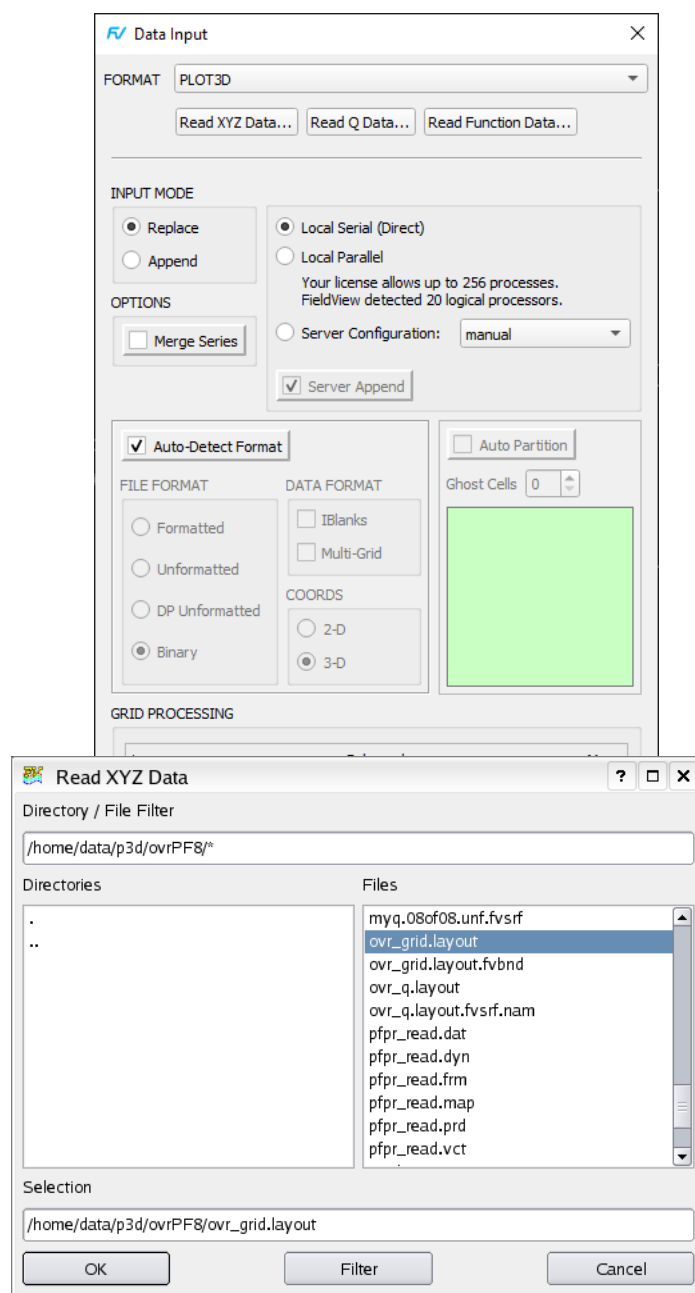


Figure 44 Reading a PLOT3D PFPR layout file

From **Figure 43**, note that there are boundary surfaces defined for each of the 8 partitions making up this dataset. And, note from the FVBND listing that there are surface based results for each computational surface. What this implies is that we must have a file containing surface based results for each partition. These surface based results files are associated with the volume results or Q files through

the standard naming convention of adding `.fvsrcf`. For this specific example, the Q files and surface based results files for each partition are:

<code>myq.01of08.unf</code>	<code>myq.01of08.unf.fvsrcf</code>
<code>myq.02of08.unf</code>	<code>myq.02of08.unf.fvsrcf</code>
<code>myq.03of08.unf</code>	<code>myq.03of08.unf.fvsrcf</code>
<code>myq.04of08.unf</code>	<code>myq.04of08.unf.fvsrcf</code>
<code>myq.05of08.unf</code>	<code>myq.05of08.unf.fvsrcf</code>
<code>myq.06of08.unf</code>	<code>myq.06of08.unf.fvsrcf</code>
<code>myq.07of08.unf</code>	<code>myq.07of08.unf.fvsrcf</code>
<code>myq.08of08.unf</code>	<code>myq.08of08.unf.fvsrcf</code>

The layout file for the results, `ovr_q.layout`, is very similar to the grid layout file. Since the file naming convention automatically associates the surface based results with the correct computational surface within each grid in the matching partition, these results are automatically read.

Finally, it is a requirement that the names of the surface based results must be the same for each partition. As a result, only one surface based results name file is needed. This file must follow the standard naming convention, `ovr_q.layout.fvsrcf.nam`, in order to be read when the `ovr_q.layout` is read. The listings for the `ovr_q.layout` and `ovr_q.layout.fvsrcf.nam` are provided below:

```

FIELDVIEW LAYOUT 1
myq.01of08.unf
myhost
/usr2/QA/p3d
myq.02of08.unf
myhost
/usr2/QA/p3d
myq.03of08.unf
myhost
/usr2/QA/p3d
myq.04of08.unf
myhost
/usr2/QA/p3d
myq.05of08.unf
myhost
/usr2/QA/p3d
myq.06of08.unf
myhost
/usr2/QA/p3d
myq.07of08.unf
myhost
/usr2/QA/p3d
myq.08of08.unf
myhost
/usr2/QA/p3d

```

Density (Q1)
 x-momentum (Q2)
 y-momentum (Q3)
 z-momentum (Q4)
 stagnation-energy (Q5)

Partition File Parallel Reader Overload

The fundamental requirement of a dataset in order for it to show any scaling performance with **FieldView** Parallel is that it must be made up of multiple grids. Datasets having multiple grids can have one of two file structures: Either all of the grids are stored in a single file or multiple files, each containing one or more grids, are used to partition the dataset. **FieldView** is capable of reading data and creating surfaces/rakes for both file structures. For optimized operation on partitioned datasets with relatively few partitions, **FieldView** can read one partition (or file) for each worker server process, as illustrated on the left.

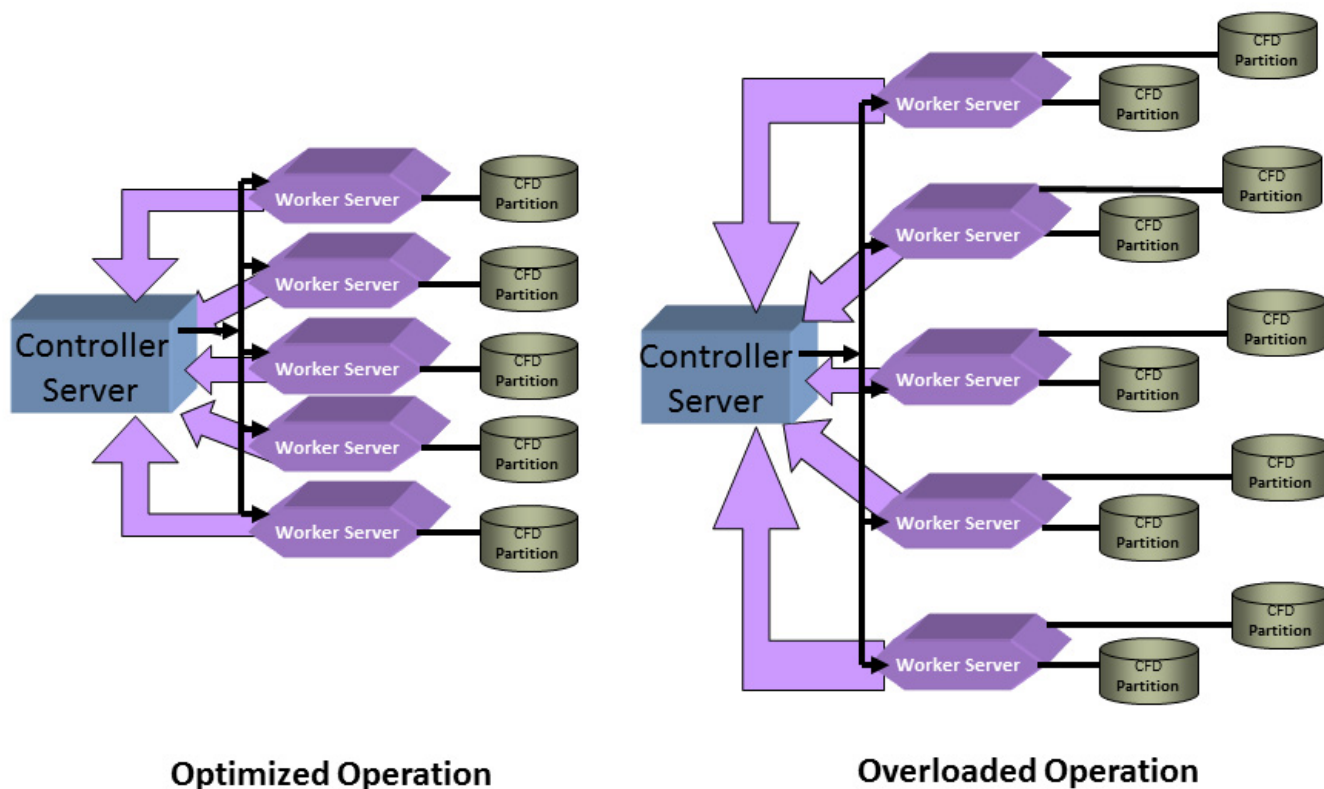


Figure 45 Partition File Parallel Operation


However, for partitioned datasets with hundreds of grids, it may be more practical to run **FieldView** in an overloaded mode, where each worker server is capable of reading more than one partition, as illustrated on the right. Overloaded operation for partitioned file parallel also overcomes the situation in which your parallel license provides for fewer processes than partitions. Overload operation is auto-

matic and requires no special settings. No changes to existing layout files, RESTARTS and **FVX** programs are needed.

Exports to FieldView Formats

AcuSolve

(www.altair.com)

A translator is available within AcuSolve to export results to the **FieldView** Unstructured File Format. Supp  Viewer Options... Demote: Simple formats. This is particularly useful for cases in which the grid data does not change with respect to time. Consequently, it is possible to save the grid file only once (for a transient sequence), and then save just the results data for each time step in the series. The export of **FieldView** Unstructured Files can be automated within AcuSolve.

CFD-ACE

(www.esi-group.com)

An export to the PLOT3D file format is available from this solver. This file format can be read directly into **FieldView**. Please review the documentation for this solver to find out more information concerning the export to the PLOT3D file format.

CFX

(www.ansys.com)

CFX data *must* be exported to the **FieldView** Unstructured data format in order to be read. This export capability is available with CFX.3 and later. CFX is a finite volume based solver. The **Field-View** Unstructured data format is nodal based. Interpolation from the cell centers to the nodes is carried out by CFX during the export process. This interpolation may sometimes lead to small errors when comparing integrated quantities such as mass flow rates or forces on boundaries.

All named boundaries within CFX will be included in the **FieldView** Unstructured File export. They are accessible as Boundary Types on the Boundary Surface panel within **FieldView**.

To export data, you will use the CFX Solver Manager. From the main level menu, choose Eile... and then select Export. This will bring up the Export Panel. On the export panel you can choose to read in a CFX results file to be converted. If you have just obtained a problem solution, this field will already be filled in. A default filename will have the extension `.fv` appended to it. Leaving the Export File field blank will *not* result in a successful export and will produce an error.

The export utility offers several options. First, you can select a **FieldView** version to export to. The range of choices is 6, 7 or 8. This is an effort on the part of CFX to match their export format with the **FieldView** version. Strictly speaking this is incorrect and does not follow the revision versions of the **FieldView** Unstructured File Format. All of these choices will create a file which can be read by any version of **FieldView** starting from 6 on.

Next, you can select an output level, ranging from 1 to 3. For a selection of 1, the number of scalars written will be minimal, and this selection offers the advantage of creating the smallest export file. If you select 3, this will produce the largest file, including nodal interpolated results for boundary scalars.

You can choose the option of: “Use Surface Data on Boundary Nodes.” The values at boundary nodes are part of the solution that the CFX Solver calculates. The values at the boundary nodes are dependent on the solution near these nodes. Hence, these values may be different than the specified boundary conditions set by the user. The user has the option to output the solved boundary node values *or* the specified boundary condition values. If the “Use Surface Data on Boundary Nodes” button is ON, then the boundary values are *corrected* and reset to the boundary condition values. If this option is OFF, then the solved boundary nodes are used and exported to the file.

For quantitative calculations, we recommend that uncorrected values should be used. Exporting data through the CFX Solver Manager as explained above uses *corrected* values by default (button is ON). CFX is a finite volume based solver. Interpolation from the cell centers to the nodes for export to the **FieldView** Unstructured File format is handled by the CFX Solver.

Exporting results for transient series will automatically create one **FieldView** Unstructured file for each time step. It is possible to limit the range of exported time steps by making appropriate selections on the Export panel.

COBALT

(www.cobaltcfd.com)

An option to export to the **FieldView** Unstructured File format is available directly within COBALT. This export option supports the use of split grid and results files. This is particularly useful for cases in which the grid data does not change with respect to time. Consequently, it is possible to save the grid file only once (for a transient sequence), and then save just the results data for each time step in the series. The export of **FieldView** Unstructured Files can be automated within COBALT.

COBALT is a finite volume based solver. Results are interpolated by COBALT to the nodes for export to the **FieldView** Unstructured File Format. All boundary surfaces defined within COBALT will be written out, and will be available as Boundary Types on the Boundary Surface panel within **FieldView**. On these surfaces, face based, or non-interpolated results will also be available. These results can be used for integrations of lift and drag - it is expected that these integrated results calculated by **FieldView** will exactly match those values reported by COBALT.

CONVERGE™

Native FV-UNS results, generated from the standalone post_convert program supplied by Convergent Science, Inc., can be read directly into **FieldView**, as illustrated in **Figure 46** below.

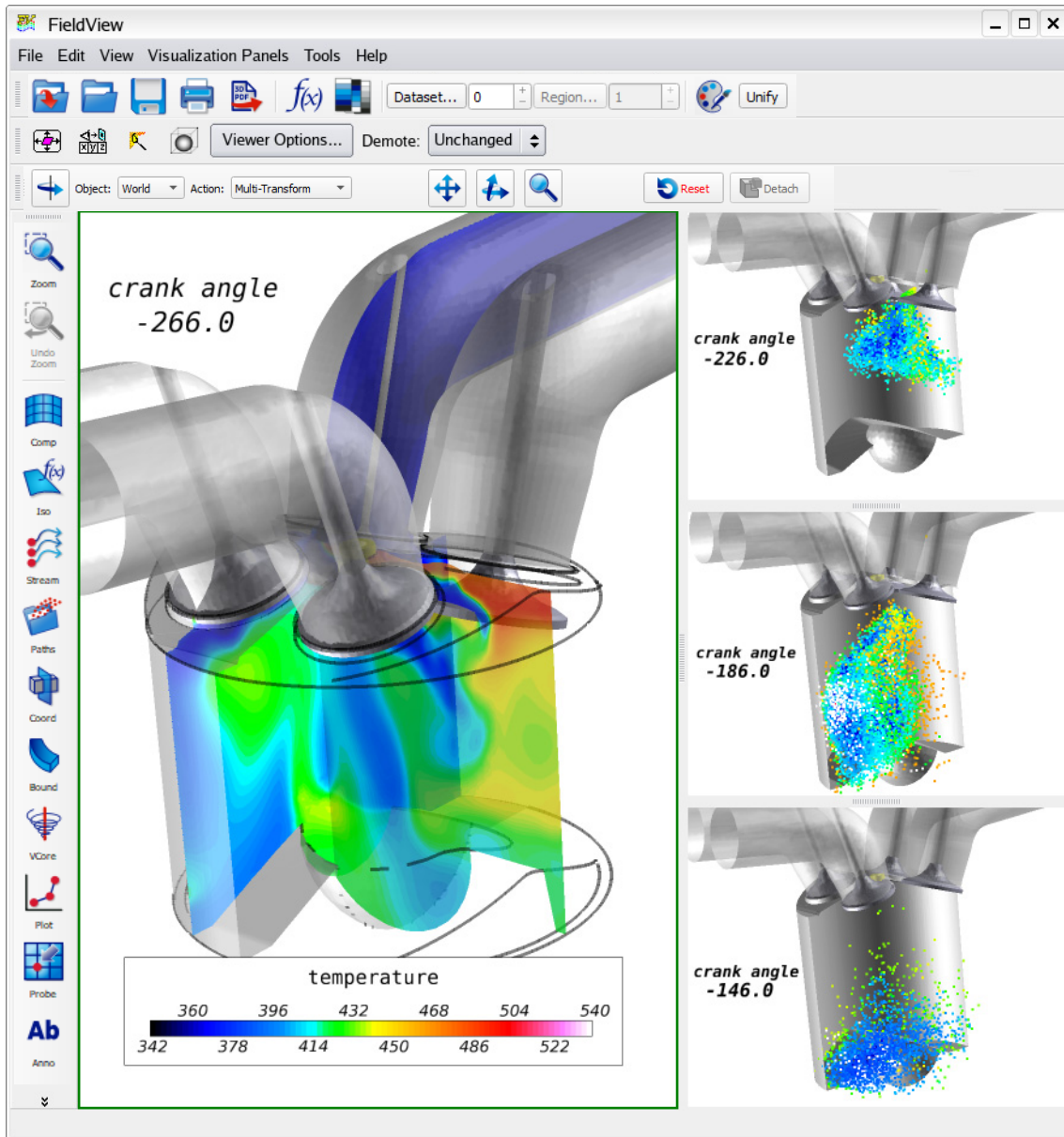


Figure 46 CONVERGE Internal Combustion Spray Modeling

Volume data and spray combustion data, exported using the **FieldView** Binary Particle Set format, can be visualized simultaneously. Transient cases are automatically recognized as such on read-in, and time synchronization between the volume data and the spray droplet data is automatic.

Note: The solution time field in the **FieldView** unstructured file is used to store the crank angle. Consequently, an annotation with the **escape sequence**, %%T, can be used to show the crank angle, and will update during a transient sweep.

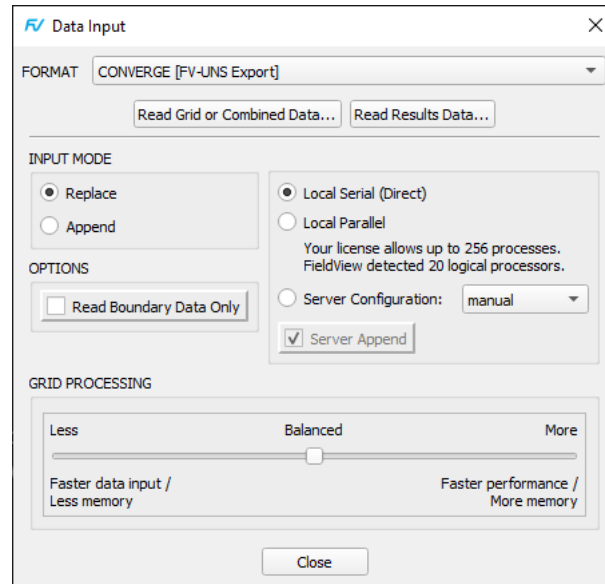


Figure 47 CONVERGE [FV-UNS Export] Data Input Panel

To read CONVERGE results, start by selecting the Read Grid or Combined Data... button on the Data Input panel (**Figure 47**). You will be presented with a file browser which will let you navigate to the location of the **FieldView** unstructured .uns file that you wish to read. For transient cases, you will see a series of datasets. Selecting one of these datasets will be sufficient for **FieldView** to recognize the series of data as transient.

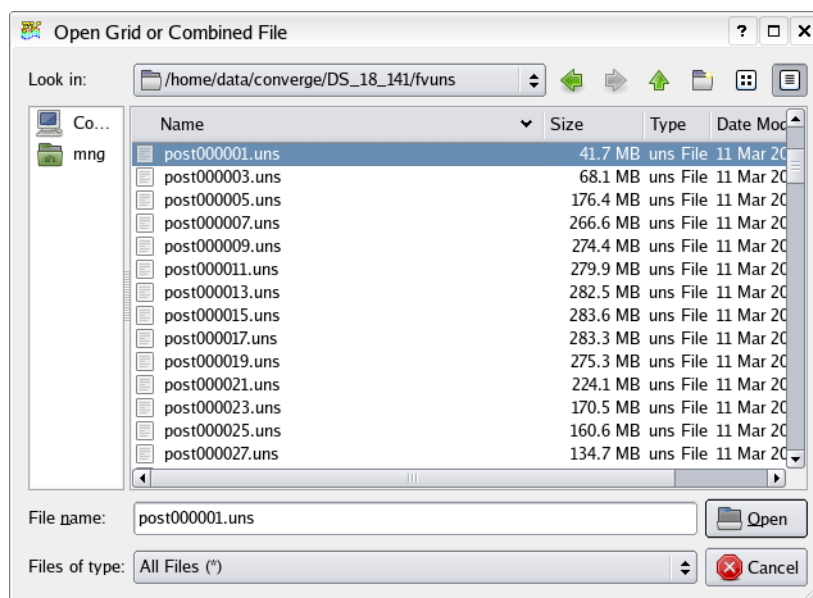


Figure 48 CONVERGE [FV-UNS Export] File Browser

Once a file selection has been made, click the Open button. If the case is transient, you will see the Transient confirmation popup, as illustrated in **Figure 49**.

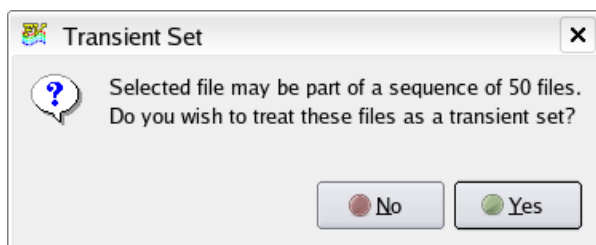


Figure 49 Transient Set Confirmation

To read spray droplet data, navigate to the Particle Paths visualization panel. Click on the Import... button. The Particle Path Data Input panel will provide access to a file browser where a selection for a particle path set can be made. This set of steps is illustrated in **Figure 50** below.

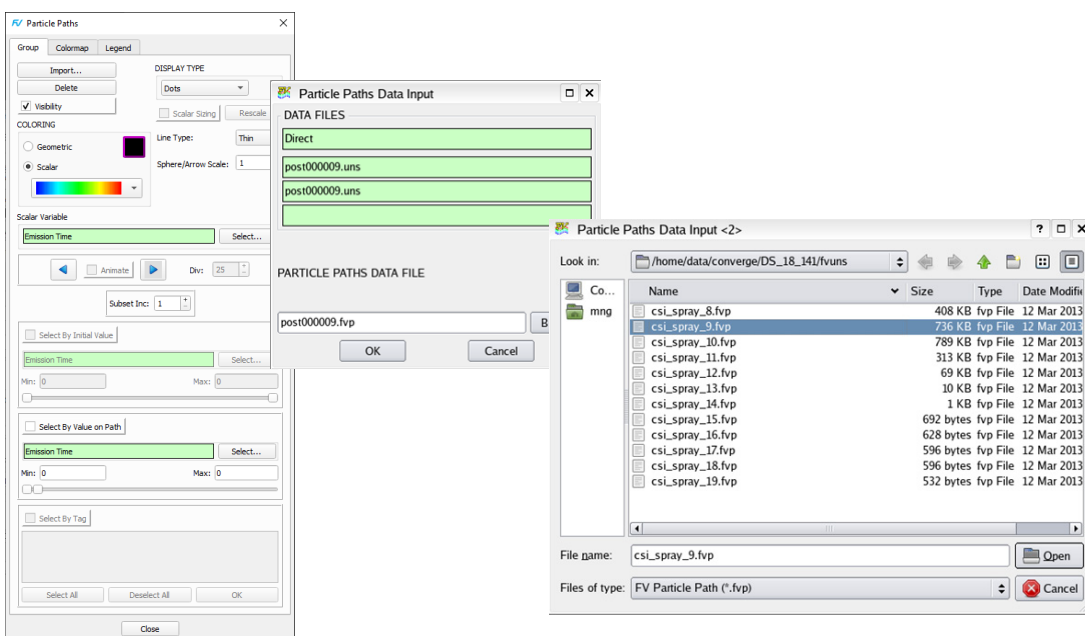


Figure 50 Importing Spray Droplet data

Once a particle path set file has been read, it will be automatically synchronized in time with the volume data on the basis of the filename. Further details are provided in the [Streaklines](#) section of [Chapter 6](#).

Known Limitations

A parallel-compatible **FieldView** unstructured data format is currently not supported by the post_convert export utility.

DROP3D

(www.ansys.com)

An export to the FieldView Unstructured File format is available from this solver. See the documentation for this solver to find out more information concerning this export

FENSAP

(www.ansys.com)

An export to the FieldView Unstructured File format is available from this solver. See the documentation for this solver to find out more information concerning this export.

FIDAP

(www.ansys.com)

Versions of FIDAP for release 8.x and later will support the export to the **FieldView** Unstructured File Format. This export is a specially marked file which can be read by the general **FieldView** package which is supplied by **Tecplot Inc.**

FIDAP is a finite element based solver. There is no interpolation between FIDAP and the export to **FieldView**, so you should expect integrated quantities to match.

Any boundary surfaces defined within FIDAP will be correctly exported, and will be available as Boundary Types on the Boundary Surface panel within **FieldView**.

Fine/Turbo

(www.numeca.be)

An export to the PLOT3D file format is available from this solver. This file format can be read directly into **FieldView**. Currently there is no support to carry over boundary surface information. Also, surface based or face based results are not written out. Please review the documentation for this solver to find out more information concerning the export to the PLOT3D file format.

FLUENT

(www.ansys.com)

To read data from FLUENT into **FieldView** using the FLUENT [FV-UNS Export] entry on the Data Input menu, it must be first exported. We recommend that the **FieldView** Unstructured file format (*.fvuns) be used for all 3d and 3ddp (double precision) data. The export to the **FieldView** Unstructured file format also works for the parallel version of FLUENT. For FLUENT 6.3.26 and earlier, the **FieldView** Case+Data (*.fvc and *.fvd) export option, available only thru the TUI, should not be used as it is now no longer supported. Only legacy **FieldView** Case (*.fvc) and Data (*.fvd) exported files can be read using the native RAMPANT-FLUENT/UNS interface within **FieldView**.

To output **FieldView** Unstructured files from FLUENT, select the Export... item under the File pull-down menu of the FLUENT Solver interface. This interface is shown in **Figure 51**.

Press this button to bring up the Export panel which will allow you to export your results in the **FieldView** Unstructured data format.

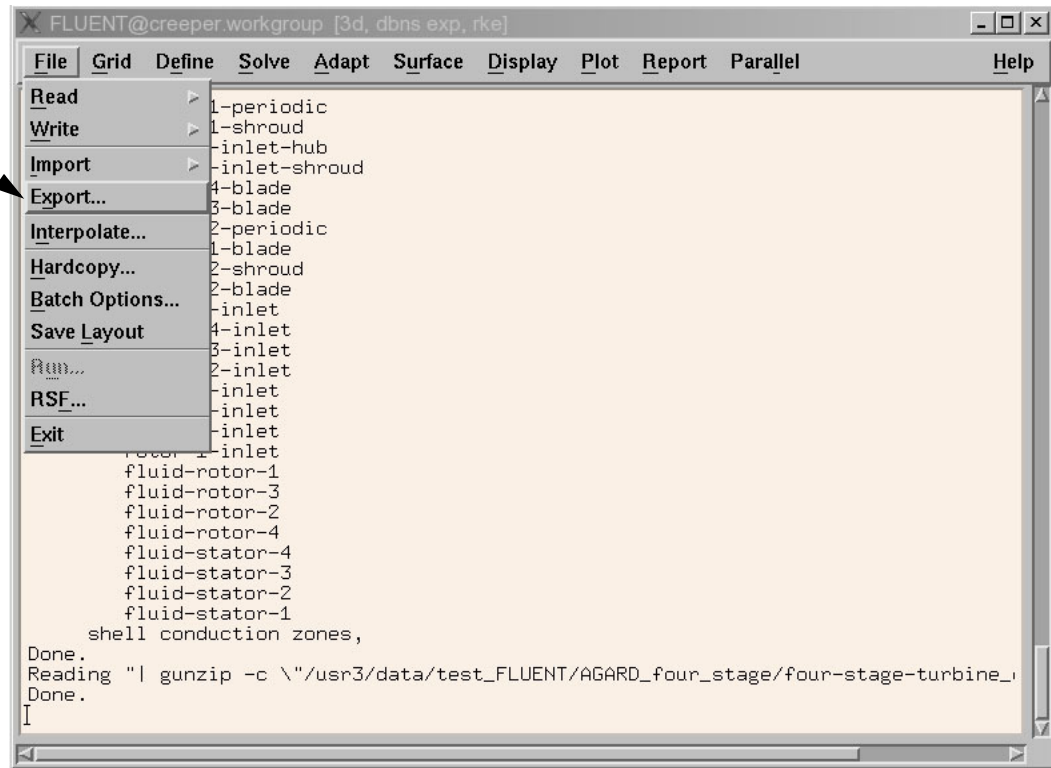


Figure 51 FLUENT Export Pull-down

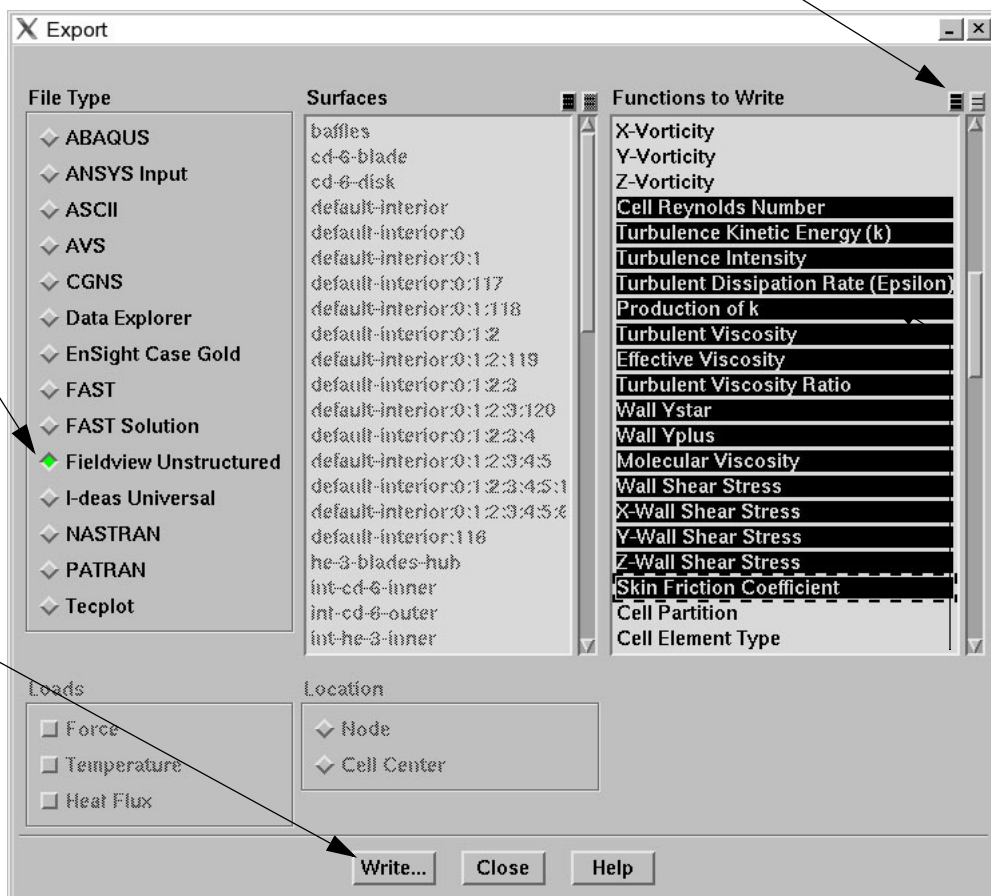
Choosing the **Export...** item on the **File** menu brings up the Export panel. This is where exporting takes place. The Export panel is shown in **Figure 52**.

From the export panel, select the functions to write out to the exported file. By default, the exported file will have the same root name as the data in memory. Therefore, if the data is called `transit.cas/transit.dat`, then the resulting **FieldView** output file will be `transit.fvuns`.

Choose this option to output data in the **FieldView** Unstructured data format.

These buttons are use to select all functions (left) or to deselect all functions (right). Selecting all functions is not recommended. See accompanying text.

Press this button to export the results. This will bring up a file browser showing the default output filename. See **Figure 53**.



This section allows you to select which functions you wish to be exported. The list of functions shown will vary depending on the model. Note: The X, Y, and Z Velocity functions are automatically exported and should not be selected. See accompanying text.

Figure 52 FLUENT Export Panel

By default, the velocity vector (X, Y, and Z Velocity) is always exported. Therefore, if the entire “Functions to Write” section is deselected, then the only output will be the velocity vector.

Warning: Since the velocity vector (X, Y, and Z Velocity) is automatically exported, *do not* select it from the “Functions to Write” panel. Doing so will result in multiple velocity vectors being exported to the file, causing function conflict pop-ups when the data is read into **FieldView**.

FLUENT is a finite volume based solver. Interpolation from the cell centers to the nodes, necessary for exporting to the **FieldView** Unstructured file, is handled by FLUENT.

The variables that are listed in the “Functions to Write” field will depend on the model. If the physical model includes turbulence, discrete phase modeling, radiation modeling or chemical reactions, for example, additional functions will appear. When the desired functions to export are selected, press the Write button. This will bring up a browser that will allow you to change the root name of the output file and change the directory to which the file is written.

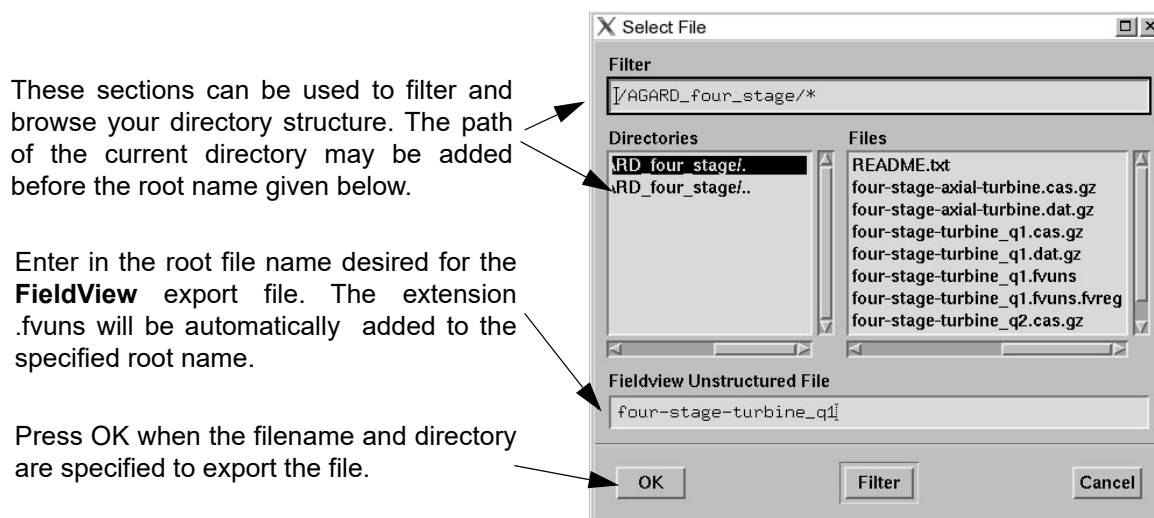


Figure 53 FLUENT Export File Browser

Regions are supported with the export to the **FieldView** Unstructured File format. You should expect to see a file with the extension `.fvreg` created when you create an export. The region file contains information on the grids in the model, and it can be used by **FieldView** to provide additional functionality. A full discussion of the Region File is provided in [Chapter 3](#). Note that at present, the FLUENT export creates a Version 1 FVREG, or region, file.

Transient Data

For transient data, we recommend that you make use of the FLUENT Text User Interface (TUI), and the Execute Command feature to automate the writing of **FieldView** Unstructured files for a given set of time step intervals. To bring up this form select the `Execute Command...` item under the `Solve` pull-down menu of the FLUENT Solver interface. The root filename and the actual variables to be written are included in the Execute Command.

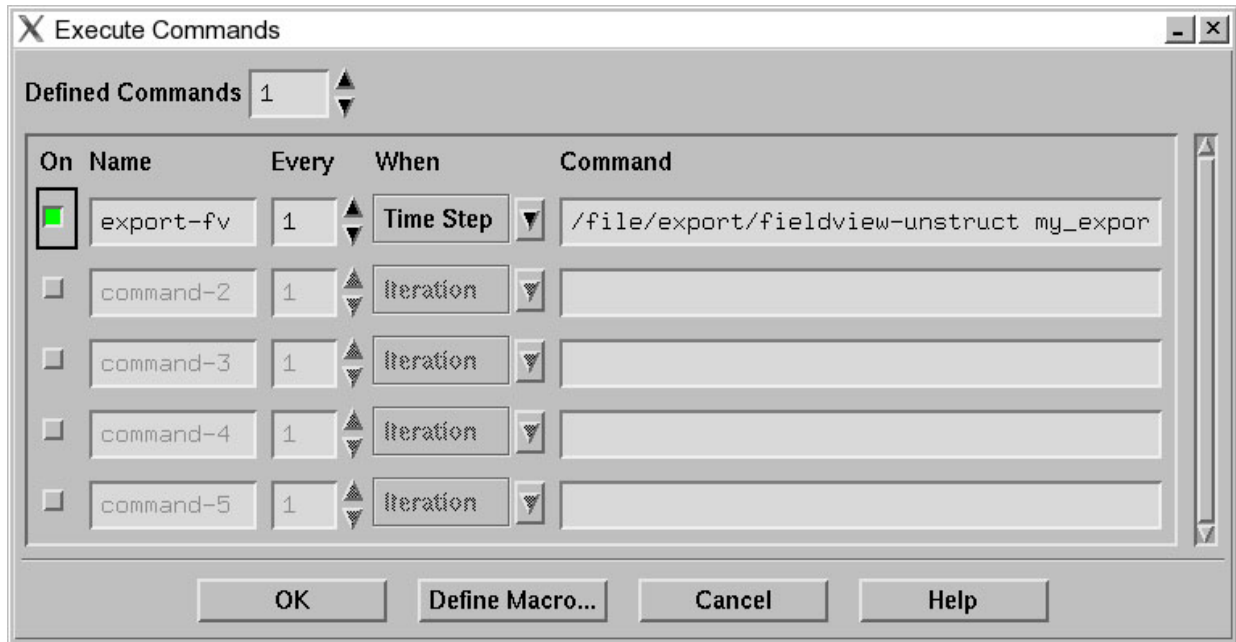


Figure 54 FLUENT Execute Command GUI

At the interface prompt within the FLUENT window, you can issue the command

```
> file/export/fieldview test
```

and hit return twice. This will create a list of all of the available scalars and vectors which can be exported to the **FieldView** unstructured file format.

To export the data and scalar variables, you need to include this information in the Command line:

```
/file/export/fieldview-unstruct my_export my_list_of_scalars () * () q
```

If you include a "%3t" to the end of the filename, FLUENT will automatically export the time step. For example, if you wanted to export the files at each time step along with scalars of pressure and temperature, the command line:

```
/file/export/fieldview-unstruct my_export%3t () * () pressure temperature q
```

will automatically produce these files:

```
my_export001.fvuns
my_export001.fvuns.fvreg
```

```
my_export002.fvuns
my_export002.fvuns.fvreg
my_export003.fvuns
my_export003.fvuns.fvreg
...etc
```

Note: The `.fvreg` extension is the region file associated with the dataset. For more information about region files, please see [Chapter 3](#) of the **Reference Manual**.

Exporting Particle Trajectories

Steady State Case

For steady state cases, the particle trajectories can be exported with the FLUENT Export Particle Data panel that is available in the Solve/Particle History sub-menu (**Figure 55**). Choose the injections that you want to export and click on Write and make sure that the file is saved with the `.fvp` extension.

Note: The standard FLUENT export for two-phase data is based directly on the native **FieldView** Particle Path format. No conversions are required to read this type of data.

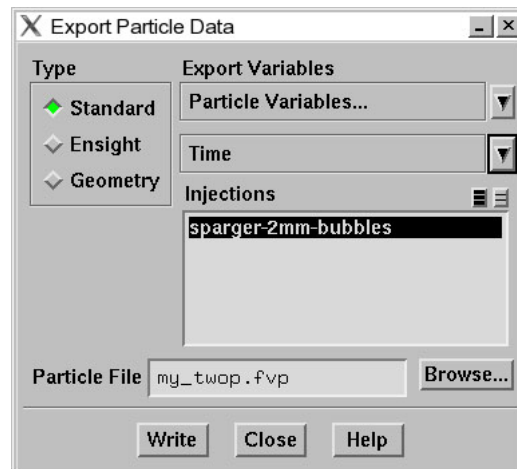


Figure 55 Export Particle Data Panel

To save selected particles (only what is displayed), first display the particles that you want exported in the Display/Particle Tracks panel and then write out the file with the Export Particle Data panel.

Standalone Particle Translator for Steady State Cases

A standalone translator is also available to convert FLUENT Particle Trajectory reports into **FieldView** particle path files. Again, this provides the ability to visualize two-phase flow results with the exported FLUENT files. The FORTRAN 77 source (`fluent2fvp.f`) and executables for each of the **FieldView** hardware platforms are located in the `/FLUENT` directory of the `/translators` directory on the **FieldView Installation** CD-ROM. The `/translators` directory is *not* installed by default and the desired files should be copied to your system. The translator `fluent2fvp` will convert FLUENT version 5.2 (and higher) trajectory reports to the **FieldView** ASCII Particle Path format. This format is described in [Appendix L](#) of the **Reference Manual**. Once the file is converted into an `*.fvp` file, you can import it with the Particle Path panel.

Transient Case

FLUENT will automatically save the particle path file when the **FieldView** Unstructured files are saved via Execute Command. When the solution iterates, FLUENT will dump the particle path data to the file based on the time you write your **FieldView** Unstructured file as specified in the Execute Commands.

These are the steps to initialize the export of the transient particle path data:

1. Initialize your dataset.
2. Initialize the injections by choosing Reset DPM Sources.
3. Open the Export Particle Data panel available in the Solve/Particle History sub-menu (**Figure 56**). Choose the injection that you want exported and click on Apply.

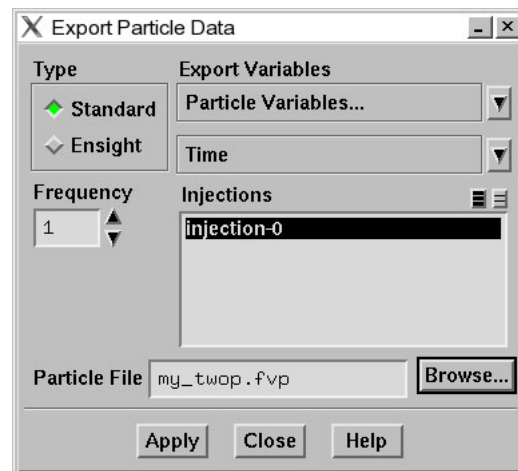


Figure 56 Export Particle Data Panel

4. Enter a file name with the `fvp` extension (**Figure 57**), choose OK and close the Export Particle Data panel.

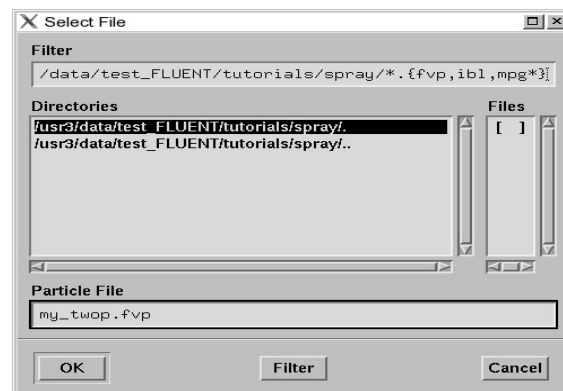


Figure 57 Save File Panel

FUN3D

(<https://fun3d.larc.nasa.gov>)

The NASA FUN3D software package is an unstructured CFD code that has been developed for a wide range of applications which may include incompressible, compressible or hypersonic flow. Solution approaches include steady and unsteady calculations with single or multiple species in the fluid domain. Prominent applications by NASA include studies of airframe noise, space transportation vehicles, flow control devices and rotorcraft. Advanced algorithms are employed to address dynamic grid adaption, design optimization, fluid structure interaction, design optimization and adjoint-based error estimation. The parallel efficiency of the solver has been well developed for application to many large-scale systems.

Volume variable data can be exported to the following file formats: **FieldView** unstructured (FV-UNS), VTK, CGNS and Tecplot Binary (FUN3D 13.5 manual, B.4.29 *&volume_output_variables*). Boundary and sampling variable output can be exported to Tecplot Binary files (FUN3D 13.5 manual, B.4.30 *&boundary_output_variables* and B.4.31 *&sampling_output_variables*). The **FieldView Reference Manual** includes relevant sections for each of these file formats and the Tecplot Binary file format can be read with the [Ensign Reader](#).

The benefits of **FieldView**'s parallel efficiency can be realized if the solver writes the data export with data partitioned into multiple blocks within one file or partitioned into multiple files. The **FieldView** unstructured (FV-UNS) format is the only file type that is enabled for a Parallel Read operation (see [FieldView Parallel Datasets](#)). **FieldView**'s [Partitioned File Parallel Reader \(PFPR\)](#) capability can be used, with any file type read by **FieldView**, for a set of files written on a per-solver-process basis. The volume variable output section of the FUN3D manual describes some file formats written in this manner. The PFPR layout-file should be manually created by the user if it is not already written by the solver.

GASP

(www.aerosft.com)

An export to the PLOT3D file format is available from this solver. This file format can be read directly into **FieldView**. Currently there is no support to carry over boundary surface information. Also, surface based or face based results are not written out. See the documentation for this solver for more information on the export to the PLOT3D file format.

POLYFLOW

(www.ansys.com)

Versions of POLYFLOW for release 3.10.x and later will support the export to the **FieldView** Unstructured File Format. This export is a specially marked file which can be read by the general **FieldView** package supplied by **Tecplot Inc.**

POLYFLOW is a finite element based solver. There is no interpolation between POLYFLOW and the export to **FieldView**, so integrated quantities are expected to match.

Any boundary surfaces defined within POLYFLOW will be correctly exported, and will be available as Boundary Types on the Boundary Surface panel in **FieldView**.

STAR-CCM+

(www.cd-adapco.com)

An export to the native **FieldView** Unstructured (FV-UNS) file format is available from this solver. Scalars and vectors can be chosen from a complete list with the STAR-CCM+ interface. Face based results are supported with this export, and all boundaries defined within STAR-CCM+ are written to the exported file.

Tetrex

(www.tetraresearch.com)

An export to the native **FieldView** Unstructured (FV-UNS) file format is available from this solver. This file format can be read directly into **FieldView**. Boundaries defined in Tetrex are written to the exported file. Currently, surface based or face based results are not written out. Refer to the documentation for this solver for more information on export to the FV-UNS format.

ThermoAnalytics

(www.thermoanalytics.com)

An export to the native **FieldView** Unstructured (FV-UNS) file format is available from this solver. This format can be read directly into **FieldView**. Boundaries defined in ThermoAnalytics models are written to the exported file. Currently, surface based or face based results are not written out. Refer to the documentation for this solver for more information on export to the FV-UNS format.

Exports to FieldView Parallel Compatible Formats

Datasets must contain multiple grids to demonstrate any performance benefits of FieldView Parallel (unless the **Auto Partitioner** is used). This is because the grids making up the entire dataset are distributed across the available server processes. Elements within each grid may vary significantly from one grid to the next. Optimum parallel performance for reading datasets is realized when the elements within each of the grids are evenly distributed across each of the available worker processes. **FieldView** will attempt to balance the load across the available processes by applying the rule to distribute the grids successively to the least loaded process. For example, consider a multigrid dataset composed of several grids, with one of the grids containing many more elements than the others. **FieldView** will use one process to read the grid with the large number of elements, and then distribute the rest of the grids over the remaining processes. As a general rule, it is best to have grids containing a relatively equal number of elements. And, it is preferable to have many more grids than available server processes.

Multiple grid PLOT3D cases are directly compatible with **FieldView** Parallel, and most of the solvers creating data having this format (i.e. OVERFLOW) will generate datasets of this type.

Datasets which use the **FieldView** Unstructured (FV-UNS) file format may or may not be written to contain multiple grids. To obtain performance benefits for FV-UNS datasets with **FieldView** Parallel, multiple grids are required. A brief summary of commercial solvers that write or export FV-UNS files, and whether they support multiple grids follows:

Solver	Compatible w/ FieldView Parallel	Notes
AcuSolve	YES	Use <code>acuTrans -fvr subdomain</code> to apply solver partition information to create multiple grid FV-UNS exports.
CFD++	YES	Use <code>genplif</code> with the <code>fieldviewmgb</code> (FieldView multigrid binary) option to maintain the solver partition information in the FV-UNS export.
CFD-ACE	Unknown	Multigrid export is currently unavailable.
CFX	YES	Contact ANSYS CFX support.
COBALT	NO	Multigrid export is currently unavailable.
DROP3D	Unknown	Contact Ansys for more information.
FIDAP	NO	Multigrid export is currently unavailable.
Fine/Turbo	YES	Multigrid exports based on the PLOT3D format will be compatible with FieldView Parallel.
Fire	YES	Contact AVL directly to obtain the FV-UNS translator utility.
ANSYS Fluent	YES	Each cell zone is split into np grids, where np is the number of parallel processes used by the ANSYS Fluent session
FENSAP	Unknown	Contact Ansys for more information
FrontFlow	YES	This solver creates FV-UNS Partition files.
GASP	Unknown	
POLYFLOW	NO	Multigrid export is currently unavailable.
STAR-CCM+	YES	Each region is split into np grids, where np is the number of parallel processes used to read the dataset.
USM3D	NO	Multigrid export is currently unavailable.

Solvers capable of Multigrid FV-UNS exports

Standalone Translators to FieldView Formats

BANFF

(www.reaction-eng.com)

A translator to convert to the PLOT3D file format is available from this solver. This file format can be read directly into **FieldView**. Boundaries defined in BANFF are supported as part of the translation. Currently, surface based or face based results are not written out. Please review the documentation for this solver to find out more information concerning the export to the PLOT3D file format.

CFD++

(www.metacomptech.com)

A translator to convert results to the **FieldView** Unstructured File format is available for this solver. Please contact Metacomp directly to obtain information on this translator.

CFX-TASCflow

(www.ansys.com)

We have included an executable (`tasc2p13d`) and documentation that will allow you to convert native CFX-TASCflow, grid (`grd`) and results files (`bcf`, `rso`) into PLOT3D file formats which can be read into **FieldView**. The executable and documentation can be found in the `/tascflow` subdirectory of the `/translators` directory on the **FieldView** DVD. Note: the contents of the `/translators` directory are *not* installed by default and will have to be manually copied to your hard disk from the DVD.

COBALT₆₀

A standalone translator is available to convert the government version of Cobalt₆₀ datasets into **FieldView** Unstructured files. Please contact the Air Force Research Lab where you obtained Cobalt₆₀ to acquire this translator.

PowerFlow

(www.exa.com)

A stand-alone translator is available from EXA to convert their files to the native **FieldView** Unstructured File Format. Please contact EXA to obtain this translator.

FIRE

(www.avl.com)

A translator is available from AVL to convert their output into the **FieldView** Unstructured File Format. Please contact AVL directly to obtain this translator code.

GLACIER

(www.reaction-eng.com)

A translator to convert to the PLOT3D file format is available from this solver. This file format can be read directly into **FieldView**. Boundaries defined in GLACIER are supported as part of the translation.

Currently, surface based or face based results are not written out. Please review the documentation for this solver to find out more information concerning the export to the PLOT3D file format.

USM3D

A stand-alone translator is available from NASA to convert their files to the native **FieldView** Unstructured File Format. Please contact **Tecplot Inc.** for information on how to obtain this translator.

VECTIS

(www.ricardo.com)

A translator is available to convert VECTIS 3.8 files to the **FieldView** Unstructured File format. This translator is capable of handling arbitrary polygons.

All named boundaries within VECTIS 3.8 files will be included in the **FieldView** Unstructured File export. They are accessible as Boundary Types on the Boundary Surface panel within **FieldView**.

This translator code will be available on the **FieldView** DVD in the `/translators` directory.

User Defined Plugin Readers for FieldView

This feature is used to specify and read in geometric data, results data, or both together, in your own format. For more information about user-defined readers, consult [Chapter 9](#) of this **Reference Manual**. Some general comments concerning the use of Plugin Readers follow here.

There are two key Plugin Toolkit features. The first is to provide support to be able to read a dataset with the format of having one single file for each transient time step. This change should eliminate problems associated with very large files generated for transient cases with many time steps, and for transient cases with large meshes. The second improvement is to provide internal support for native arbitrary element handling. The capability to handle native arbitrary elements is fully supported. The Plugin Toolkit has been automatically extended to cover this feature.

Some commercial solvers have written their own User Defined Plugin Readers to enable the reading of their file formats into **FieldView**.

AVUS

This code, formerly COBALT₆₀ (Government Version) has a User Defined Reader available. Please contact **Tecplot** support to obtain information on this reader.

User Defined Plugin Readers for FieldView Parallel

User Defined or Plugin Toolkit readers support operation in parallel. On a related note, all of the current plugin toolkits supplied with **FieldView** (`acusolve_reader`, `cgns_reader`, `flow3d_reader`, `fluent_reader` and `ls-dyna_reader`) will already have support for **FieldView** parallel. However, at present, there is no data available for these readers which is compatible with parallel.

The API for registering a User-Defined Data reader is located in one of the following two files:

`ftn_register_data_readers.f` (FORTRAN)

`register_data_readers.c`

These files are located in the `/user` subdirectory of your **FieldView** installation, and they contain additional comments on how to register a user defined data reader.

Single file multigrid parallel

For single file multigrid parallel, **FieldView** assigns different subsets of the grids to different parallel processes (different worker server processes inside a **FieldView** parallel server). In this way, the work of reading and post-processing the data is distributed and load-balanced. **FieldView** data readers are divided into a "query" phase (which returns certain summary information such as the number of nodes in each grid), and a "data read" phase which reads one grid at a time (the grid number is supplied to the data read phase). It is required that your data reader can read a subset of the grids in a dataset. These grids will be in ascending order, but there will be gaps. For example, the data read phase may be asked to read grid 3 (skipping over grids 1 and 2), then grid 5, and then grid 9. If your data reader supports reading selected grids like this, you can enable parallelization by setting the `FV_GRID_PARALLEL_READER` option as described in `register_data_readers.c` and `ftn_register_data_readers.f`.

If your data reader is slow to skip over grids (such as having to read significant portions of grid 4 in order to skip from grid 3 to grid 5), then you may not get any parallel speed-up during the grid read phase. However, you should still get parallel speed-up during many post-processing operations, such as creating surfaces.

The query phase is currently called on all parallel (worker) processes. Therefore, if your query phase is slow, you may not get any parallel speed-up during your data read. However, you should still get parallel speed-up during many post-processing operations.

Partitioned file parallel

Partitioned-file parallelization is for datasets that are split into "partition files", each of which contains a subset of the entire dataset. For example, a parallel solver may split the dataset into partitions, assign a partition to each sub-process in the solver, and then write each partition into a separate file.

In partitioned-file parallelization, **FieldView** assigns each partition to a different parallel process (a different worker process inside the **FieldView** parallel server). The assignment of partitions to server

processes is controlled by a "layout" file, which is simply a text file that lists the partition files and which host machines should process each partition; see the section in this chapter [Description of Layout File Format](#).

Unlike grid-file parallelization, all user-defined data readers automatically support partitioned-file parallelization. There is no need to set any special data reader registration options.

Each **FieldView** server worker process only sees the single partition assigned to that process, so it behaves like an ordinary non-parallel data reader. The extra work of splitting or merging the operations on the dataset is done automatically by **FieldView** using the information in the layout file.

There are restrictions on the partition files. All of the partition files in a single dataset must have the same variable names (including boundary variable names if these are present). However, the partition files can have different boundary types; the boundary types will be automatically merged by **FieldView**.

Partitioned-file parallel does not support grid subsetting by the user during the data read. If the reader has enabled this, it is forced off.

There are no special requirements for efficiency. However, if all of the partition files are located in the same filesystem, then parallel speed-up during data reads can be hurt by competition for access to the filesystem.

Unsupported features for Parallel Data Readers

The following features are not supported for single file multigrid parallel or partitioned-file parallel data readers.

The following functions cannot be called from inside a parallel data reader:

```
fetch_element
fetch_element_ex
ftn_fetch_element
ftn_fetch_element_ex
```

If they are called from a parallel data reader, they will return an error code (-1 for failure, instead of 0 for success). These functions can be called from inside user-defined functions (parallel or not), just not from parallel data readers. Grid numbers inside each parallel process are local to that process; they are not the same as the grid numbers seen in the **FieldView** user interface. The grid number passed as an input argument to the user-defined functions is this kind of localized grid number. However, you can pass this localized grid number to the `fetch_element` family of functions, and they will return correct values for the grid. Be careful about using this grid number for anything except calling the `fetch_element` family.

Temporary region files created by the data reader are not supported. Therefore, if a parallel data reader calls:

```
open_tmp_fvreg  
ftn_open_tmp_fvreg
```

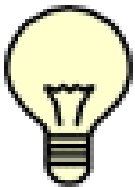
then these functions will return an error code (-1 for failure, instead of 0 for success).

Parallel data readers, including the PLOT3D and **FieldView** Unstructured readers provided with **FieldView**, do not support the following **FieldView** features:

- Dataset Sampling
- Create Wall Boundaries
- Create Exterior Boundaries

DataGuide™ is supported for partitioned file parallel readers (PFPR), but not for single-file multigrid parallel readers.

Server Append and Dataset Comparison (via Server Append) are supported as of **FieldView** 14.



Reminder:

The data file formats currently supported for **FieldView** Parallel:

1. PLOT3D (binary, unformatted and double precision)
2. FV-UNS (binary, combined and split grid & results)
3. All plugin readers

To realize any benefits from parallel, such as speed-ups for creating or sweeping coordinate planes or isosurfaces, the dataset(s) must first be read using **FieldView** parallel.

Chapter 2

Functions

2

The Function Specification panel contains four registers for holding various data values. These four registers are: Iso-Surface, Scalar, Vector and Threshold. Each one may contain: a variable that has been read-in from a file, a formula that was created with the Formula Specification panel, a user-defined function, an intrinsic **FieldView** function (e.g. `atan(y/z)`) or a PLOT3D function. Functions are separately stored in **FieldView** for each dataset when multiple datasets are read in. When the Function Selection panel is brought up, it will only have the functions available for the *current* dataset displayed. Functions *cannot* be used or transferred from one dataset to another.

The Function Specification panel is also used to define new scalar or vector quantities by accessing the Formula Specification panel.

Function Specification Panel

This panel is used to define which variables are to be used when creating surfaces, as well as to define any new variables. If you press any of the Iso-Surface, Scalar, Vector or Threshold buttons, the Function Selection panel is presented. The Function Selection panel is used to pick the variable to load into the selected register. Note that for all registers other than Vector, only scalar variables will be displayed. For the Vector register, only vector variables will be displayed.

The Create and Edit buttons are used to define new variables, using the Formula Specification Panel. The new variables may be defined from any of the variables currently in memory, constants, and any of a series of pre-defined math functions.

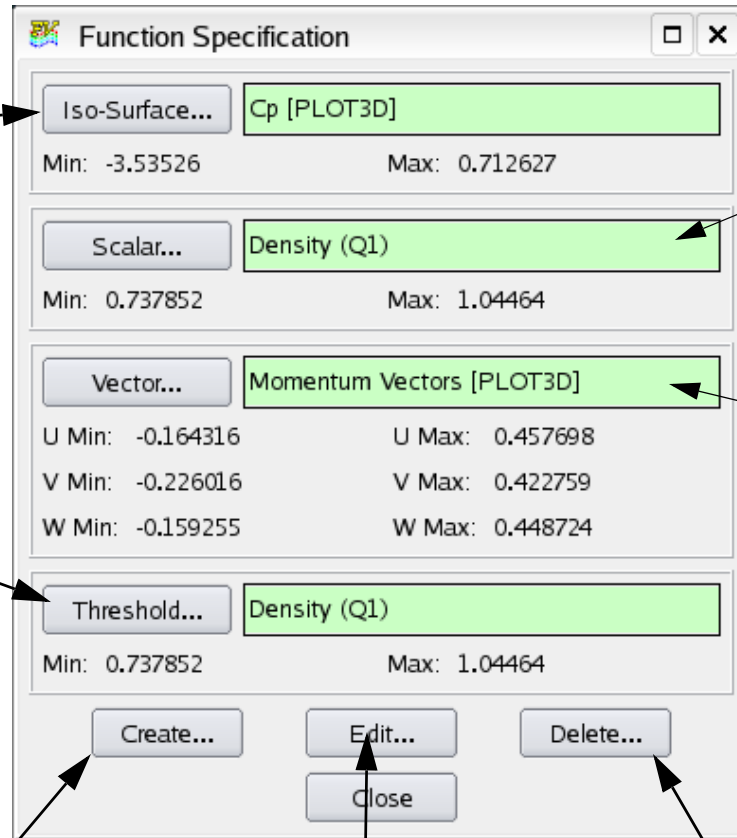
Face Data and the Function Specification Panel

The function Min and Max values shown on the Function Specification panel (**Figure 58**) will normally show the global min/max values for the entire nodal dataset. However, when face data exists (for either PLOT3D or **FieldView** - Unstructured data) and are being used on the current boundary surface, then the min/max fields will show the min/max values for the face data. These min/max values will be those for *all* boundaries, not just those for the *current* boundary surface (if there is one).

Iso-Surface will load a scalar into the iso-surface register. Iso-surfaces are surfaces of constant value, such as regions of constant temperature and pressure.

Threshold will load a scalar into the Threshold Register. This function is used to limit the boundaries of surfaces to areas within a range of values for a quantity, such as temperature or pressure.

Create will bring up the Function Formula Specification panel. This panel is used to create a new function based on any previously defined functions, constants, functions read from an external file or predefined math functions.



Scalar will load a scalar quantity, such as temperature or pressure, into the scalar register.

Vector will load a vector function into the vector register. Vector functions are used for streamline integration.

Delete is used to delete a Formula. Note that a Formula may not be deleted if it is in use, which means that it is currently loaded into a register, is being used to define one or more surfaces or is used in another formula.

Figure 58 Function Specification Panel

Face Data and the Function Selection Panel

If either PLOT3D or **FieldView** - Unstructured face data has been read in, then the Function Selection panel (**Figure 59**) will list face results (either scalars or vectors, when appropriate). These face data function names will appear after normal nodal data, but before all user-defined functions. These face data functions will only be available through the Function Selection panel when the Boundary Surface panel is up and face data exists for the current dataset.

After selecting which register you wish to load, you will be presented with all of the functions that may be loaded into the selected register. To select a new function, you may use the scroll bar on the right to select any variable from the list. Only functions available for the current dataset are listed here (apart from the intrinsic functions **FieldView** supplies).

Once you have selected the function you wish to load, press the calculate button to load it into the register.

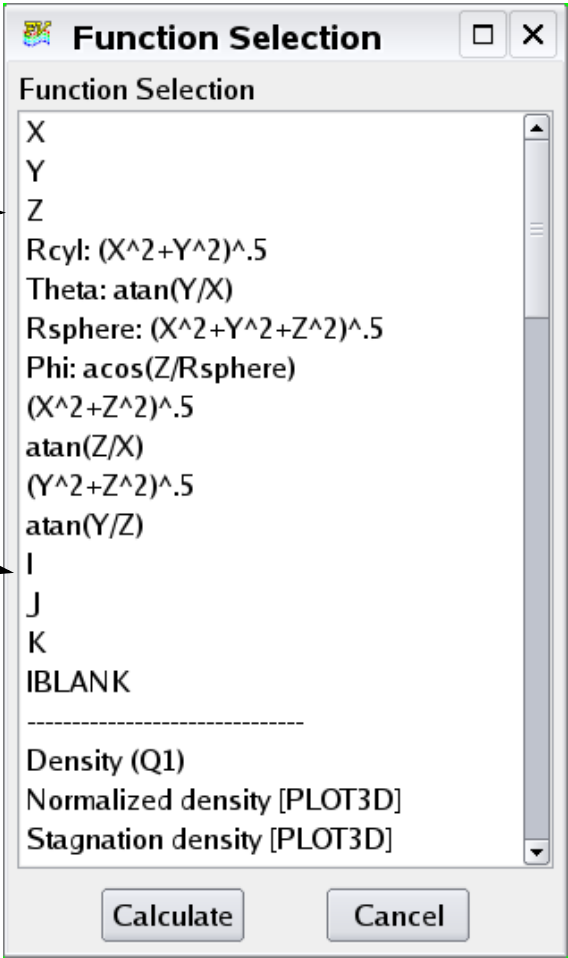


Figure 59 Function Selection Panel

Using the Functions Panel

Defaults after Reading a Data Input File

	Q File	Other Results File(s)
Iso-Surface	none	none
Scalar	Density (Q1)	F1 (1 st scalar)
Vector	Momentum (Q2-Q4)	V1 (1 st vector); if no vector, none.
Threshold	Density (Q1)	f1 (1 st scalar)

What are the 4 registers shown on the Functions panel?

FieldView has 4 basic function registers: Iso-Surface, Scalar, Vector and Threshold. These four registers are used to hold various scalar and vector quantities that have been read in from a file, are intrinsic, or have been calculated using the Function Formula Specification panel.

By using the Iso-Surface panel, a surface will be displayed that shows all areas in the volume where the currently loaded Iso-Surface function is equal to the value specified on the panel.

On any of the Visualization Panels, if the button marked Scalar is pressed, the current surface/rake will be colored by the scalar function that is currently loaded into the Scalar register.

If a surface is displayed as vectors those vectors will represent the vector variable that is currently loaded into the Vector register. Additionally, the current rake of streamlines is calculated based upon the function currently loaded into the Vector register.

If the Thresholding option on any of the Visualization panels is used, the current surface will be clipped to show only areas that are within the range specified on the panel, of the currently loaded Threshold function.

How do I change the function that is loaded into a register?

In order to change functions, press the appropriate register button you wish to change (e.g., if you wish to change the function that is loaded into the Scalar register, press the Scalar... button). When you press the button of your choice, you will be presented with the Function Selection panel (see **Figure 59**). This panel will list all of the functions available to be loaded into the register you have chosen (only vector functions for the Vector register, non-vector functions for the other registers). You may either double click on the correct function, or select it with a single mouse click, and press the Calculate button. This will cause the function to be calculated and loaded into the register you have selected. Once it is loaded, the minimum and maximum values for that function will be displayed.

Face Data and the Function Formula Specification Panel

Face data results *cannot* be used in user-defined formulas. This means that Face data function names will *not* appear in the Quantities list of functions available for use in a formula.

Using the Function Formula Specification Panel

Constants

The following constants are supported:

Alpha, FSMach, Re, Time - These are the four constants associated with *each grid* in a PLOT3D Q, NPARC/WIND or FV-UNS (Version 2.4 and higher) file. The constants will only be accessible when the data has been read in.

PI - the constant 3.1415... (π)

Lists the Name of the function that is being edited. It will be blank at other times.

The Constants section contains values that **FieldView** obtains from certain solver results files. For some solvers the number of constants available may be fewer or greater than the number shown here.

The Operations and Keys are the built-in functions supported by **FieldView**. Any formula that can be expressed in terms of these operations and solver variables can be created and saved.

When any of the Constants, Operations or Quantities are selected, the Formula created will be displayed in this area. This area may also be edited so that functions can be typed in directly.

Function Formula Specification

Name:

Constants

PI	UnitX	UnitY	UnitZ
Alpha	FSMach	Re	Time
Pinf	Tinf	R	Gamma

CFD Functions

Qcriterion

λ_2 criterion

Operations / Keys

VecX	VecY	VecZ	nrmlz
dot	cross	curl	mag
exp	ln	log	grad
sin	cos	tan	div
asin	acos	atan	atan2
sqrt	abs		
()	E	^
7	8	9	/
4	5	6	*
1	2	3	-
0	.	,	+

☐ Dataset Comparison

Quantities

(Y^2+Z^2)^.5
atan(Y/Z)
I
J
K
IBLANK

Density (Q1)
Normalized density [PLOT3D]
Stagnation density [PLOT3D]
Norm. stag. density [PLOT3D]
Log(norm. density) [PLOT3D]
Pressure [PLOT3D]
Norm. pressure [PLOT3D]
Stagnation pressure [PLOT3D]

Formula:
grad(grad(mag("velocity vectors")))

OK

Cancel

This toggle allows switching Dataset Comparison mode on and off.

Lambda2 (λ_2) criterion is based on the observation that, in regions where lambda2 is less than zero, rotation exceeds strain, and in conjunction with a pressure minimum, indicates the presence of a vortex.

The Q-Criterion function is based on the general form of the criterion proposed by Hunt in 1988. The function is generalized such that no assumption is made on compressibility.

Quantities lists all of the functions that are valid for the current dataset, including all of the geometric grid-based functions.

Figure 60 Function Formula Specification Panel

Operations/Keys

The operations allowed on the Function panel are described below:

Operator	Input	Output	Description
Qcriterion	1 vector	1 scalar	Built-in CFD Function Q-Criterion
$\lambda 2$ criterion	1 vector	1 scalar	Built-in CFD Function $\lambda 2$ -Criterion
VecX	1 vector	1 scalar	extract X component of a vector
VecY	1 vector	1 scalar	extract Y component of a vector
VecZ	1 vector	1 scalar	extract Z component of a vector
nrmlz	1 vector	1 vector	compute $V/\text{mag}(V)$
dot	2 vectors	1 scalar	dot product of 2 vectors
cross	2 vectors	1 vector	cross product of 2 vectors
curl	1 vector	1 vector	curl of a vector
mag	1 vector	1 scalar	magnitude of a vector
exp	1 scalar	1 scalar	take e to a power
log, ln	1 scalar	1 scalar	log or natural log of a scalar
log, ln	1 vector	1 vector	log or natural log of each component of a vector
grad	1 scalar	1 vector	gradient of a vector
sin, cos, tan	1 scalar	1 scalar	perform one of these trig operations on a scalar
div	1 vector	1 scalar	divergence of a vector
asin, acos, atan	1 vector	1 scalar	perform one of these trig operations on a scalar
atan2	2 scalar	1 scalar	output in radians
abs	1 scalar	1 scalar	absolute value
sqrt	1 scalar	1 scalar	square root
/, *	2 scalars	1 scalar	divide or multiply 2 scalars
-, +	2 scalars	1 scalar	subtract or add 2 scalars
-, +	1 vector 1 scalar	1 vector	multiply or divide a vector by a scalar
^	2 scalars	1 scalar	raise a scalar to a power
Unitx	none	1 vector	$\langle 1, 0, 0 \rangle$, unit vector along x-axis
Unity	none	1 vector	$\langle 0, 1, 0 \rangle$, unit vector along y-axis
Unitz	none	1 vector	$\langle 0, 0, 1 \rangle$, unit vector along z-axis

Figure 61 Function Operations

Frequently Asked Questions

How do I Create a new formula?

To create a new formula, first press the Create button on the Function Specification panel, which will bring up the Function Formula Specification panel. This panel is used to create new formulas based upon variables already in memory (but only for the current dataset), formulas previously defined, some pre-defined constants, and a set of pre-defined math functions shown in the table above.

You may use the mouse to press any of the buttons on the panel, or select any of the variables shown in the quantities section. For example, to define a new formula that is equal to the `pressure` divided by 2, you would perform the following operations:

- Select the variable `pressure` from the Quantities section
- Press the `/` button
- Press the number `2`

As you press these buttons, you should see the formula being created in the Formula section of the panel. You may also type into this area to create or modify the formula.



Naming Convention

Names of quantities must be enclosed in double-quotes (""); this is automatic if you select them with the mouse.

When you have finished entering the formula, press the OK button. At this point, you will be given the option to give your formula a unique name. You may either type in a name, or press the Use Formula button to have the name be the formula itself (in the example above, the name would be "`pressure`"/2). Note that Function Formula names are case insensitive (i.e., U is the same as u).

Creating a formula will not cause any calculation and will not load the formula into any of the function registers. In order to calculate the function and load it into a register, you simply use the Function Specification panel normally, and select the formula to load.

How can I Edit a previously created formula?

Once a formula has been created, it may be edited by pressing the Edit button on the Functions panel. When this button is pressed, you will be given a list of all previously created formulas to select from. After you pick which formula you wish to edit, the Function Formula Specification panel will be brought up, with the selected formula shown in the Formula section of the panel. At this point, you may change the formula by using the panel normally.

Note: You can edit formulas as long as they are not in use, or in use only by the current surface. If you try to edit a formula that is being used by a surface other than the current surface, you will be given an error message. If a formula is being used in a second formula, then you may only edit the first formula if the second formula is not in use or is in use only by the current surface. Editing the first formula will change the definition of the second formula as well.

*Can I save more formulas for use in another **FieldView** session?*

By using the Formula Restart option on the Restart Files menu, all of your formulas may be saved into a file to be reused in a later session of **FieldView**. In this way, you can build up a library of formulas that only need to be programmed once. For more information on Restart Files, please see [Chapter 5](#) of this **Reference Manual**.



Displaying Formulas

Only formulas that are valid for the current data will be shown in the Function Specification panel. For example, if you create a formula that uses the variable Pressure, and then read in data that does not contain Pressure, the formula will not be displayed in the Function Specification panel.

Possible Issues

Cannot edit this function. It is in use by 2D plots, or by surfaces other than the current surface.

A Formula may not be deleted or edited if it is “in-use” by any surface other than the current surface. This means that if the formula is loaded into any of the registers for any surface other than the current surface, you will be unable to edit or delete it.

Cannot edit this function. It is in use by one or more formulas, which in turn are used by 2D plots or by surfaces other than the current surface.

If you use a formula as part of the expression to define a second formula, then you will not be able to edit the first formula when the second formula is in use by any surface other than the current surface.

Differences between Datasets

The ability to see the differences between two or more datasets can help to determine the scope and magnitude of any changes between them. Dataset Comparison mode allows the creation of formulas that may reference quantities of more than one dataset. This permits the creation of dataset comparison formulas, such as the difference in pressure between two datasets.

The Dataset Comparison toggle on the Function Formula Specification panel enables this mode. All datasets in memory are tested against the current dataset for compatibility. Note that the *File.. Data Input.. Server Append* option must be checked ON (the default) to support Dataset Comparison if data is read with FieldView server(s).

WARNING: In order for a dataset to be compatible with the current dataset it must have the same number of nodes and the same grid count and grid dimensions. In order for the comparison to be valid the nodes in each dataset must be in the same order and have the same position in space, although **FieldView** does not check for this. **FieldView** makes no attempt to map results onto a different topology or connectivity. In short, the results of compatible datasets should be in a configuration where they could be appended to the geometry of the current dataset. For information on comparing datasets with different grids and results, see [Dataset Sampling](#) in [Chapter 14](#) of **Working with FieldView**.

This feature is intended to be useful for comparing different solver runs of the same case or for comparing different time steps of the same solver run, where various time steps may be appended one after another in **FieldView** memory. Comparing solutions from different solvers is beyond the current scope of this feature unless the solutions meet the criteria discussed above.

To identify quantities of specific datasets, quantity names are tagged by a dataset number and a colon (:) delimiter, as in 1:"pressure". This denotes the pressure scalar variable of dataset 1. A for-

mula to describe the pressure difference between datasets 1 and 2 may be written 1:"pressure"-2:"pressure".

Constants may also be prefixed with a dataset identifier. A formula for the C_p (coefficient of pressure) of an NPARC/WIND dataset:

$$(2 / (\text{Gamma} * \text{FSMach} * \text{FSMach})) * (\text{"Pressure [WIND]"} / \text{Pinf} - 1)$$

could be rewritten to yield the ΔC_p of datasets 1 and 2:

$$(2 / (1:\text{Gamma} * 1:\text{FSMach} * 1:\text{FSMach})) * (1:\text{"Pressure [WIND]"} / 1:\text{Pinf} - 1) - (2 / (2:\text{Gamma} * 2:\text{FSMach} * 2:\text{FSMach})) * (2:\text{"Pressure [WIND]"} / 2:\text{Pinf} - 1)$$

Note that constants are not quoted and that the dataset number prefix is before the quotes of quantity names. When selections are made from the Function Formula Specification list of quantities, the selections are pasted into the formula string buffer in this form.

Constants and quantities for comparable datasets are listed in the Quantities list of the Function Formula Specification panel prefixed by the dataset number while Dataset Comparison mode is enabled. The buttons for constants in the Function Formula Specification panel, except for π button, are disabled while Dataset Comparison mode is enabled. Thus, if datasets 1, 2, and 4 were matched as being comparable, and one of these datasets was the current dataset in **FieldView**, one might see the constant α for each of the datasets...

1:Alpha
2:Alpha
4:Alpha

... and a function velocity for each of the datasets...

1:velocity
2:velocity
4:velocity

Built-in quantities such as X , $\text{atan}(Y/Z)$, J and IBLANK will appear in the Quantities list tagged for each dataset, as in...

1:X
2:X
4:X

When Dataset Comparison mode is on, formulas containing no tagged quantities will appear in the Quantities list with a dataset prefix preceding the formula name. Such formulas will appear once for each dataset for which the formula can be calculated. Dataset Comparison formulas that contain tagged quantities will appear in the Quantities list only once. No dataset prefix will be added to the names of these formulas.



Formulas cannot be created for Surface based results.

If a PLOT3D dataset is dataset 1 and the equivalent NPARC/WIND dataset is dataset 2, and the geometry is identical, this could be a valid formula for `delta density`:

```
1:"Density (Q1)" - 2:"rho, density"
```

However, there may be differences in units and dimensionalization that may need to be accounted for. This may also be true of other data types.

Out of Range Handling

When a dataset is initially read into **FieldView**, all scalar and vector functions are checked for the presence of Inf and NaN values. If a scalar or vector function contains values that are not finite, a message is printed to the console window.

```
Variable "function-name" contains out of range values.
```

All scalar and vector function values that are not finite are skipped when computing the Min and Max values which are shown in the SCALAR COLORING section on the Colormap tab for all surfaces and rakes. If a scalar or vector function has no finite values at all, then the Min and Max values are both set to zero.

When a function is used for scalar coloring, values that are not finite are automatically mapped to the color **magenta**. When a function is used to define a vector surface, vectors that are not finite are not displayed.



If you see **magenta** colored areas like this on your visualization object, setting Threshold ON for that function will turn off the visibility of those areas.

Chapter 3

Region Files

3

Introduction

Regions are sub-volumes of datasets. Regions are defined using a simple ASCII text file having the *.fvreg extension. The Region file (FVREG) is used to group grids and define the type (Cartesian or cylindrical) and location of the coordinate system. In addition, the file is used to pass blade row information for rotating machinery and turbomachinery applications.



Important Note: The existence of FieldView Region Files (.fvreg) corresponding to data files for the Pratt *PW Common File* or *Acusolve [Direct Reader]* Data Input types will be ignored. Those two readers handle region files internally.

Region Features

With regions defined, you can:

- Create sub-volumes of your data for any data file format.
- Transform the origin of your dataset (useful if your symmetry plane does not lie at a zero axis, i.e. $x = 0$).
- Independently manipulate subvolumes of your data by transforming them and controlling their visibility.
- Use regions as subsetting tools to control the visibility for surfaces and rakes.
- Change from XYZ Coordinate Surfaces to RTX or RTZ Cylindrical Coordinate Surfaces (see note below).
- Create streamline rakes based on RTX or RTZ coordinates.
- Point probe in RTX or RTZ coordinates.
- Define a machine axis of either X or Z for rotating machinery cases.
- Define direction vectors for the machine axis and the zero theta plane.
- Define blade row parameters such as wheel speed, blade count and period to create periodic rotational copies.
- Switch between absolute and relative frames of references for the calculation of streamlines and the display of velocity vectors.

Regions form a separate hierarchy in **FieldView** with some attributes of a dataset and some attributes of surfaces. When regions are present for the *current* dataset, then there will always be a *current* region. This is similar to the concept of *current* dataset, rake or surface. The *current* region is that which will be transformed when Object: is Region. This added hierarchy level affects many areas and panels in **FieldView** which require separate controls and fields for information relevant to regions. In this section, those controls and fields will be detailed.

FieldView Panel	Change
Transform Controls	Like Dataset number, the current Region number is displayed in a separate field. In addition, +/- buttons allow you to easily change the <i>current</i> region. This section is grayed out if there are no regions defined for the <i>current</i> dataset. See Region Controls in Working with FieldView for more information.
Transform Controls, Object: pull-down menu	Like Datasets, Surfaces, etc., Regions can be transformed (translated, rotated and zoomed). The Object: pull-down contains Region in its list. If Object: is set to Region but there are no regions defined for the <i>current</i> dataset, attempting an object transform will produce an error. See Region Controls in Working with FieldView for more information.
Point Probe	This panel will display the region number and name if you probe on a grid belonging to a region. See Chapter 13 of Working with FieldView for more information.

DataGuide™: This feature can be used in conjunction with a region file (*.fvreg). However, once the **DataGuide™** files are created using a particular region file, they become associated with that *specific* FVREG file. If the associated FVREG file is changed, the **DataGuide™** files will be out of sync and will not be used. Important Note: If a dataset transform exists in the FVREG file, a **DataGuide™** results file (*.fvres) will not be created with the current implementation of regions. Changes in the region file can indicate that a change in the grid occurred. Therefore the old **DataGuide™** results file may be incorrectly associated with a new solver results file.

Region Subsetting

Regions add additional subsetting to **FieldView**. A surface (Coordinate or Iso-Surface) can be subsetted (restricted to) specific grids. That is, an Iso-Surface can be displayed on only grids 1-3 of a 9 grid geometry (for example). Region subsetting is available for Coordinate and Iso-Surfaces as well as Streamlines. Region subsetting for Computational surfaces is not required, since a Computational surface consists of only a single grid, and a grid either belongs to a region or does not. A grid cannot span more than one region like the other surface types mentioned above. In addition, *grid subsetting* is available for Streamlines (see [Chapter 6](#) of **Working with FieldView** for more information). If regions are used, then the Iso-Surface can be subsetted so that it will not be displayed over specified regions. This is performed with the surface's Subset Parameters panel. See [Subset Parameters](#)

Sub-panel in **Working with FieldView** for more information about the basic operation of the Subset Parameters sub-panel.

Turning *off* a region is equivalent to turning off all of the grids that belong to a region. If the current grid on the Subset Parameters panel belongs to the region chosen to be subsetted *off*, then a warning (in red) will appear on the panel informing you of this fact.

Once Region and Grid Subsetting has been applied, **FieldView** will determine the intersection of the region and grid subsetting. Each surface has *both* region and grid subsetting applied and the result to the visualization will be the *intersection* of your settings. That is, a grid is off if it belongs to a region that is off *or* the grid itself has been turned off. A grid is *on* only if the region it belongs to is also on.

Converting Data into Cylindrical Coordinates

The Region file can be used to permit visualization of XYZ data using an RTX/Z (Radius, Theta, X/Z) system in **FieldView**. This affects many of the panels and fields in **FieldView** and will be described in detail below (as well as mentioned in the specific panel chapters of **Working with FieldView**). Note that the input grid(s) values must *always* be Cartesian. **FieldView** facilitates display of this Cartesian data using RTX/Z coordinates upon read-in. **FieldView** will not recognize non-Cartesian grid data

A Cylindrical entry in the `fvreg` file affects a large number of different panels and operations. The areas affected are summarized in the following table. This information is also presented in notes in the specific chapters dealing with the features in question.

FieldView Panel	Change
Coordinate Surface	The Coordinate Surface panel will exhibit R, T, and X or Z surfaces and sliders instead of X, Y, and Z. See Chapter 9 of Working with FieldView for more information.
Vector Options Uniform Sampling	Sampling directions will be RTX or RTZ, instead of XYZ. See Vector Options Sub-panel in Working with FieldView for more information.
Streamlines	The usual XYZ seeding mode will be displayed and operate as RTX or RTZ seeding. In addition, the XYZ Auxiliary Seed Plane will instead be RTX or RTZ. See Chapter 6 of Working with FieldView for more information.
2D Plot	Three changes: i) The XYZ Curve plot types will be displayed and operate as RTX/RTZ Curve plots, ii) The XYZ choice on the Horizontal Axis/Plotting Direction will change to RTX/RTZ, and iii) the end points on the Edit Points panel will now be R, T, and X or Z. See Chapter 12 of Working with FieldView for more information.
Iso-Surface	The Cutting Plane type-in and edit fields will be RTX or RTZ instead of XYZ. See Chapter 5 of Working with FieldView for more information.

Point Probe	This panel will display point location in RTX or RTZ coordinates instead of the usual XYZ coordinates. See Chapter 13 of Working with FieldView for more information.
Import	Both Point Probe and 2D Plot import will be assumed to be in RTX or RTZ coordinates.
Export	2D Plot, Point Probe, Iso-Surface, Coordinate and Boundary Surface export will reflect RTX or RTZ coordinates in both labels and values. Note: Streamlines and Vortex Cores / Surface Flows will <i>not</i> be exported in RTX or RTZ coordinates.
Sweep Integration	The output file created by this form of integration will contain RTX or RTZ labels and values. See Integration Controls in Working with FieldView for more information.

Region Hierarchy

Regions are a distinct hierarchy in **FieldView** below Dataset but above grids and rakes. Since some surfaces span Regions (Iso-Surface, Boundary and Coordinate) and some do not (Computational surface), Regions and Surfaces can share a parallel level in the hierarchy. Also, the results of using Cylindrical Coordinates on the affected panels is mentioned throughout this **Reference Manual** and its companion, the **User's Guide**.

Region hierarchy and the effect of region transform is represented by the following two diagrams, where the various levels have been labeled for clarity. The hierarchy shown in this particular example is:

World

Dataset #1

Region #1

Region #2

Rake

Dataset #2

Surface

Region #1

Region #2

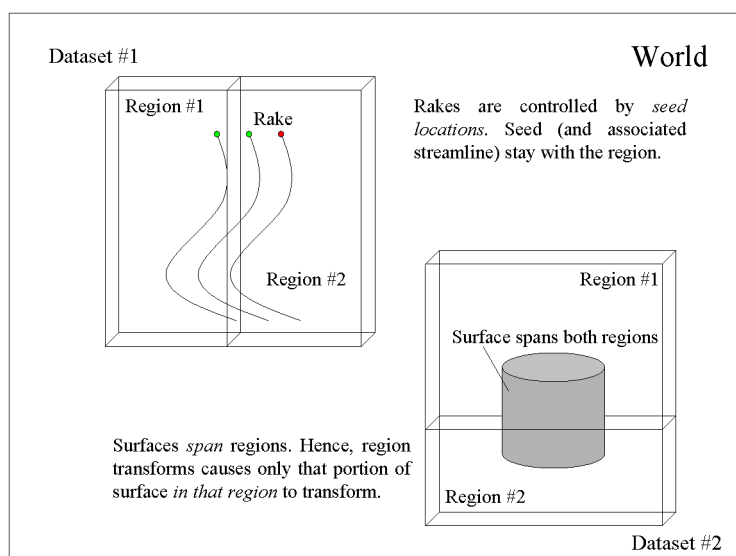


Figure 62 Region Hierarchy Schematic

Transforms affect surfaces and rakes differently. Rakes transform with the region their seeds belong to; surfaces will ‘split-up.’

- World
 - Dataset #1
 - Region #1
 - Region #2
 - Rake
 - Dataset #2
 - Surface
 - Region #1
 - Region #2

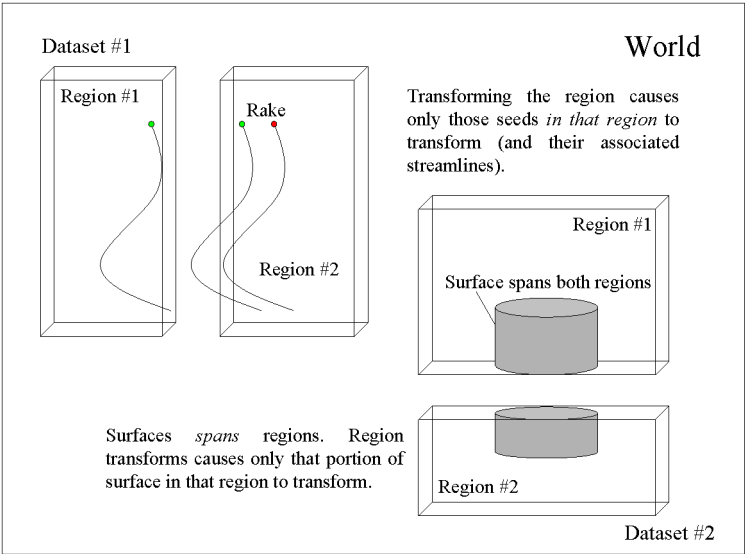


Figure 63 Transformed Region Hierarchy Schematic

Region Controls Panel

This section indicates the total number of regions for the current dataset as well as the current region. The region name as specified in the *.fvreg file is also shown. The Select... button allows you to select a specific region by name.

Here, the Visibility can be turned on and off.

The DUPLICATION section of the Region Controls panel allows you to set Region Mirroring or Rotational Duplication. (See next figures.)

Controls the Region Scale, Rotation, and Translation attributes, just as you can for the Dataset, using Dataset Controls Panel.

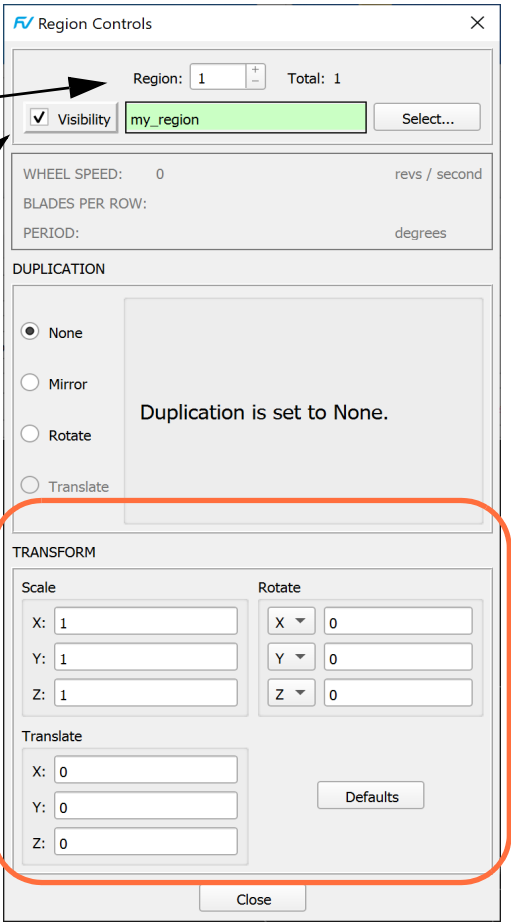


Figure 64 Region Controls Panel

This panel appears when the Select... button on the Region Controls panel is pressed. It allows you to select a specific region by name. The currently selected region (region [1] in this example) will be highlighted. Press the Close button once you have selected the desired region. Note: Only the *current* region can be translated or transformed.

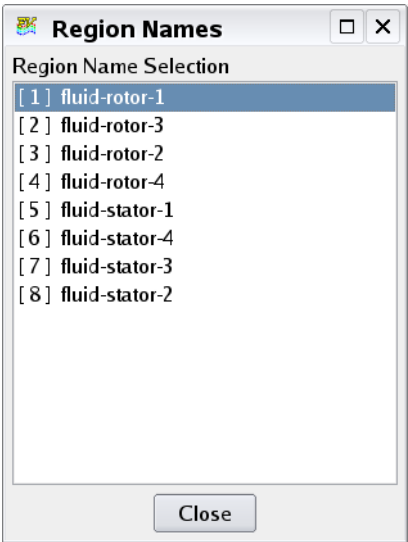


Figure 65 Region Names Panel

The DUPLICATION section of this panel changes to show Mirror Parameters, when *Mirror* is selected.

The section will change to show Rotation Parameters, when *Rotation* is selected.

These settings allow you to rotationally duplicate defined regions. Blade Row regions will use the value of the constant PERIOD from the FieldView Region (.fvreg) file in the Delta Sweep field. Non-Blade Row regions will use the default value of 360. Rotational duplication can be done in both the positive and negative angular direction.

WHEEL SPEED and BLADES PER ROW settings, if set in the Region File, will also be displayed on this panel, but do not provide any default panel settings.

Note: The *Translate* button on this panel applies only to Dataset Controls.

The figure displays two screenshots of the 'Region Controls' panel. The top screenshot shows the 'DUPLICATION' section with 'Mirror' selected, displaying 'X', 'Y', and 'Z' checkboxes. The bottom screenshot shows the 'DUPLICATION' section with 'Rotate' selected, displaying 'Copies [+]', 'Copies [-]', and 'Delta Sweep' fields, along with 'WHEEL SPEED' and 'BLADES PER ROW' settings.

Figure 66 Region Mirror and Rotate Parameters

The Mirror and Rotational duplication for Regions functions are similar to those for Datasets, shown on [“Dataset Controls Panel” on page 109](#) of the Working With FieldView PDF. However, unlike Dataset rotational duplication, Region Rotate does *not* count the current Region as part of the number of copies. Blade Row Regions will display the value of Period (from the `fvreg` file) in the Delta Sweep field. Hence, the panel above would result in the original Blade Row, two copies in the positive angular direction and one copy in the negative angular direction for a total of four. If more copies (Copies [+] + Copies [-] + Original) exist than are allowed by the 360 degree limit, **FieldView** will issue a pop-up warning and ignore, but not correct the value. If the Region Rotate Parameters panel is exited and then brought up again, the previous (good) value will be shown. The Delta Sweep field can be edited. A value larger than that set by the Period will result in gaps between the original and duplicated regions. A value smaller than the Period will result in regions that overlap.

Non-Blade Row Regions will show 360 in the Delta Sweep field. This default value should be set to the proper angular value for the region in question. Failure to do so will result in duplicate copies of the region 360 degrees apart, or at the same location, resulting in no visual difference but doubling the number of surfaces **FieldView** needs to draw, thus slowing performance. Since no period is provided

for a generic Region, more copies than necessary to fill 360 degrees can be input, with no warning messages.

For additional information, see [Periodic Streamlines](#) in **Working with FieldView**.

Region File Naming Convention

The additional hierarchy of regions (as well as dataset transform information) is communicated to **FieldView** through the use of the FVREG or **FieldView** Region file - (*.fvreg). This file is supported by all **FieldView** data readers, and uses the same file name as the grid file (or combined grid/results file), but with the additional .fvreg extension. Therefore, if you have a PLOT3D XYZ file called `turbo_x.bin`, then the associated FVREG file would be called `turbo_x.bin.fvreg`. If you were using a **FieldView** Unstructured (FV-UNS) combined grid/results file called `mydata.uns`, the region file would be named `mydata.uns.fvreg`.

Example:

<code>f16.xyz.bin</code>	Binary grid (XYZ) PLOT3D file
<code>f16.q.bin</code>	Binary Q (Results) file
<code>f16.xyz.bin.fvreg</code>	Region file



Note: **FieldView** will only look for all upper or all lower case suffix names. Mixed case suffixes will not be seen. That is, `f16.xyz.bin.Fvbnd` and `f16.xyz.bin.fvREG` are invalid Structured Boundary file names.

Transient FV-UNS and PLOT3D

If a region file is used for transient FV-UNS or PLOT3D files, then there are two valid file naming conventions. The easiest option is to have one global FVREG file for the entire transient sequence. This file would take the root name of the grid, or grid/results file *without* any embedded time step values. The other option is to use one FVREG file for *each* grid, or grid/results file, using the same naming convention. These two options are illustrated in the following example for the case of transient PLOT3D data:

Grid File	Separate FVREG Files	Global FVREG File
<code>duct_0010.g.bin</code>	<code>duct_0010.g.bin.fvreg</code>	<code>duct_.g.bin.fvreg</code>
<code>duct_0020.g.bin</code>	<code>duct_0020.g.bin.fvreg</code>	
<code>duct_0030.g.bin</code>	<code>duct_0030.g.bin.fvreg</code>	
...	...	
<code>duct_1080.g.bin</code>	<code>duct_1080.g.bin.fvreg</code>	



Important Note: Although the “one file per time step” naming convention is allowed, every FVREG file must have identical region, vector and blade row definitions.

This follows the same naming convention as Structured Boundary Files (for PLOT3D). See [Appendix H](#) of this **Reference Manual** for more information.

Region File Version 2 Format

Version 2 is the latest Region File format and its description file is ASCII. Its length and contents will depend on the amount of information you need to pass to **FieldView**. After the format description, several examples will show different forms of the FVREG file and its uses.



Note: If spaces exist before the beginning of a comment line, an error may result. Make sure that all spaces have been deleted so that the comment starts at beginning of the line.

Note: **Tecplot Inc.** continues to support Version 1 of the Region Files. Please see [“Region File Version 1 Format” on page 137](#) for more information.

Note: We expect that the CFD solver which you are using will be capable of writing a Region file with the correct settings automatically. At present, some commercial solvers do create Region Files, however, they may require some editing to capture the correct context for rotating machinery problems.

Below is a general example of a complete **FieldView** Region File:

```
! Comment lines preceded by '!'
! Begin with the Version Number
FVREG 2

DATASET_COORD_TYPE      CYLINDRICAL
MACHINE_AXIS            X
ROTATION_ORIENTATION    CW

! Basic Origin Definition [DEFAULT]
ORIGIN      0.000000 0.000000 0.000000

! AXIS DIRECTION VECTORS
! To point in X-direction, use:
MACHINE_AXIS_VECTOR      1.000000 0.000000 0.000000

! For a Zero Theta Plane at Y = 0, use:
ZERO_THETA_VECTOR        0.000000 1.000000 0.000000
```

```

! Refine display of R surfaces
FACET_COUNT      160

! If Velocity Components [rad], [tang], [axial] are needed,
! add the VELOCITIES section
!
VELOCITIES 1
velocity

! A simple region definition
REGION
  Inlet
  NUM_GRIDS 2
    2
    3

! To get Transformed, Relative Velocities, a
! BLADE_ROW Definition is needed
BLADE_ROW
  BLADES_PER_ROW      30
  WHEEL_SPEED          -125
  PERIOD               12
  NUM_REGIONS 1
  REGION
    fluid-rotor-1
  NUM_GRIDS 1
    1

```

A description of each section within the region file follows:

Region File Section	Description
FVREG 2	The Region File Format Version Number is set with the integer number 2. FieldView is backward compatible to recognize Version 1. Descriptions below apply to Region File Format Version 2.
DATASET_COORD_TYPE CARTESIAN/ CYLINDRICAL	There are two possible settings for this parameter: CARTESIAN or CYLINDRICAL. If CYLINDRICAL is chosen, then you will have the option to work directly with either RTX or RTZ coordinates. This parameter definition <i>must</i> be present within the Region File
MACHINE_AXIS X/Z	If the DATASET_COORD_TYPE is set to CYLINDRICAL then you <i>must</i> define this parameter. The choices are either X or Z, and this will determine whether you will work using RTX or RTZ coordinates within FieldView , respectively.

ROTATION_ORIENTATION CCW/CW	This parameter must be specified if the DATASET_COORD_TYPE is set to CYLINDRICAL. The options are for either CCW or CW and this defines the direction in which a Theta coordinate surface will be swept. This will also be incorporated into the formulas for the tangential velocity component.
ORIGIN xx yy zz	The ORIGIN is defined by the point with the Cartesian coordinates xx yy zz. For a non-zero set of values, a one-time transformation to all grid data will be done during the data read-in.
MACHINE_AXIS_VECTOR mx my mz	A MACHINE_AXIS_VECTOR can be defined to transform the dataset axis to match the axis of rotation for the model. As with the ORIGIN specification, this is a one-time transformation, applied to all grid data during the data read-in. The vector is defined by mx my mz. If this is not explicitly specified, then defaults are applied based on the MACHINE_AXIS choice.
ZERO_THETA_VECTOR tx ty tz	<p>The ZERO_THETA_VECTOR locates the plane where the Theta coordinate surface will be displayed. As with the ORIGIN specification, this is a one-time transformation, applied to all grid data during the data read-in. The vector is defined by tx ty tz. If this is not explicitly specified, then defaults are applied based on the MACHINE_AXIS choice.</p> <p>Note: The choice of the MACHINE_AXIS_VECTOR and the ZERO_THETA_VECTOR <i>must</i> form a perpendicular axis system. Otherwise, an error pop-up will inform you. If this happens, then the data will <i>not</i> be read. If this occurs as part of a restart read-in, the sequence will abort and only the data read in <i>prior</i> to the error will be in memory. Since the restart will abort during the Dataset restart, no surfaces will be drawn, etc., for any of the datasets.</p>



Note: If DATASET_COORD_TYPE is CARTESIAN, the default MACHINE_AXIS_VECTOR is (0,0,1) and the default ZERO_THETA_VECTOR is (1,0,0). If DATASET_COORD_TYPE is CYLINDRICAL, the defaults for these vectors and the formulas for R and THETA are shown in the table below. As shown in the table, they depend on the definition of MACHINE_AXIS and the ROTATION_ORIENTATION in the Region File.

	R Definition		Theta Definition	Machine Axis Vector (default)	Zero Theta Vector (default)
Z	$\sqrt{X^2+Y^2}$	CW	$\tan^{-1}(X/Y)$	(0,0,1)	(1,0,0)

	R Definition		Theta Definition	Machine Axis Vector (default)	Zero Theta Vector (default)
		CCW	$\tan^{-1}(Y/X)$	(0,0,1)	(0,1,0)
X	$\sqrt{Y^2+Z^2}$	CW	$\tan^{-1}(Y/Z)$	(1,0,0)	(0,1,0)
		CCW	$\tan^{-1}(Z/Y)$	(1,0,0)	(0,0,1)

FACET_COUNT <i>nfacet</i>	R Coordinate surfaces are drawn in FieldView using a default value of 120 facets per 360 degrees of span. For narrow blade passages, this default value may be too low to produce a smooth surface. The <code>FACET_COUNT</code> parameter, <i>nfacet</i> , lets you increase this to a maximum of 360 facets per 360 degrees.
VELOCITIES <i>nv</i>	<p>A region file does <i>not</i> have to contain a <code>VELOCITIES</code> section. If it does exist, it <i>must</i> be <i>after</i> the dataset section and <i>before</i> any <code>REGION</code> or <code>BLADE_ROW</code> section. It lets you specify <i>nv</i> vector functions (by name) which will be transformed based on the information for the <code>BLADE_ROW</code> section (see below). The following scalar functions will be automatically derived for a vector called <code>velocity</code>:</p> <pre> velocity [axial] velocity [radial] velocity [tangential] velocity [axial][rel] velocity [radial][rel] velocity [tangential][rel] </pre> <p>For a non-zero <code>WHEEL_SPEED</code> in the <code>BLADE_ROW</code> section, you will also have an additional vector available called <code>velocity [rel]</code> which is a transformed velocity for the absolute frame of reference.</p> <p>Important Note: All velocity vectors passed to FieldView <i>must</i> be in an absolute Cartesian reference frame to be correctly transformed to a relative reference frame.</p>

FIXED_VECTORS nfv	<p>A region file does <i>not</i> have to contain a FIXED_VECTORS section. If it does exist, it <i>must</i> be <i>after</i> the dataset and VELOCITIES sections, and <i>before</i> any REGION or BLADE_ROW section. This section allows you to specify which vector functions will <i>not</i> be transformed.</p> <p>Note: When there is a transform specified in the FVREG file, all vector variables will be rotated to match the new coordinate system, regardless of whether or not they are tagged as velocities. This is because vector variables will almost always need to be transformed, even if they are not a velocity (e.g. momentum).</p>
REGION region_name NUM_GRIDS ng	<p>A region file does <i>not</i> have to contain a REGION section. A region is a collection of one or more grids, ng. Not all grids in a dataset need belong to a region. However, no grid may belong to more than one region. You can have as many regions as there are grids within your dataset. Regions <i>group</i> grids together so that they may be treated as a single sub-volume unit for translations, rotations, duplication and detachment. Region names appear on the Region Controls panel.</p>
BLADE_ROW BLADES_PER_ROW bpr WHEEL_SPEED ws PERIOD np NUM_REGIONS nr REGION region_name NUM_GRIDS ng	<p>A blade row applies a set of properties to one or more regions. The BLADES_PER_ROW, bpr, is equal to the number of blades or passages that would result in a full 360 degree geometry, <i>not</i> the number of blades/passages that make up the grid file.</p> <p>The WHEEL_SPEED, ws, is the rotational velocity of the blade in units of revs/sec. A blade row can have a value of ws = 0, as would be the case for a stator row in a turbine.</p> <p>The PERIOD, np, is the 'sweep' angle used for rotational duplication. Hence, a period of 20 (degrees) will cause the Delta Sweep field on the Region Rotate Parameters to be 20, which will be used when copies are made.</p> <p>Important Note: All velocity vectors passed to FieldView <i>must</i> be in an absolute Cartesian reference frame to be correctly transformed to a relative reference frame.</p>

Omega Built-in Function

There is a built-in function, `omega`, available through the function calculator, which is similar to the `unitx`, `unity` and `unitz` unit direction vectors. This function can be used to show the wheel speed in units of [rad/s] for any given region if it is used as a scalar to color any surface within **FieldView**. Note that it has different units than the wheel speed, which are specified in [rev/s].

The definitions of the transformed cylindrical components of the velocity vectors will depend upon the definition of the MACHINE_AXIS and ROTATION_ORIENTATION. Additional definitions are needed for the *relative* cylindrical velocity components. These formulas are summarized in the following table:

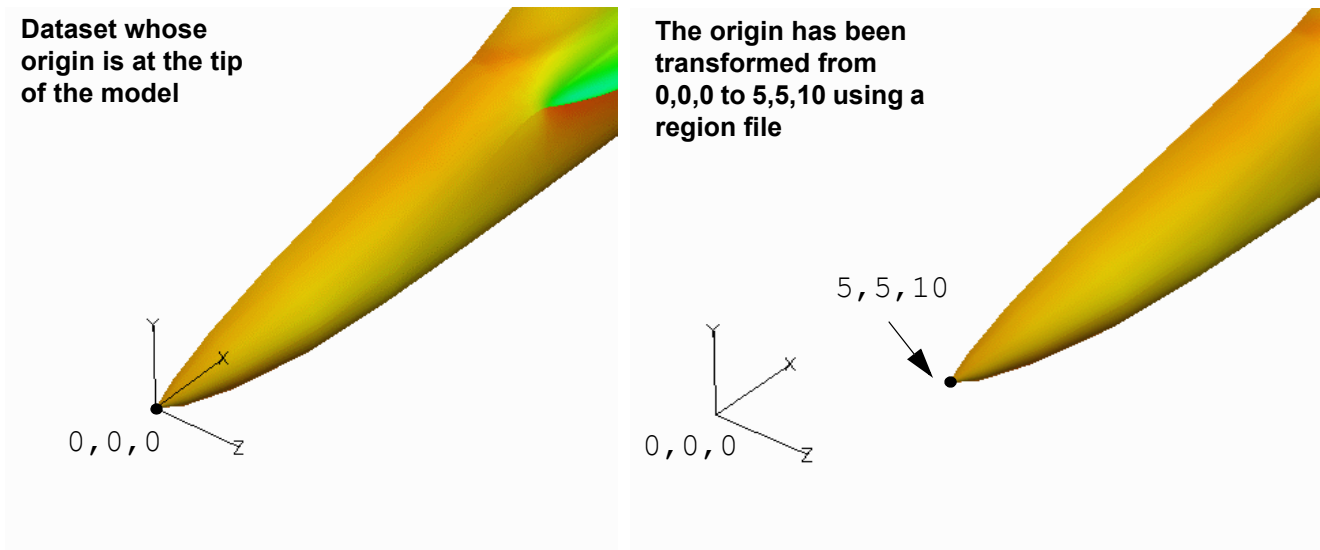
Z	Absolute	Vel[radial]	$V_x(x/R) + V_y(y/R)$
		Vel[tangential]	$V_y(x/R) - V_x(y/R)$
		Vel[axial]	V_z
	Relative		
X	Absolute	Vel[radial]	$V_y(y/R) + V_z(z/R)$
		Vel[tangential]	$V_y(z/R) - V_z(y/R)$
		Vel[axial]	V_x
	Relative		
	CCW	Vel[radial][rel]	$V_x(x/R) + V_y(y/R)$
		Vel[tangential][rel]	$V_y(x/R) - V_x(y/R) - R\Omega$
		Vel[axial][rel]	V_z
	CW	Vel[radial][rel]	$V_x(x/R) + V_y(y/R)$
		Vel[tangential][rel]	$-V_y(x/R) + V_x(y/R) - R\Omega$
		Vel[axial][rel]	V_z
	CCW	Vel[radial][rel]	$V_y(y/R) + V_z(z/R)$
		Vel[tangential][rel]	$V_y(z/R) - V_z(y/R) - R\Omega$
		Vel[axial][rel]	V_x
	CW	Vel[radial][rel]	$V_y(y/R) + V_z(z/R)$
		Vel[tangential][rel]	$-V_y(z/R) + V_z(y/R) - R\Omega$
		Vel[axial][rel]	V_x

Region File Examples

Basic Coordinate Transform Example

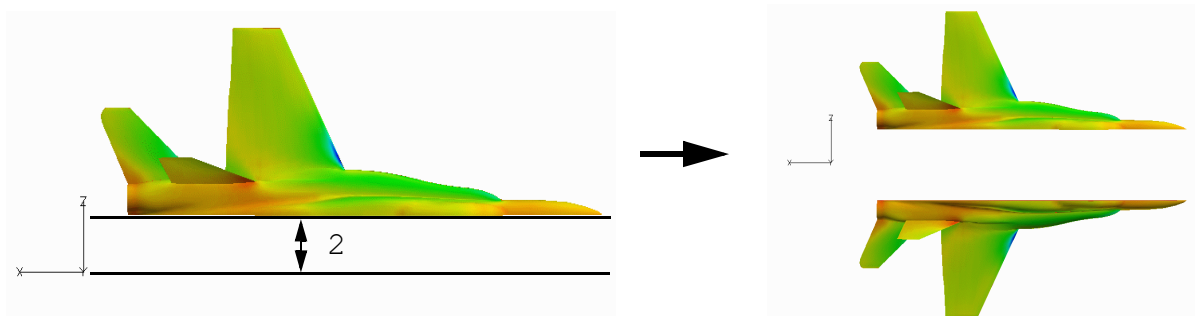
One of the basic uses of region files is to transform the dataset in cartesian space. The region file below transforms the origin of the dataset located at the tip of the model from 0,0,0 to 5,5,10.

```
FVREG 2
DATASET_COORD_TYPE CARTESIAN
ORIGIN 5 5 10
```

**Figure 67 Transforming Datasets in Cartesian Coordinates**

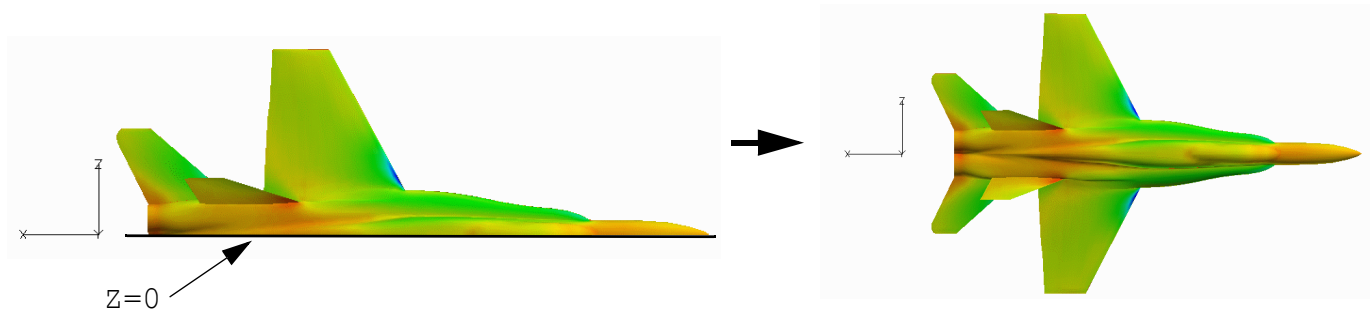
Mirroring

One of the benefits of using region files is to transform a model so that it can be properly mirrored. In some cases, the origin is not located on the symmetry plane and the model cannot be properly transformed as in **(Figure 68)** below. In this case the plane of symmetry is offset from by 2 meters.

**Figure 68 Mirror Offset Problem**

This problem can be corrected by using a region file that shifts the mirror plane by -2 along the z axis:

```
FVREG 2
DATASET_COORD_TYPE CARTESIAN
ORIGIN 0 0 -2
```

**Figure 69 Mirror Offset Correction**

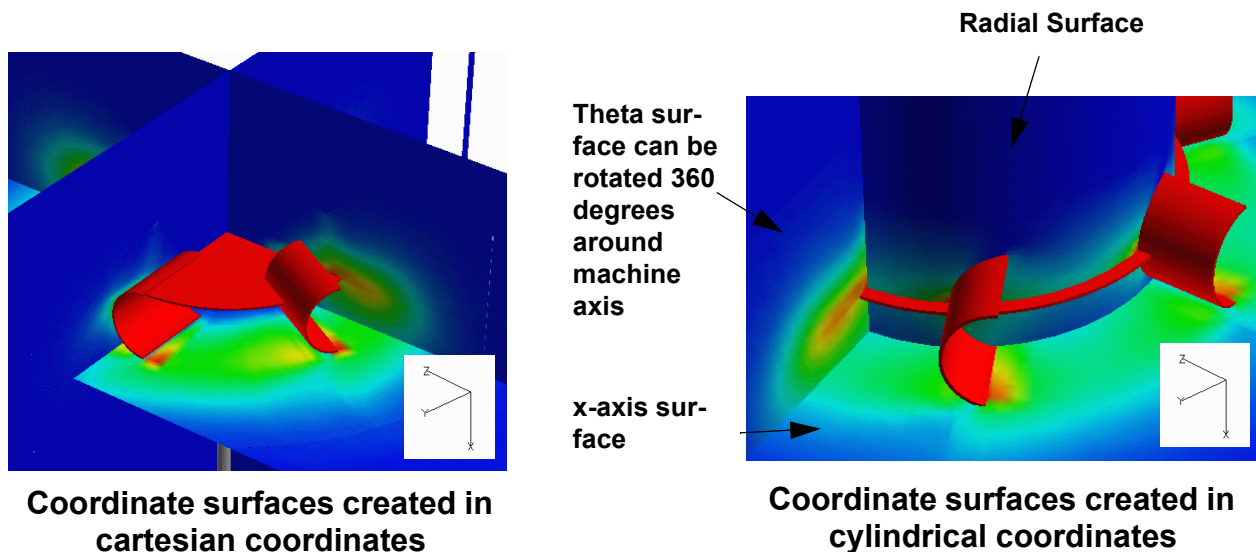
Cylindrical Coordinate Example

Region files allow the user to transform datasets from Cartesian to Cylindrical Coordinates. This Region File not only transforms the data into cylindrical coordinates, but orients the machine axis along the X-axis (which is the default definition in this case) and sets a rotation orientation in the clockwise direction (see **Figure 70** below):

```
FVREG 2
```

```

DATASET_COORD_TYPE      CYLINDRICAL
MACHINE_AXIS            X
ROTATION_ORIENTATION    CW
  
```

**Figure 70 Transforming Data into Cylindrical Coordinates**

Creating Smooth Radial Surfaces

In some cases the radial surface may have a jagged display as shown in **Figure 71**. To smooth the surface, the region file allows the user to increase the number of facets on the radial surface. **Figure 71** shows a total of 15 facets used in the display of the radial surface. By increasing the facet count to 80 with the below region file, the radial surface has become quite smooth as shown in **Figure 72**.

```
FVREG 2
```

```

DATASET_COORD_TYPE      CYLINDRICAL
MACHINE_AXIS             X
ROTATION_ORIENTATION     CW

```

```
FACET_COUNT      80
```

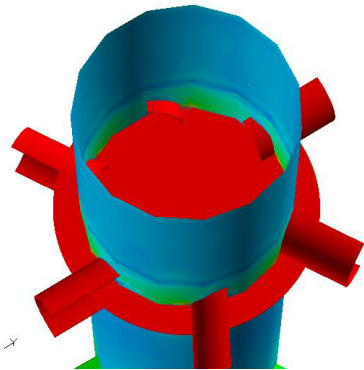


Figure 71 Jagged Radial Surface

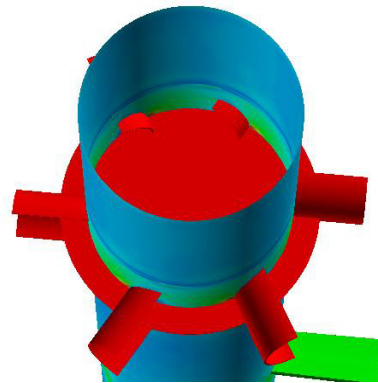


Figure 72 Smooth Radial Surface



Note: The facet count is optional. The default facet count is set to 40 which may generate smooth radial surfaces and therefore a facet section in your region file may not be needed.

Transforming Velocity Vectors

The cylindrical coordinates of velocity can be transformed and each component extracted by adding a velocity section. The below region file will transform the two velocity vectors named `phase-1_velocity` and `phase-2_velocity`.

```
FVREG 2
```

```

! Change to RTX Coordinates and
! used default axis definitions

```

```
DATASET_COORD_TYPE      CYLINDRICAL
```

```
MACHINE_AXIS      X
ROTATION_ORIENTATION  CW
```

```
!Transform velocity vectors to include
!axial, radial and tangential components
```

```
VELOCITIES 2
phase-1_velocity
phase-2_velocity
```

The region file will produce these velocity components that will be automatically loaded into the function panel:

```
phase-1_velocity [radial]
phase-1_velocity [tangential]
phase-1_velocity [axial]
phase-2_velocity [radial]
phase-2_velocity [tangential]
phase-2_velocity [axial]
```

These velocity components can be readily displayed on the model as can be seen from **Figure 73** and **Figure 74**.

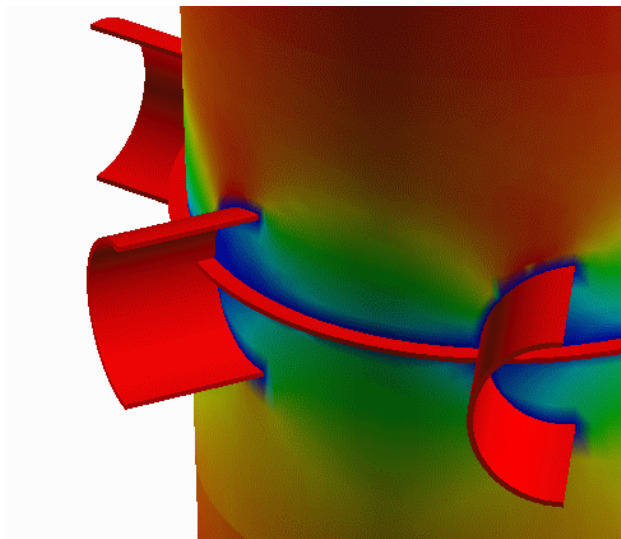


Figure 73 Tangential phase-1_velocity component on R surface

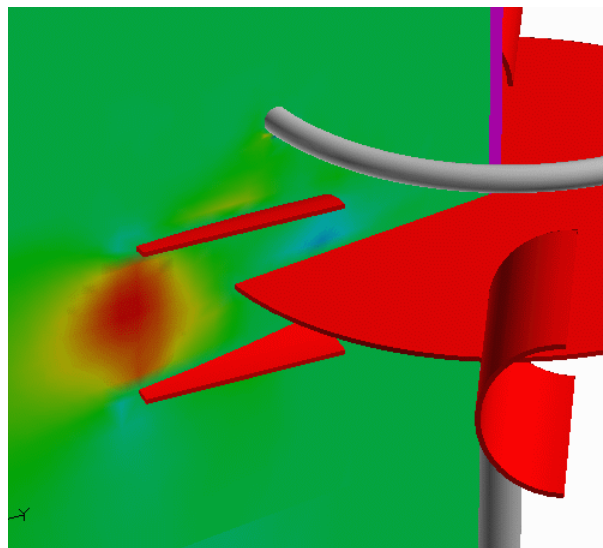


Figure 74 Radial phase-2_velocity component on theta surface

Adding Regions

Regions can be specified by adding a region section. Each region section includes the region name (specified by the user), the total number of grids grouped into the region and the grid numbers. If you do not know the grid numbers, they can be obtained by using the **FieldView** Point Probe Tool (see [Chapter 13](#) for more information on point probing). The below region file specifies 3 regions: the fluid region, the impeller with 6 blades and the impeller with 3 blades (see **Figure 75**).

```
FVREG 2

! Change to RTX Coordinates and
! used default axis definitions
DATASET_COORD_TYPE      CYLINDRICAL
MACHINE_AXIS            X
ROTATION_ORIENTATION    CW

! Transform velocity vectors to include
! axial, radial and tangential components
VELOCITIES 2
phase-1_velocity
phase-2_velocity

! Region Definitions
REGION
    fluid
NUM_GRIDS 1
    1

REGION
    6-blade_impeller
NUM_GRIDS 1
    2

REGION
    3-blade_impeller
NUM_GRIDS 1
    3
```

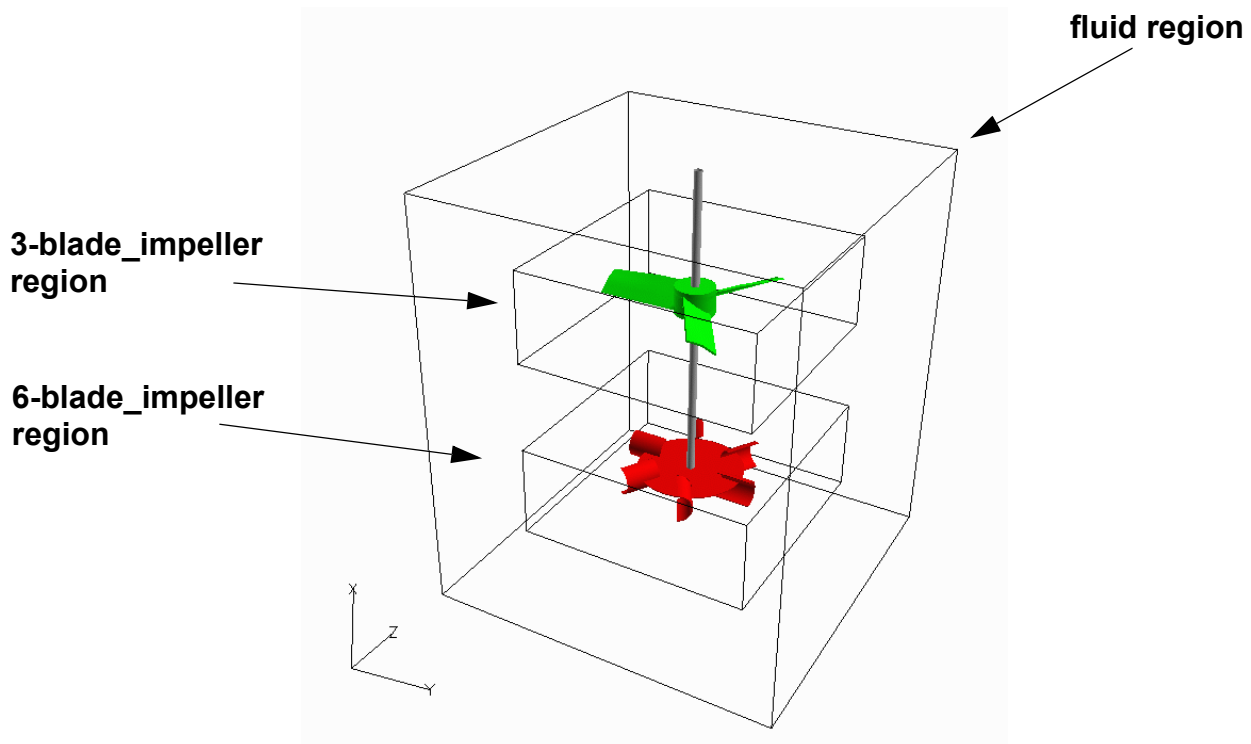


Figure 75 Region Definitions

Blade Row Example

In this section we will discuss how to create a region file for a blade row as shown in **Figure 76** below.

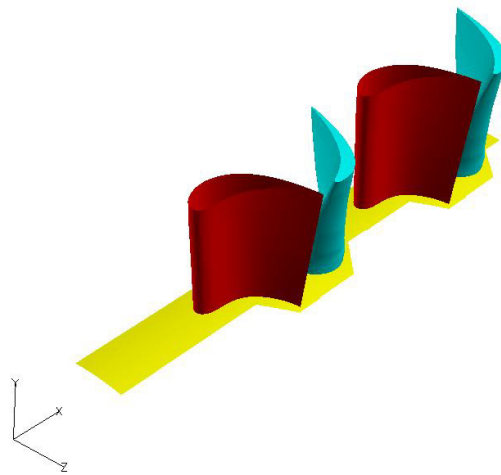


Figure 76 Blade Row

Defining Machine and Zero Theta Axis

The Machine Axis and the point where the Theta Axis begins at zero degrees can be defined in the region file. The below region file transforms the data into cylindrical coordinates, defines the Machine Axis as X (see **Figure 77** below), Rotation Orientation in the Counter-Clockwise direction (CCW), sets the Zero-Theta axis in the positive Y direction.

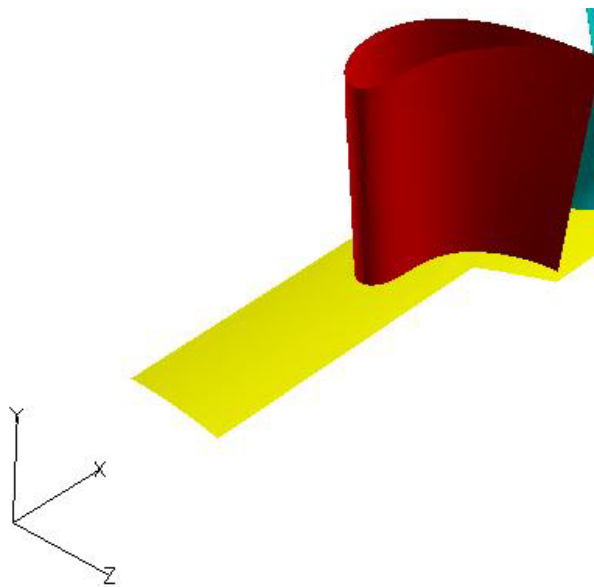
```
FVREG 2

DATASET_COORD_TYPE      CYLINDRICAL
MACHINE_AXIS            X
ROTATION_ORIENTATION    CCW

! Basic Origin Definition
ORIGIN                  0.000000 0.000000 0.000000

! AXIS DIRECTION VECTORS
! Machine Axis follows X-direction
MACHINE_AXIS_VECTOR     1.000000 0.000000 0.000000

! Rotation axis vector has no constraint
ZERO_THETA_VECTOR       0.000000 1.000000 0.000000
```



**Figure 77 Machine Axis
Defined in X-Direction**

Theta surface rotates around the machine axis and is defined so that at Theta = 0 it lies along the positive Y axis

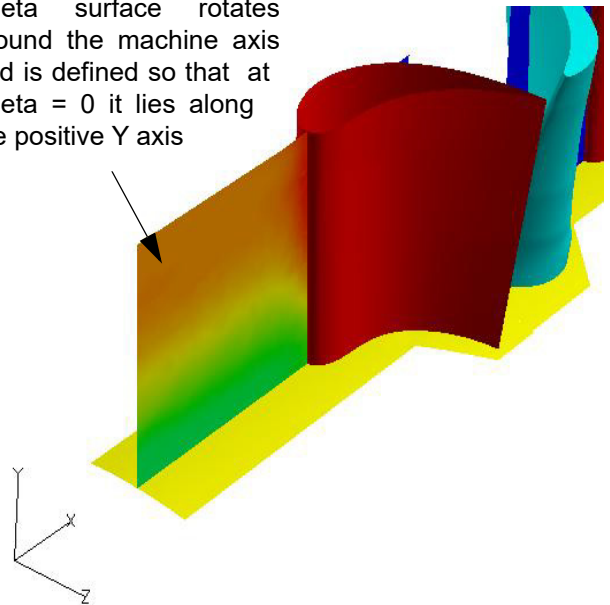


Figure 78 Zero Theta Surface

Adding Blade Rows

Each blade row can be defined by a region file. The information that you must provide are the number of blades per row, the wheel speed, period, the number of regions and the region definition. Before you create a region file, reviewing some basic definitions will be useful.

Definition of the Period

The period is the amount of degrees swept by a single blade. For example, the blade shown in **Figure 79** has a period of 12 degrees.

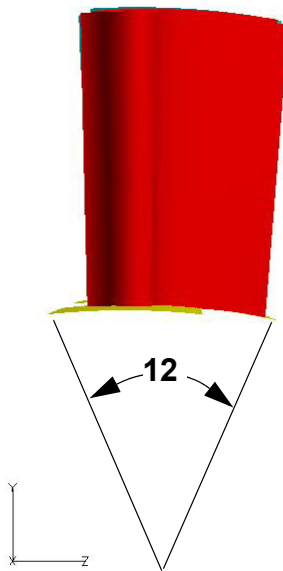


Figure 79 Period Definition

Definition of Blades Per Row

The blades per row is determined by how many blades determine a full 360 rotation. For example, if a blade has a period of 12 degrees the number of blades equals 360 divided by 12 resulting in 30 blades. To learn how to copy the blades, see [“Rotational Duplication of Regions” on page 136](#).

This row has 30 blades each with a period of 12 degrees rotated around the machine axis.

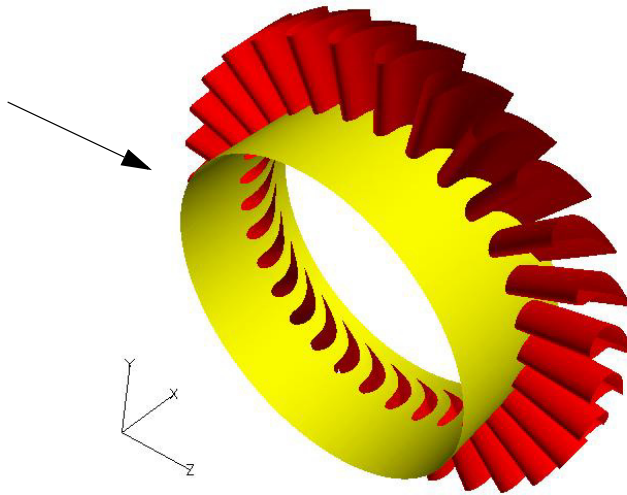


Figure 80 Blade Row Definition

Definition of Wheel Speed

The wheel speed is the rotational speed of the blade row and it can be set in either the clockwise or counter-clockwise direction. The units are always in revolutions per second (rev/s).

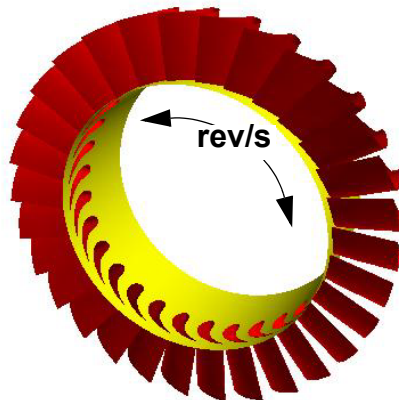


Figure 81 Wheel Speed

Creating the Region File

The below region file transforms the dataset into cylindrical coordinates, transforms the velocity and creates four blade row regions. It is important to note that you can have a blade row with a wheel speed of zero. See **Figure 82** for a picture of the model.

```
FVREG 2

DATASET_COORD_TYPE      CYLINDRICAL
MACHINE_AXIS            X
ROTATION_ORIENTATION    CCW

! Basic Origin Definition
ORIGIN                  0.000000 0.000000 0.000000

! AXIS DIRECTION VECTORS
! Machine Axis follows X-direction
MACHINE_AXIS_VECTOR     1.000000 0.000000 0.000000

! Rotation axis vector has no constraint
ZERO_THETA_VECTOR       0.000000 1.000000 0.000000

! Change display of R surfaces
FACET_COUNT             160

VELOCITIES 1
velocity
```

```

BLADE_ROW
  BLADES_PER_ROW 30
  WHEEL_SPEED -125
  PERIOD 12
  NUM_REGIONS 1
  REGION
    fluid-rotor-1
    NUM_GRIDS 1
    1

```

```

BLADE_ROW
  BLADES_PER_ROW 30
  WHEEL_SPEED -125
  PERIOD 12
  NUM_REGIONS 1
  REGION
    fluid-rotor-2
    NUM_GRIDS 1
    2

```

```

BLADE_ROW
  BLADES_PER_ROW 30
  WHEEL_SPEED 0
  PERIOD 12
  NUM_REGIONS 1
  REGION
    fluid-stator-2
    NUM_GRIDS 1
    3

```

```

BLADE_ROW
  BLADES_PER_ROW 30
  WHEEL_SPEED 0
  PERIOD 12
  NUM_REGIONS 1
  REGION
    fluid-stator-1
    NUM_GRIDS 1
    4

```

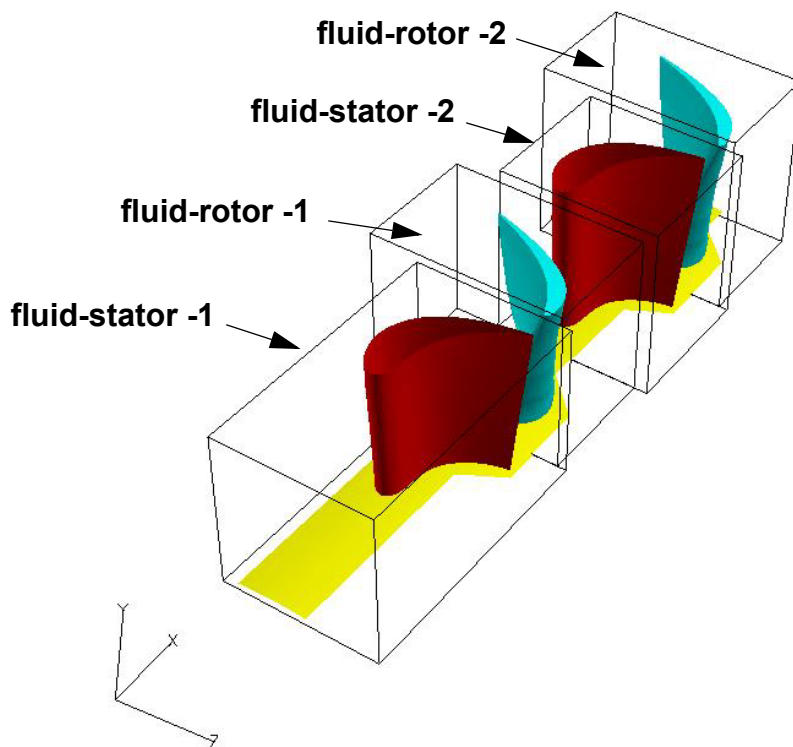


Figure 82 Turbo Region Example

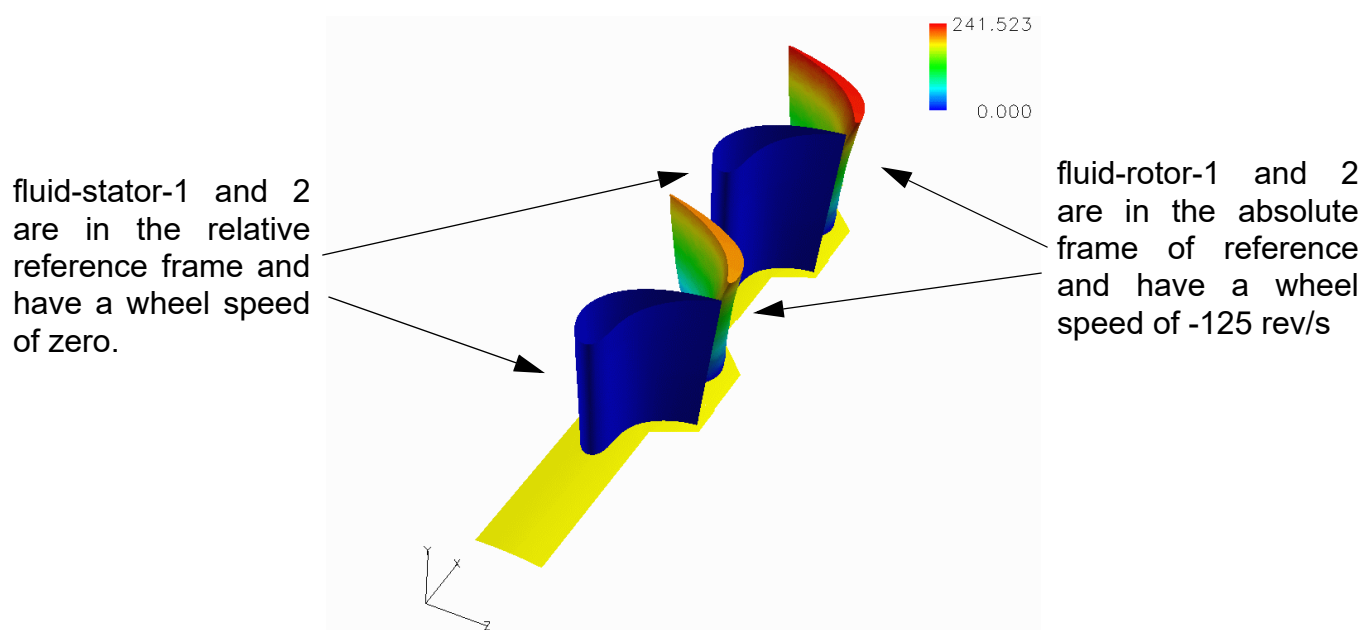


Figure 83 Transformed Velocity Magnitude on the Blades

Omega

Displaying omega as scalar on the boundary surfaces of the blades shows the wheel speed in radians/s. The scalar formula is created by simply typing in “omega” in the function creation panel.

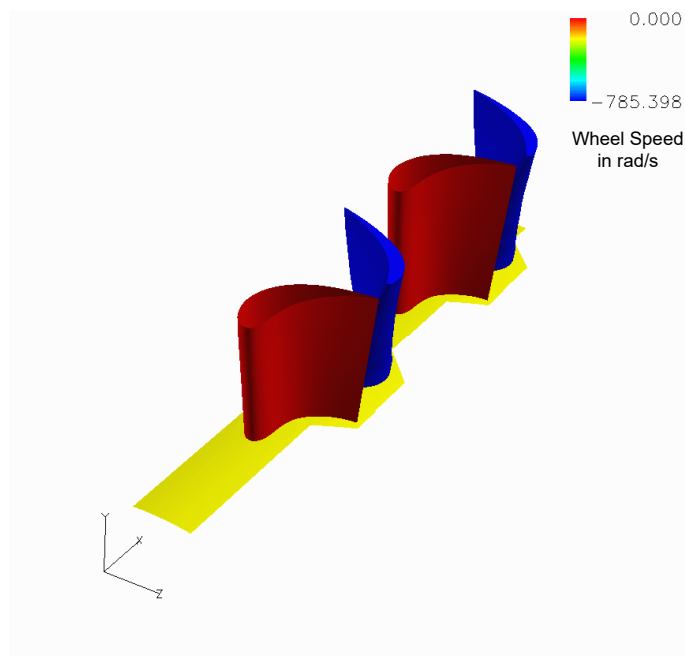
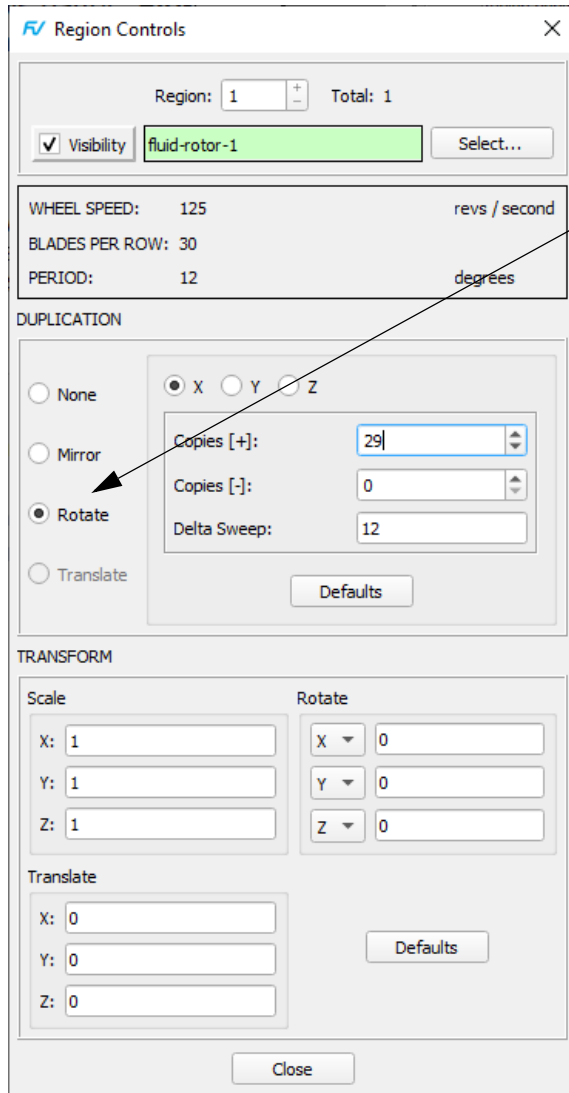


Figure 84 Displaying Omega

Rotational Duplication of Regions

Each region can be rotated independently of the other regions using the Region Controls panel. **Figure 85** below shows the settings in the region panel to rotate the fluid-stator-1 region 360 degrees around the Machine Axis.



Clicking on the **Rotation** button will fill the panel with settings for Region Rotate Parameters, and Delta Sweep. Blade Row regions will use the value of the constant PERIOD from the FieldView Region (.fvreg) file in the Delta Sweep field. Non-Blade Row regions will use the default value of 360. Rotational duplication can be done in both the positive and negative angular direction.

WHEEL SPEED and BLADES PER ROW settings, if set in the Region File, will also be displayed on this panel, but do not provide any default panel settings.

Twenty nine copies of the blade have been rotated around the machine axis resulting in a total of 30 blades for the row.



Figure 85 Copying Regions

Region File Version 1 Format

This format has been superseded by Version 2. Regions can be used to group grids for visualization purposes, whether it involves turbomachinery or not. Region formation provides you with better control over a multi-grid geometry much the same way as the Structured Boundary File for PLOT3D data allows better control of groups of computational surfaces.

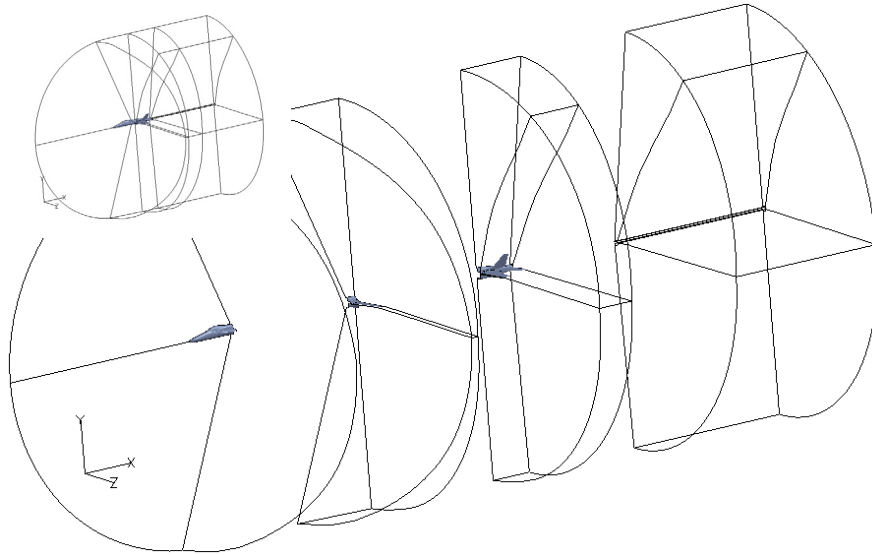


Figure 86 Region Example 2

In the above image, the F18 dataset (used in both the Basic Aerospace Tutorial, [Chapter 3](#) of the **User's Guide** and the `demo` script), is sectioned into four regions. The region file that defines the regions of **Figure 86** is shown on the following page.

FVREG 1	
DATASET_COORD_TYPE cartesian	Dataset
ORIGIN 0 0 0	
Z_AXIS 0 1 0	
THETA_AXIS 1 0 0	
REGION	Region #1
Forebody	
NUM_GRIDS 1	
1	
REGION	Region #2
Midbody	
NUM_GRIDS 2	
2	
3	
REGION	Region #3
Afterbody	
NUM_GRIDS 3	
4	
5	
6	
REGION	Region #4
Rear	
NUM_GRIDS 3	
7	
8	
9	

Region File Section	Description
FVREG 1	The Region File Format Version Number is set with the integer number 1. FieldView is backward compatible to recognize Version 1. Descriptions below apply to Region File Format Version 1.

DATASET_COORD_TYPE CARTESIAN/ CYLINDRICAL	There are two possible settings for this parameter: CARTESIAN or CYLINDRICAL. If CYLINDRICAL is chosen, then you will have the option to work directly with RTZ coordinates. This parameter definition <i>must</i> be present within the Region File
ORIGIN xx yy zz	The ORIGIN is defined by the point with the Cartesian coordinates xx yy zz. For a non-zero set of values, a one-time transformation to all grid data will be done during the data read-in.
Z_AXIS mx my mz	A Z_AXIS direction vector can transform the dataset axis to match the axis of rotation for the model. As with the ORIGIN specification, this is a one-time transformation, applied to all grid data during the data read-in. The vector is defined by mx my mz.
THETA_AXIS tx ty tz	The THETA_AXIS direction vector locates the plane where the Theta coordinate surface will be displayed. As with the origin specification, this is a one-time transformation, applied to all grid data during the data read-in. The vector is defined by tx ty tz. Note: The choice of the Z_AXIS and the THETA_AXIS <i>must</i> form a perpendicular axis system. Otherwise, an error pop-up will inform you. If this happens, then the data will <i>not</i> be read. If this occurs as part of a restart read-in, the sequence will abort and only the data read in <i>prior</i> to the error will be in memory. Since the restart will abort during the Dataset restart, no surfaces will be drawn, etc., for any of the datasets

When the DATASET_COORD_TYPE is set to CYLINDRICAL, the specification of the Z_AXIS and THETA_AXIS direction vectors fixes the definitions for R and Theta coordinates. It will always be the case that:

$$R = \sqrt{X^2 + Y^2} \quad \text{and} \\ \text{Theta} = \tan^{-1}(X/Y)$$

VELOCITIES nv	<p>A region file does <i>not</i> have to contain a VELOCITIES section. If it does exist, it <i>must</i> be <i>after</i> the dataset section and <i>before</i> any REGION or BLADE_ROW section. This section allows you to specify which velocity functions (by name) you wish to have available in cylindrical form, relative form, or both. For each (Cartesian) velocity function listed in this section, FieldView will create a derived cylindrical version of the vector quantity. This will have the same name as the (Cartesian) velocity, but will be given the [cyl] suffix. In addition, if a BLADE_ROW section (see below) with a non-zero value for WHEEL_SPEED exists, then FieldView will create a relative version of the Cartesian and cylindrical velocities. Given a vector, velocity, FieldView will make the following vector functions available:</p> <pre> velocity [cyl] velocity [rel] velocity [cyl][rel] </pre> <p>Important Note: All velocity vectors passed to FieldView <i>must</i> be in an absolute Cartesian reference frame to be correctly transformed to a relative reference frame.</p> <p>PLOT3D Note: All PLOT3D derived vector functions are computed from the Momentum [PLOT3D] vector (which is formed from the “Q” quantities Q2, Q3 and Q4). Hence, if Momentum [PLOT3D] is placed in the VELOCITIES section, all derived vector functions (Velocity [PLOT3D], Vorticity [PLOT3D], etc.) will be transformed, and [cyl] (and [rel] quantities if a BLADE_ROW section exists) versions made available as well. For a complete list of all PLOT3D functions, see Appendix A of this Reference Manual.</p>
---------------	---

Using Version 1 of the Region File Format, several vectors are available from within the function selection listing, typically titled: velocity, velocity [cyl], velocity [rel] and velocity [rel][cyl], where velocity was the name of the vector in the optional VELOCITY transform section of the region file. Because the [cyl] vectors had components based on R, Theta and Z, they could not be displayed correctly, since all vectors in **FieldView** are shown using a Cartesian (X/Y/Z) reference frame.

<p>FIXED_VECTORS nfv</p>	<p>A region file does <i>not</i> have to contain a FIXED_VECTORS section. If it does exist, it <i>must</i> be <i>after</i> the dataset and VELOCITIES sections, and <i>before</i> any REGION or BLADE_ROW section. This section allows you to specify which vector functions will <i>not</i> be transformed.</p> <p>Note: When there is a transform specified in the FVREG file, all vector variables will be rotated to match the new coordinate system, regardless of whether or not they are tagged as velocities. This is because vector variables will almost always need to be transformed, even if they are not a velocity (e.g. momentum).</p>
<p>REGION region_name NUM_GRIDS ng</p>	<p>A region file does <i>not</i> have to contain a REGION section. A region is a collection of one or more grids, ng. Not all grids in a dataset need belong to a region. However, no grid may belong to more than one region. You can have as many regions as there are grids within your dataset. Regions <i>group</i> grids together so that they may be treated as a single sub-volume unit for translations, rotations, duplication and detachment. Region names appear on the Region Controls panel.</p>
<p>BLADE_ROW BLADES_PER_ROW bpr WHEEL_SPEED ws PERIOD np NUM_REGIONS nr REGION region_name NUM_GRIDS ng</p>	<p>A blade row applies a set of properties to one or more regions. The BLADES_PER_ROW, bpr, is equal to the number of blades or passages that would result in a full 360 degree geometry, <i>not</i> the number of blades/passages that make up the grid file.</p> <p>The WHEEL_SPEED, ws, is the rotational velocity of the blade in units of revs/sec. A blade row can have a value of $w_s = 0$, as would be the case for a stator row in a turbine.</p> <p>The PERIOD, np, is the 'sweep' angle used for rotational duplication. Hence, a period of 20 (degrees) will cause the Delta Sweep field on the Region Rotate Parameters to be 20, which will be used when Copies are made.</p> <p>Important Note: All velocity vectors passed to FieldView <i>must</i> be in an absolute Cartesian reference frame to be correctly transformed to a relative reference frame.</p>

Chapter 4

FieldView Extension Language (FVX)

4

Introduction

FVX (the **FieldView** extension language) is a powerful language designed to help users extend and customize **FieldView**. Restart files and Scripts predate **FVX** as mechanisms to automate operations in **FieldView**. While powerful, there are some limitations that result from the fixed nature of the restart files and the “linear” (non-programmable) nature of the scripts. **FVX** will let you read datasets, create and manipulate surfaces, and perform complex quantitative post-processing tasks using a real programming language. A primary goal of **FVX** is automation of Quantitative Analysis such as computing figures of merit (e.g. efficiency). A secondary goal is to permit advanced visualization by mixing **FVX** with restart-based scripts. Rather than having redundant commands, all existing [Restart Files and Script Language](#) syntax from [Chapter 5](#) can be used within **FVX** by using the `fv_script()` command (see [page 246](#)).

FVX is based on Lua4. Additional functions and other details beyond what is documented here can be found in <https://www.lua.org/manual/4.0/manual.html>.

Some of the highlights of **FVX** are:

- Looping and Control Structures (`if`, `while`, etc.),
- User-defined variables and function calls,
- Built-in functions for string manipulation and mathematical functions,
- Ability to read and write files for parameters, reports, etc.,
- Functions to read CFD Datasets (PLOT3D and FV-UNS) and Query things like Grid topology, XYZ Min/Max, Function names, Boundary surface types, etc.,
- Create, Modify, Query and Delete Boundary Surfaces, Cutting Planes, Iso-surface, Streamlines, Coord and Computational Surfaces,

- Output the results of Integrate and Probe into variables,
- Functions to create basic GUI Panels and sliders (same look & feel for Linux/Windows),
- High quality, user-defined 2D Plots and bar charts that output via Postscript,
- A built-in Debugger that has been made to resemble the UNIX dbx tool.

FVX Syntax

FVX syntax is provided in this section, enabling you to write programs that can access a significant portion of **FieldView** functionality.

Chunks

In **FVX**, a 'chunk' is a segment of code that is a self-contained program. It can represent an independent program or can be part of another main program that is read-in and executed during the execution of the main program. It can range in length from a minimum of one line to a maximum whose limit is determined by the hardware environment.

Lexical Conventions

Variables in **FVX** can begin with any letter or the underscore '`_`' symbol. Variables may not begin with numbers. See [FieldView Limits](#) for maximum name length and number of variables. **FVX** is case-sensitive. Therefore, a variable '`temp`' is not the same as '`Temp`'. Comments can be placed in **FVX** programs. They begin with two dashes (`--`) and their scope extends to the rest of the line. The following words are reserved and may not be used as variables:

<code>and</code>	<code>break</code>	<code>do</code>	<code>else</code>	<code>elseif</code>
<code>end</code>	<code>for</code>	<code>function</code>	<code>if</code>	<code>in</code>
<code>local</code>	<code>nil</code>	<code>not</code>	<code>or</code>	<code>repeat</code>
<code>return</code>	<code>then</code>	<code>until</code>	<code>while</code>	

There is no need to declare variables in an **FVX** program. The scope of any variable is global unless explicitly declared as `local`. See [Variable Scope](#) for details.

Types

An **FVX** variable by itself does not have a type. The type of the variable depends on the assigned value. **FVX** values can have any one of five different types. They are `nil`, `number`, `string`, `function` and `table`.

`nil`

Any variable declared in the program which does not have a value assigned is automatically of type `nil`. In essence, it is a representation of the absence of a value for the relevant variable.

`number`

A variable of this type is a double-precision floating-point number.

`string`

A variable of type `string` consists of characters. Each character is represented by eight bits. Strings can be represented by single-quotes or double-quotes and the behavior is the same for both. One type may be used to enclose the other. The type of opening quote must be the same as the closing quote. It is better to be consistent with respect to which style is used. If the string has one type of quote as part of it, then the other type may be used to enclose the string. Strings may also have escape characters like those used in the C language.

`function`

Functions in **FVX** are represented by variables of type `function`. Because functions are represented by variables, they can be passed as arguments to other functions. They can also be returned as the result of other functions.

`table`

A variable of type `table` implements an associative array. Associative arrays are those that can be indexed not only with numbers but also with any other data type (with the exception of `nil`). The values may consist of all types in **FVX** (`nil`, `number`, `string`, `function`, `table`).

Working with Tables

Tables are created by *constructor* expressions which are represented by `{}`. The example below shows many methods of table creation and usage.

```
--create an empty table
iso_surface = {}

--create a table of string values with number indices
surface_types = {"wall", "inlet", "outlet"}

--create a table with a string index
-- this line creates the string variable "country"
country = "UK"
-- the table must be created here before being used below
capital = {}
capital[country] = "london"    -- use "country" as an index
print (capital["UK"])        -- prints "london"
```

The variable that represents a table only refers to it. When another variable is assigned this variable, both point to the same table, i.e., a copy is not made. For example,

```
-- assign the reference of this table to another variable
surface_types = {}
```

```

my_surface_types = {}
my_surface_types = surface_types
-- change the value for the second index
my_surface_types[2] = "manifold"
print (surface_types[2])           -- prints "manifold"

```

There is another method of accessing values in a table. This is of the form `table.index`. Note: This syntax may not be used for numerical indices. For example,

```

print (surface_types[3])           -- prints "outlet"
print (surface_types.3)            -- wrong syntax
print (capital["UK"])              -- prints "london"
print (capital.UK)                 -- prints "london"

```

As mentioned earlier, the indices of a table can be a combination of all types mixed together. For example,

```

test = {}
test[1] = 10
test[2] = 25
test["temp"] = 30
test.demo_func = function () print "Hello, World" end
print (test[1])                   -- prints "10"
print (test[2])                   -- prints "25"
print (test["temp"])              -- prints "30"
test.demo_func()                  -- prints "Hello, World"

```

In the above example, carefully note the syntax used to create and then call a function which is part of a table. See [Functions](#) for more details.

When creating a table with different types of indices by means of a constructor, the values using number indices should be separated from values using other types of indices by means of a semi-colon. Only a single instance of a semi-colon may appear in a table constructor. For example,

```

test = {"wall", "surface", "inlet"; length=10, width=8, height=3}
print(test[2])                    -- prints "surface"
print(test.width)                 -- prints "8"

```



Note: While tables can easily be used to construct one dimensional arrays, there is no native support for multi-dimensional arrays.

Variable Scope

As mentioned earlier, all variables in **FVX** automatically have a global scope. The only exception is when they are explicitly declared as `local`. The scope of a `local` variable is limited to the block in which it is declared. A 'block' in this case can be a control structure, the body of a function or a chunk of code. Variables with `local` scope have better performance. If there is a need to create a block within a segment of code, it can be done by starting it with the `do` statement and finishing it with the `end` statement.

Type Casting

FVX allows for automatic conversion of variables from type `number` to `string` and back again. Conversion from `number` to `string` does not cause loss of precision. This can be observed if this `string` is converted back to a type `number`. The disadvantage is that the appearance of `number` variables may not be desirable when printed. For the purpose of presentation, the `format` function described in a later section should be used.

Operators

FVX has operators standard to most computer languages: arithmetic, relational, logical and concatenation. These are covered below followed by information on operator precedence.

Arithmetic

Arithmetic operations are performed with variables of type `number`. Any `string` variable with a number in an arithmetic expression is automatically converted to `number` before the arithmetic operation is performed. The standard addition (+), subtraction (-), multiplication (*), division (/) and exponentiation (^) are available. Please note that values of type `number` are double-precision floating point values.

Relational

FVX supports the following relational operators: greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), equality (==) and inequality (~=). These operators return a value of type `nil` for false and a value of type other than `nil` for true. They can be applied to two variables where both variables are of type `number` or `string`. Variables of type `number` are compared as expected. Strings are compared based on alphabetical order, similar to other languages. However, string comparison is dependent on the locale set for the machine running **FieldView**. For instance, with the European Latin-1 locale, we have "acai"<"acorde". Other types can only be compared for equality and inequality. Tables and functions are compared by reference, i.e., two tables are considered equal only if the variables point to the very same table.

Logical

The **FVX** logical operators are `and`, `or` and `not`. These operators consider `nil` as false and all other types as true. The operator `and` returns `nil` if its first argument is of type `nil`. Otherwise, the second argument is returned. The operator `or` returns the

first argument if it is different from `nil`. Else, the second argument is returned. Here is one **FVX** idiom that uses logical operators:

```
x = y or z
```

where `x` is assigned the value `y` if `y` has a value of type other than `nil`. If the value of `y` is `nil` then `x` is assigned the value of `z`. If `z` is also `nil`, then `x` is assigned the value `nil`.

Concatenation

In **FVX**, two strings can be concatenated with the operator “`..`” (two successive dots). If one of the operands in the concatenation operation is a value of type `number`, it is automatically converted to type `string` before the concatenation operation. The operation does not affect the operands in any way because the output is a new `string`.

Example:

```
a = "Hello, "
b = " World"
c = a..b
print (c)           -- prints "Hello, World"
```

Operator Precedence

FVX operators are listed below in the order of decreasing precedence:

```
^ (exponentiation)
not  - (unary)
*    /
+    -
..
<>  <=  >=  ~=  ==
and  or
```

Within each line, the precedence is from left-to-right with the exception of the exponentiation operation whose precedence is from right-to-left. To avoid confusion, it is better to use parentheses to nest expressions. Note that for ‘not equal to’, the syntax used is ‘`~=`’ as opposed to ‘`!=`’ in other languages.

Statements

Assignment

FVX assignments set the value of variables. For example,

```
x = 1
```

assigns the variable `x` the value `1`. **FVX** has the unusual ability where it is possible to assign values to more than one variable in one assignment statement. For example,

```
x, y = a, b
```

In this example, `x` is assigned the value `a` and `y` is assigned the value `b`. When the number of variables on the left side is greater than the number of values on the right side, the extra variables are assigned the value `nil`. For example,

```
p, q, r, s = 1, 2
```

In this example, `p` and `q` are assigned the values `1` and `2` respectively. `r` and `s` are both assigned the value `nil`. Similarly, if the right side has more values than the number of variables on the left side, the extra values are ignored. For example,

```
m, n, o = 10, 11, 12, 13
```

Here, `m`, `n` and `o` are assigned the values `10`, `11` and `12` respectively. The value `13` is ignored.

Control Structures

FVX has the following type of control structures: `if`, `while`, `repeat`, and `for`. Control structures consider test values of `nil` as `false` and all others as `true`. In **FVX**, in test conditions, zero and non-zero values are evaluated to be `true`. Only `nil` values are considered as `false`.

`if`

An `if` control structure first tests for a condition and uses the `then` section for `true` and `else` section for `false`. The `if` control structure ends at the keyword `end`.

Example:

```
if x > 0 then
  y = 1
else
  y = 2
end
```

In this example, the `x>0` condition is tested and if this results in a value other than `nil` to be returned, the `then` segment of the structure is executed. Otherwise, the `else` segment is executed. More conditions can be tested after the initial `if` through the use of `elseif`.

Example:

```
if x > 20 then
  y = 1
elseif x > 10 then
  y = 2
elseif x > 0 then
```

```
    y = 3
else
    y = 4
end
```

`while`

The `while` control structure first tests the condition and then executes the body of the structure if the condition passes. The condition is re-tested always after the completion of the structure and it will repeat if it passes. The condition has to fail for the loop to end.

Example:

```
a = 1
b = 10
while a<b do
    c = a^2
    a = a + 1
end
```

As seen above, the delimiter of the `while` structure is the `end` statement.

`repeat`

The `repeat` control structure is similar to the `while` structure with the distinction that the test condition is at the end of the control structure. Therefore, the program will execute the body of the structure at least once.

Example:

```
local x = 1
repeat
    print (x)
    x = x + 1
until x>10
```

The `repeat` control structure uses the `until` statement to begin the test condition as seen in the above example.

`for`

The numeric `for` control structure uses a counter to progress through the loop for a set number of times. The increment parameter to the counter is optional. When not specified, the increment is set to 1. The control structure finishes at the statement `end`. A loop counter is set to an initial value for the first iteration of the loop. It is incremented for subsequent iterations of the loop. Execution of the loop stops when the loop counter exceeds the terminal value.

Example:

```
x = 0
for i = 1, 10, 2 do
```

```

    x = x + 1
end

```

At the end of the loop, the value of the variable `x` will be 5 because of the increment value 2. It is possible to set the increment to a negative integer.



Note: The variable used to represent the counter is automatically declared local. Therefore, when the loop exits, the value of this variable is not retrievable, i.e., the value of `i` after the end of the loop will be `nil`. If there is a need to preserve this value, it is necessary to assign it to another variable.

Within a loop, the value of the loop counter should never be changed. Changing this value inside the loop will cause unpredictable results. If there is a need to exit the loop, the `break` statement should be used.

Example:

```

y = 1
for j = 1, 20 do
    if j>10 then
        x = j
        break
    end
end

```

In the above example, the loop exits gracefully after it goes through eleven iterations. In addition, the value of the counter `j` is preserved in the variable `x`.

The table `for` loop can also walk through a table with numerical indices. By default, numerical indices begin with 1.

Example:

```

t = { "l", "m", "n", "o" }
for i = 1, getn(t) do
    print (t[i])
end

```

In the above example, the function `getn()` is used to retrieve the number of elements of table `t`.

The table `for` loop traverses through index-value pairs in the specified table, in lieu of using a counter. As each index-value pair is retrieved, appropriate action can be taken. The order in which the index-value pairs are retrieved is random. The number of times this control structure is traversed is equal to the number of indices in the table.

Example:

```
t = {a = 9,b = 6,c = 15,d = 27,tmp = 12}
for x, y in t do
    print (x, y)
end
```

Functions

Functions in **FVX** are similar to functions in other languages in that they perform a programmed set of tasks in a block and may return values. Input arguments may be provided. Variables declared as 'local' within functions have a scope restricted to the function, i.e., after the function returns, these variables and their values are destroyed:

Example:

```
function xyz (a, b, c)
    d = a + b + c
    return d
end

-- call the function
sum = xyz(3,2,3) -- the value stored in "sum" is 8
```

As in control structures, a function definition terminates with the `end` statement. In order for a function to be accessible in an **FVX** program, the function definition needs to be introduced to the program before it is called. When the above example is called with `a=1, b=2` and `c=3`, the function returns the value 6. Please note that `xyz` is a variable of type `function`, as mentioned previously. It is a direct reference to the function. It can be manipulated in ways similar to other types of variables. For example, it can be passed in as an argument to another function. It can also be a field in an **FVX** table.

Like any other value, a function may be part of a table. Functions may be defined as they are stored in a table. In this case, the function does not require a name. For example,

Example:

```
--define the function
t.func = function(x)
    return 5*x
end

--call the function
x = t.func(4) -- the value stored into variable "x" is 20.
```

A table can contain a function as well as data that the function can manipulate. In this case, the table may be considered an 'object' in the Object Oriented Programming sense.

Example:

```

--create a table
ambient = {
    soundspeed = 330,    -- [m/s]
    temperature = 273,  -- [deg. K]
    density = 1.0,      -- [kg/cm3]
    gamma = 1.4,        -- [-]
    pressure = 0.795,   -- [pascal]
    R = 0.008314        -- [(kg*m2) / (s2*kg-mol*pascal)]
}

--define function to calculate ambient speed of sound
ambient.soundcalc = function(ambient)
    ambient.soundspeed = sqrt(ambient.gamma*ambient.R*ambi-
ent.temperature)
end --end definition of function "soundcalc"

--define function to calculate ambient pressure
ambient.pressurecalc = function(ambient)
    ambient.pressure = ambient.density*ambient.R*ambient.tem-
perature
end -- end definition of function "pressurecalc"

```

In this example, the table holding ambient atmospheric properties may have its pressure and speed of sound values updated with functions that are also held in the table. In this sense, the table is a self-sufficient object which does not need outside functions to maintain its values. However, note that it was necessary to pass the table `ambient` as an argument since the function is not aware of other values in the table. **FVX** does have a *colon* notation which makes the act of passing a table to a function that is part of the table more convenient. The colon notation uses a hidden first argument called `self`. `self` is the table holding the function. The previous example is rewritten below to use the colon notation. Here, `self` refers to the table `ambient`:

Example:

```

--define function to calculate ambient speed of sound
function ambient:soundcalc()
    self.soundspeed = sqrt(self.gamma* self.R*self.tempera-
ture)
end

--define function to calculate ambient pressure
function ambient:pressurecalc()
    self.pressure = self.density * self.R * self.temperature
end

```

It is possible to write a function with a variable number of arguments. The special argument '...', when used as the last argument to a function definition, indicates a variable number of arguments. Inside the function, **FVX** creates a table `arg` which contains all the arguments that are passed in. The example below illustrates this feature:

Example:

```
function sum(...)
  local arg_count=1
  local total=0
  while arg_count<=arg.n do
    total = total + arg[arg_count]
    arg_count = arg_count + 1
  end
  return total
end

--call function
print( sum(1,2,3) ) -- returns 6
print( sum(0,3,8,6) )-- returns 17
```

A `return` statement is used to end a function or to return results from the function. Any function has an implicit `return` statement by default, so it is not required. In this case, no value is returned from the function. For syntactical reasons, a `return` statement can only appear as the last statement of a block or chunk or just before an `end` or `else` statement. Sometimes, there may be a need to have a return statement in the middle of a block. In such cases, the return statement can be used by wrapping it with an explicit `do` block.

Example:

```
function test(x)
  return          --incorrect syntax

  if x>2 then
    return        --incorrect syntax

    x = x + 1
  end

  --"return" is the last statement in this block
  do return end   --correct syntax

  ...
  ... --remainder of the function
  ...
end
```

Errors

When an error occurs during program execution, a pop-up window appears as an indication of the occurrence of the error along with some information on the error. When the popup window is dismissed, the **FVX** debugger is started. This provides access to the **FVX** environment for examination using debugger commands (see [FVX Debugger](#)). Upon completion of the examination, the program execution may not be continued. However, control may be returned to the **FieldView** GUI by entering the `quit` command.

General Function Library

Basic Functions

`dofile(filename)`

Receives a file name, opens the named file, and executes its contents as an **FVX** chunk, or as pre-compiled chunks. When called without arguments, `dofile()` executes the contents of the standard input (`stdin`). If there is any error executing the file, then `dofile()` returns `nil`. Otherwise, it returns the values returned by the chunk, or a non-`nil` value if the chunk returns no values. It issues an error when called with a non-string argument. Note: If an error occurs within a program executed by `dofile()`, then the program execution will resume at the line after this function.

An example would be using `dofile` to implement the **FVX** utility called `set_view` that allows for manipulation of the view parameters which will permit user defined orientation of a given dataset (see [FVX View Controls](#)).

Example:

```
dofile("set_view.fvx")
```

`dostring(string [, chunkname])`

Executes a given `string` as an **FVX** chunk. If there is any error executing the string, then `dostring` returns `nil`. Otherwise, it returns the values returned by the chunk, or a non-`nil` value if the chunk returns no values. The optional parameter `chunkname` is a reference used in error messages to point to the correct segment of code where the error occurred. This feature is most useful when the number of chunks used in a program is relatively high.

Example:

```
message = "print ("Hello, World")"  
dostring(message)
```

`getn(table)`

This function takes as input argument a `table`. It returns the number of entries in the `table`.

Example:

```
tmp_table = { a, b, c, d, e }
```

```
print (getn(tmp_table))  --prints "5"
```

```
tinset(table [, pos] , value)
```

Inserts element value at table position `pos`, shifting other elements to open space, if necessary. The default value for `pos` is `n+1`, where `n` is the result of `getn(table)`, so that a call `tinset(t,x)` inserts `x` at the end of table `t`. This function also sets or increments the field `n` of the table to `n+1`.

Example:

```
t = {3,6,12,8}
tinset(t,3,4) -- insert "4" at position 3
print t[3]    -- this statement outputs "4" instead of "12"
```

```
tonumber(string [, base])
```

This function has a variety of uses. Its main use is to convert strings to numbers. In addition, it may be used to check whether a string is a valid numeral. In this case, for invalid input, `nil` is returned. Finally, it can be used to convert numerals written in other bases. For this case, there is a second argument that specifies the base. The base may be any number between 2 and 36, inclusive. The letters correspond to the digits from 10 (A or a) to 35 (Z or z).

Example:

```
print(tonumber("10010.3"))  --prints "10010.3"
print(tonumber("kfjdas"))   --prints "nil"
print(tonumber({}))         --prints "nil"
print(tonumber("10010", 2)) --prints "18"
print(tonumber("1EF", 16))  --prints "495"
```

```
tostring(value)
```

This function accepts as input all data types and returns a string describing a value. The function `print()` automatically uses it to determine how to display the input value. For input arguments of type `string`, it returns the string itself. For an input argument of type `number`, it returns the number converted to a string in a reasonable format. For complete control of the conversion from type `number` to type `string`, such as the number of significant digits to be displayed, the function `format()` may be used. For the remaining types, this function returns the type name plus an internal identification (such as the memory address of the variable).

Example:

```
test = tostring(print)
--returns something like "function: 00482100"
test = tostring({})
--returns something like "table: 00486BA0"
test = tostring(_INPUT)
--returns something like "userdata(6): 00469A40"
test = tostring("FV")  --returns "FV"
```

```
tremove (table [, pos])
```

Removes from table the element at position `pos`, shifting other elements to close the space, if necessary. Returns the value of the removed element. The default value for `pos` is `n`, where `n` is the result of `getn(table)`, so that a call `tremove(t)` removes the last element of table `t`. This function also sets or decrements the field `n` of the table to `n-1`.

Example:

```
t = {3, 6, 4, 12, 8}
tremove(t, 2)
print t[2] -- this will output "4"
```

```
type(value)
```

This function returns a string describing the `type` of the input argument. Its results can be `function`, `nil`, `number`, `string`, `table` or `userdata`.

String Functions

```
format(formatstring, e1, e2, ...)
```

This function returns a formatted version of its variable number of arguments following the description given in its first argument (which must be a string). The `formatstring` follows the same rules as the `printf` family of standard C functions. The only differences are that the options/modifiers `*`, `l`, `L`, `n`, `p`, and `h` are not supported, and there is an extra option, `q`.

The `q` option formats a string in a form suitable to be safely read back by the **FVX** interpreter: The string is written between double quotes, and all double quotes, returns, and backslashes in the string are correctly escaped when written. For instance, the call with one argument

```
format("%q", "a string with \"quotes\" and \n new line")
```

will produce the string:

```
"a string with \"quotes\" and \
new line"
```

Conversions can be applied to the `n`-th argument in the argument list, rather than the next unused argument. In this case, the conversion character `%` is replaced by the sequence `%t$`, where `t` is a decimal digit in the range `[1,9]`, giving the position of the argument in the argument list. For instance, the call

```
format("%2$d -> %1$03d", 1, 34)
```

will result in

```
"34 -> 001"
```

The first argument 1 is affected by the format string `%1$03d`. It is forced to a string length of three because of 3 in the format string and is padded with two zeros because of 0 which specifies that the string must be padded with zeros if the resulting length is less than three. The `d` indicates that it is an integer with signed decimal notation. The second argument is affected by the format string `%2$d`. The 2 indicates the number of the argument that will be affected and the `d` indicates that it is output as an integer with signed decimal notation. The same argument can be affected by more than one format string. For example

```
format("%2$d -> %1$03d -> %2$d", 1, 34)
```

will result in

```
"34 -> 001 -> 34"
```

The table below provides the format options that may be used in format strings.

Character	Arg. Type	Converted To
d, i	int	signed decimal notation.
o	int	unsigned octal notation (with leading zero).
x, X	int	unsigned hexadecimal notation (without a leading 0x or 0X), using abcdef for 0x or ABCDEF for 0X.
u	int	unsigned decimal notation.
c	int	single character, after conversion to unsigned char.
s	char *	characters from the string are printed until a '\0' is reached or until the number of characters indicated by the precision have been printed.
f	double	decimal notation of the form <code>[-]mmm.ddd</code> , where the number of d's is specified by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
e, E	double	decimal notation of the form <code>[-]m.ddddde[+ or -]xx</code> or <code>[-]m.dddddE[+ or -]xx</code> , where the number of d's is specified by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
g, G	double	<code>%e</code> or <code>%E</code> is used if the exponent is less than -4 or greater than or equal to the precision; otherwise <code>%f</code> is used. Trailing zeros and a trailing decimal point are not printed.

```
strfind(s, pattern [, init [, plain]])
```

Looks for the first match of pattern in s. If it finds one, then `strfind` returns the indices of s where this occurrence starts and ends; otherwise, it returns `nil`. If the pattern specifies captures, the captured strings are returned as extra results. A third, optional numerical argument `init` specifies where to start the search; its default value is 1, and may be negative. A value of 1 as a fourth, optional argument `plain` turns off the pattern matching facilities, so the function does a plain "find substring" operation, with no characters in pattern being considered "magic".



Note: If `plain` is given, then `init` must be given as well. Also, two backslashes (`\\`) will be treated as one character, i.e., one backslash.

Example:

```
absolute_path = "C:\\Program Files\\FieldView\\FieldView 2023"
start,finish = strfind(absolute_path, "FieldView\\")
--value of "start" is 18, value of "finish" is 27
print (start, finish)
```

Mathematical Functions

The library provides the following functions:

```
abs acos asin atan atan2 ceil cos deg exp floor log log10
max min mod rad sin sqrt tan frexp ldexp random randomseed
```

plus a global variable `PI`. Most of them are only interfaces to the homonymous functions in the ANSI C library. Note: For the trigonometric functions, all angles are expressed in degrees, not radians. The functions `deg` and `rad` can be used to convert between radians and degrees. The function `max` returns the maximum value of its numeric arguments. Similarly, `min` returns the minimum. Both can be used with 1 or more arguments. The functions `random` and `randomseed` are interfaces to the simple random generator functions `rand` and `srand`, provided by ANSI C. (No guarantees can be given for their statistical properties.) The function `random`, when called without arguments, returns a pseudo-random real number in the range `[0,1]`. When called with a number `n`, `random` returns a pseudo-random integer in the range `[1,n]`. When called with two arguments, `l` and `u`, `random` returns a pseudo-random integer in the range `[l,u]`.



Note for FORTRAN and C programmers: When improper arguments are passed to mathematical functions in **FVX**, the operation will not cause an error exit. Instead, a value is returned from the function that corresponds to the IEEE specification for floating point numbers. Each platform may return slightly different representations of `INFINITY` or `NAN`. On Windows for example, `sqrt(-1)` returns `-nan(ind)`. Thus, the improper use of the `sqrt` function may not be detected until the result is used in another operation, such as indexing a table.

Standard I/O Functions

```
openfile(filename, mode)
```

This function opens a file in the mode specified in the string `mode`. It returns a new `file_handle`, or, in case of errors, `nil` plus a string describing the error. This function

does not modify the predefined **FVX** environment variables `_INPUT` or `_OUTPUT`. The `mode` string can be any of the following:

```

"r"      - read mode;
"w"      - write mode;
"a"      - append mode;
"r+"     - update mode, all previous data is preserved;
"w+"     - update mode, all previous data is erased;
"a+"     - append update mode, previous data is preserved,
writing is only allowed at the end of file.

```

The `mode` string is exactly what is used in the standard C function `fopen()`. Note: This string may also have a "b" at the end, which is needed in the Microsoft Windows platform to open the file in binary mode.

Example:

```

--open a new file "test.txt" in write mode
file_handle = openfile("test.txt", "w")

--export a coordinate surface and read the data from the file
fv_script("EXPORT COORD coord_surface")
data_handle = openfile("coord_surface", "r")

```

```
closefile(handle)
```

This function closes the given file. It does not modify either `_INPUT` or `_OUTPUT`. Afterwards, the handle is no longer valid.

Example:

```

--close the file "test.txt"
closefile(file_handle)

```

```
readfrom([filename])
```

This function may be called in two ways. When called with a file name, it opens the named file, sets its handle as the value of `_INPUT`, and returns this value. It does not close the current input file. When called without parameters, it closes the `_INPUT` file, and restores `stdin` as the value of `_INPUT`. If this function fails, it returns `nil`, plus a string describing the error.

If `filename` starts with a `|`, then a piped input is opened, via function `popen()`. Not all systems implement pipes. Moreover, the number of files that can be open at the same time is usually limited and depends on the system.

Example:

```

--set file_handle as the default handle _INPUT
file_handle = readfrom("test.txt")

```

```
--close the current _INPUT file and set stdin as the new
--_INPUT
readfrom()
```

```
remove(filename)
```

Deletes the file with the given name. If this function fails, it returns `nil`, plus a string describing the error.

Example:

```
--delete the file "test.txt"
remove("test.txt")
```

```
rename(name1, name2)
```

Renames file named `name1` to `name2`. If this function fails, it returns `nil`, plus a string describing the error.

Example:

```
--rename the file "test1.txt" to "test2.txt"
rename("test1.txt", "test2.txt")
```

```
read ([filehandle,] format1, ...)
```

Reads file `_INPUT`, or `filehandle` if this argument is given, according to the given formats, which specify what to read. For each format, the function returns a string (or a number) with the characters read, or `nil` if it cannot read data with the specified format. When called without formats, it uses a default format that reads the next line (see below).

The available formats are:

"*n"- reads a number; this is the only format that returns a number instead of a string.

"*l"- reads the next line (skipping the end of line), or `nil` on end of file. This is the default format.

"*a"- reads the whole file, starting at the current position. On end of file, it returns the empty string.

"*w"- reads the next word (maximal sequence of non-white-space characters), skipping spaces if necessary, or `nil` on end of file.

"number"- reads a string with up to that number of characters, or `nil` on end of file.

Example:

```
--read from current open file (from handle _INPUT) one line
--for each time read() is called
read("*l")

--read the entire contents of file "test3.txt" (with handle
--file_handle) with one call to function read()
read(file_handle, "*a")
```

```
write ([filehandle, ] value1, ...)
```

Writes the value of each of its arguments to file `_OUTPUT`, or to `filehandle` if this argument is given. The arguments must be strings or numbers. To write other values, use `tostring` or `format` before `write`. The character strings `"\r"` (carriage return) or `"\n"` (newline) can be used for line termination. Both may be needed for MS-Windows style line termination. If this function fails, it returns `nil`, plus a string describing the error.

Example:

```
-- write arguments to current file handle
x = "Hello"
y = ", "
z = "World"
write(x, y, z)

--write arguments to file "test5.txt" with handle
--file_handle_5
write(file_handle_5, x, y, z)
```

Example:

```
--export a coordinate surface and read the data from the file
fv_script("EXPORT COORD coord_surface")
data_handle = openfile("coord_surface", "r")

x[i] = read(data_handle, "*n")
-- reads number from first column
y[i] = read(data_handle, "*n")
-- reads number from second column
z[i] = read(data_handle, "*n")
s[i] = read(data_handle, "*n") -- reads the scalar value
line = read(data_handle, "*l") -- goes to the next line
```

```
writeto([filename])
```

This function may be called in two ways. When called with a file name, it opens the named file, sets its handle as the value of `_OUTPUT`, and returns this value. It does not close the current output file. Note that, if the file already exists, then it will be completely erased with this operation. When called without parameters, this function closes the `_OUTPUT` file, and restores `stdout` as the value of `_OUTPUT`. If this function fails, it returns `nil`, plus a string describing the error.

If `filename` starts with a `|`, then a piped input is opened, via function `popen()`. Not all systems implement pipes. Moreover, the number of files that can be open at the same time is usually limited and depends on the system.

Example:

```
--open the file "test.txt" and return its handle.
```

```
--test_handle becomes the same as _OUTPUT.
test_handle = writeto("test.txt")
```

```
appendto(filename)
```

Opens a file named `filename` and sets it as the value of `_OUTPUT`. Unlike `writeto()`, this function does not erase any previous file contents; instead, anything written to the file is appended to its end. If this function fails, it returns `nil`, plus a string describing the error.

Example:

```
--open "test.txt" and set its handle to _OUTPUT.
--Preserves previous content if this file already exists
appendto("test.txt")
```

System Facilities Functions

```
execute(command)
```

This function is equivalent to the C function `system`. It passes commands to be executed by an operating system shell. It returns a status code, which is system-dependent. Note: Command must be enclosed in quotes.

Example:

```
--for Unix/Linux platforms
execute("ls -l")

--for Windows platforms
execute("dir")
```

```
print(e1, e2, ...)
```

Receives any number of arguments, and prints their values using the strings returned by `tostring()`. This function is not intended for formatted output, but only as a quick way to show a value, for instance for debugging.

Example:

```
print("Hello, World")
print("Hello", ",", "World")

test_message = "Hello, World"
print(test_message) --Output is: Hello, World
```

CFD Open Post-Processing Functions

These functions provide access to **FieldView** specific features. The values of the tables listed as input arguments in the following functions are all not necessarily required. In

most cases, if a field is not specified for the input, the default value for that field is assumed.

CFD Data I/O

```
read_dataset(data_input_table)
```

This function is used to read datasets. It presently supports the following file formats: PLOT3D, **FieldView**-UNStructured (both **FieldView**-UNStructured-split format and **FieldView**-UNStructured-combined (grid&results) format) and many other commercial solver exports and formats. Several examples are outlined below. The input is the table `data_input_table`.

It returns `dataset_info_table` if overall dataset read was successful. Otherwise, it returns `nil` plus a string describing the error. For `dataset_info_table` format description, see [print_dataset_table\(dataset_info_table\)](#).

The `data_input_table` form depends on the format of the file being read.

User-defined readers are supported for the `data_format` field. The user must construct the input table according to the type of reader (Combined or Split). The value for `data_format` must match the registered reader name (i.e., the name that would appear in the Data Input menu). Reader names are case-sensitive with spaces allowed (for example, "My Reader"). **FieldView** strips trailing spaces, if present, when a user-defined reader registers its name, so the **FVX** value should have no trailing spaces.

Next, we describe the `data_input_table` for Combined (i.e., Grid & Results in a single file) formats, including Direct Reader formats and the **FieldView**-UNStructured (FV-UNS) Combined format. Direct readers are used to read AcuSolve, CGNS, FLOW-3D, FLUENT, TECPLOT 360, ENSIGHT, LS-DYNA, OpenFOAM, scFLOW, SC/Tetra, scSTREAM, STL, ultraFluidX, VTK and XDB data. Complete details on these readers are provided in [Chapter 1](#).

The `data_format` field is "unstructured" for both the FV-UNS Combined and the FV-UNS Split formats, however the `input_parameters` subtable structure is different.

For FV-UNS files written by surface sampling, the `data_format` field may be "surface_sampled_data".

The Data Input menu ([Figure 11 Data Input pulldown menu](#)) aligns FV-UNS Export files with the commercial solvers that create them, including: Acusim, ANSYS-CFX, CFD++, COBALT, FLUENT, STAR-CD and STAR-CCM+. The `data_input_table` can be configured to work explicitly with any of these solver exports. Alternately, the simple choice of "unstructured" for the `data_format` field will also work.

data_input_table

Combined (single file Grid & Results) formats

Field	Data Type	Comments	Default
data_format	string	“unstructured”, “acusolve_direct”, “acusolve_unstructured”, “ansys-cfx_unstructured”, “cfd++_unstructured”, “cgns_structured”, “cgns_unstructured”, “cgns_unstructured/hybrid”, “cobalt_unstructured”, “flow-3d animation”, “flow-3d restart”, “fluent_unstructured”, “tecplot_360”, “ensight”, “lsdyna”, “lsdyna_d3plot”, “openfoam”, “sc_flow”, “sc_stream”, “sc_tetra”, “starccm_unstructured”, “stared_unstructured”, “stl”, “surface_sampled_data”, “ufx”, “vtk_structured”, “vtk_unstructured/hybrid”, “xdb_import”	n/a
server_config	string	name of server config file without the .srv extension	
server_append	string	“on” or “off”	“off”
input_parameters	table		n/a
name	string	name of the input data file	n/a
options	table		
input_mode	string	“replace” or “append”	“replace”
transient	string	“on” or “off”	“off”
changing_number_of_grids over time	string	“on” or “off” Note: This option applies to only some readers.	“off”
boundary_only	string	“on” or “off”	“off”
grid_processing	string	“less”, “more” or “balanced”	“balanced”
initial_time_step	string or number	'first', 'last', or <integer number> If data_format is "acusolve_direct", “flow-3d animation”, “flow-3d restart” or "openfoam", otherwise ignored; takes precedence if initial_solution_time and/or initial time index are present.	'first'
initial_solution_time	string or number	'first', 'last', or <number> If data_format is "acusolve_direct", “flow-3d animation”, “flow-3d restart” or "openfoam", otherwise ignored; takes precedence if initial_time_index is present, ignored if initial time step is present.	'first'

<code>initial_time_index</code>	string or number	'first', 'last', or <positive integer> If <code>data_format</code> is "acusolve_direct", "flow-3d animation", "flow-3d restart" or "openfoam", otherwise ignored; ignored if <code>initial_time_step</code> or <code>initial_solution_time</code> is present. Use this field if the actual values of the time steps and solution times are not known prior to the data read.	'first'
<code>read_as_steady_state</code>	string	"on" or "off" If <code>data_format</code> is "acusolve_direct", "flow-3d animation", "flow-3d restart" or "openfoam", otherwise ignored	"off"
<code>extended_variables</code>	string	"on" or "off" If <code>data_format</code> is "acusolve_direct", otherwise ignored	"off"
<code>duplicate_boundaries</code>	string	"on" or "off" If <code>data_format</code> is "acusolve_direct", otherwise ignored Note: FieldView 15.1 introduced in change of default FVX behavior for duplicate boundaries, compared to older versions	"off"

Example:

```
data_input_table = {
    data_format = "unstructured",
    input_parameters = {
        -- combined grid and results file
        name = "combinedfile.uns",

        options = {
            input_mode = "replace",
            transient = "off"
        }
    }
}
read_dataset(data_input_table)
```

Example:

```
data_input_table = {
    data_format = "lsdyna",
    input_parameters = {
        name = "d3plot"
    } -- input_parameters
}
read_dataset(data_input_table)
```

Full **FVX** support has been provided to import XDB files. A sample **FVX** listing below illustrates the correct syntax.

```
local dataset_1_info_table = read_dataset( {  
  data_format = "xdb_import",  
  input_parameters = {  
    name = "/usr3/xdb/spitfire.xdb",  
    options = {  
      input_mode = "replace",  
      boundary_only = "off"  
    } -- options  
  } -- input_parameters  
} ) -- read_dataset
```

Next, we describe the `data_input_table` for Split (i.e., Grid & Results in separate files) formats, including the **FieldView**-UNStructured (FV-UNS) Split, NPARC/WIND and WIND US formats.

The `data_format` field is “unstructured” for both the FV-UNS Combined and the FV-UNS Split formats, however the `input_parameters` subtable structure is different.

For FV-UNS files written by surface sampling, the `data_format` field may be “surface_sampled_data”.

data_input_table Split (separate Grid & Results files) formats			
Field	Data Type	Comments	Default
data_format	string	"unstructured", "nparc/wind", "acusolve_unstructured", "ansys-cfx_unstructured", "cfd++_unstructured", "cobalt_unstructured", "fluent_cff_cas/dat_direct", "fluent_cas/dat_direct", "fluent_unstructured", "starcd_unstructured", "starccm_unstructured", "surface_sampled_data", "wind_structured", "wind_unstructured"	n/a
server_config	string	name of server config file without the .srv extension	
server_append	string	"on" or "off"	"off"
input_parameters	table		n/a
grid_file	table		n/a
name	string	name of the input FieldView-UNStructured- grid data file, NPARC/WIND- grid data file	none
options	table		n/a
input_mode	string	"replace" or "append"	"replace"
changing_number_of_grids_over_time	string	"on" or "off" Note: This option applies to only some readers.	"off"
transient	string	"on" or "off"	"off"
boundary_only	string	"on" or "off"	"off"
grid_processing	string	"less", "more" or "balanced" Ignored if present in the options subtable for results_file (it will be an error if present but set to an invalid value)	"balanced"
results_file	table		n/a
name	string	name of the input FieldView-UNStructured- results data file, NPARC/WIND- results data file	none
options	table		n/a
input_mode	string	"replace" or "append"	"replace"
transient	string	"on" or "off"	"off"

Example:

```
data_input_table = {

    data_format = "unstructured",
    -- data_format = "nparc/wind",
    input_parameters = {

        -- grid file information
        grid_file = {
            name = "gridfile.uns",
```

```

        options = {
            input_mode = "replace",
            transient = "off"
        }
    },-- end grid file information

    -- results file information
    results_file = {
        name = "resultsfile.uns",
        options = {
            input_mode = "replace",
            transient = "off"
        }
    }-- end results file information

}-- end input parameters

}-- end data_input_table

read_dataset(data_input_table)

```

The `data_input_table` to support the `read_dataset()` command for PLOT3D & OVERFLOW-2 formats is:

data input table - PLOT3D & OVERFLOW-2 format			
Field	Data Type	Comments	Default
data_format	string	"plot3d" or "overflow-2" Note: Reading the same data using "plot3d" vs. "overflow-2" will result in a different data table and different function names in FieldView .	n/a
server_config	string	name of server config file without the .srv extension	
server_append	string	"on" or "off"	"off"
input_parameters	table		n/a
auto_detect	string	"on" or "off"	"off"
xyz_file	table		
name	string	name of the input xyz data file	
options	table	plot3d_or_overflow-2_options	
grid_point_increment	number		1
q_file	table		
name	string	name of the input 'q' file	
options	table	plot3d_or_overflow-2_options	
function_file	table		
name	string	name of the input function file	
options	table	plot3d_or_overflow-2_options	
names_filename	string	optional	

plot3d_or_overflow-2_options

This is the specification for the options field in the table field `input_parameters` which itself is a field of the table `data_input_table`.

Field	Data Type	Comments	Default
<code>format</code>	string	"binary", "formatted", "unformatted" or "dp_unformatted"	"binary"
<code>input_mode</code>	string	"append" or "replace"	"replace"
<code>auto_partition</code>	string	"on" or "off"	"off"
<code>ghost_cells</code>	number	0, 1 or 2	0
<code>coords</code>	string	coordinate dimension, i.e., "2d" or "3d"	"3d"
<code>multi_grid</code>	string	"on" or "off"	"off"
<code>iblanks</code>	string	"on" or "off"	"off"
<code>transient</code>	string	"on" or "off"	"off"
<code>grid_processing</code>	string	"less", "more" or "balanced" Ignored if present in the options subtable for <code>q_file</code> or <code>function_file</code> (it will be an error if present but set to an invalid value)	"balanced"

Example:

```
--set data file options
plot3d_or_overflow-2_options = {
    format      = "binary",          -- default: binary
    --format=    "dp_unformatted", --double precision unformatted
    input_mode   = "replace",        -- default: replace
    coords       = "3d",             -- default: 3d
    multi_grid   = "on",             -- default: off
    iblanks      = "on",             -- default: off
    transient    = "off",            -- default: off
}

--prepare argument data_input_table
data_input_table = {

    data_format = "plot3d",
    input_parameters = {

        --xyz file information
        xyz_file = {
            name = "impeller3_xyz",
            grid_point_increment = 1,
            options = plot3d_or_overflow-2_options,
        },
    },
}
```

```

--q file information
q_file = {
  name = "impeller3_abs.q",
  options = plot3d_or_overflow-2_options,
},

}, --end input_parameters table

} --end data_input_table

--call function to read dataset
read_dataset(data_input_table)

```

Example:

```

p3ddata= {
  data_format = "plot3d",
  input_parameters = {
    xyz_file = {
      name = "x.1100",
      options = {
        format="unformatted",
        multi_grid = "on",
        iblanks = "on",
      } -- options
    },--xyz file
    q_file = {
      name = "q.1100",
      options = {
        format="unformatted",
        multi_grid = "on",
        iblanks = "on",
      } -- options
    },--q file
  }--input_parameters
}-- p3ddata
read_dataset(p3ddata)

```

An example to read an OVERFLOW-2 dataset follows here:

```

--Set data file options
plot3d_or_overflow-2_options = {
  format = "unformatted",
  input_mode = "replace",

```

```

coords = "3d",
multi_grid = "off",
iblanks = "on",
transient = "off",
}
--Prepare argument data_input_table
data_input_table = {
  data_format = "overflow-2",

  input_parameters = {
    --xyz file information
    xyz_file = {
      name = "grid.in",
      grid_point_increment = 1,
      options = plot3d_or_overflow-2_options,
    },

    --q file information
    q_file = {
      name = "q.gaminf",
      options = plot3d_or_overflow-2_options,
    },

  }, --end input_parameters table
} --end data_input_table

--Call function to read dataset
read_dataset(data_input_table)

```

Next, we describe the `data_input_table` for the Pratt & Whitney Common File format. (Please contact support@tecplot.com for information on the availability of this reader.)

data_input_table			
pw common			
Field	Data Type	Comments	Default
<code>data_format</code>	string	"pw common"	n/a
<code>server_config</code>	string	name of server config file without the .srv extension	
<code>server_append</code>	string	"on" or "off"	"off"
<code>input_parameters</code>	table		n/a
<code>xyz_file</code>	table		
<code>name</code>	string	name of the input data file	n/a
<code>options</code>	table		
<code>input_mode</code>	string	"replace" or "append"	"replace"

grid_processing	string	"less", "more" or "balanced"	"balanced"
-----------------	--------	------------------------------	------------

Finally, we describe the `data_input_table` to be able to append a dataset which has been created using the Dataset Sampling tool (see [Dataset Sampling](#) in **Working with FieldView**).

data_input_table Append Sampled Data			
Field	Data Type	Comments	Default
<code>data_format</code>	string	"append_sampled_data"	n/a
<code>server_config</code>	string	name of server config file without the <code>.srv</code> extension	
<code>server_append</code>	string	"on" or "off"	"off"
<code>input_parameters</code>	table		n/a
<code>name</code>	string	name of the input data file	n/a

In order to be able to append a sampled dataset, the grid target from which this was derived must be read into **FieldView** first. A simple example to illustrate this follows here:

```
grid_target_table = read_dataset( {
    data_format = "unstructured",
    input_parameters = {
        name = "./catheter_mod.fv",
        options = {
            input_mode = "replace",
            boundary_only = "off"
        } -- options
    } -- input_parameters
} ) -- read_dataset

sampled_dataset_table = read_dataset( {
    data_format = "append_sampled_data",
    input_parameters = {
        name = "./base_to_mod_sampled.uns"
    } -- input_parameters
} ) -- read_dataset
```

```
print_dataset_table(dataset_info_table)
```

This function is used to provide text output of the `dataset_info_table` returned by the function `read_dataset()`. The output is sent to `stdout`. The input is the table `dataset_info_table`. This is the table returned by the function `read_dataset()`. The output is text representing the dataset. (See additional notes following the below table.)

dataset_info_table

This table is the output from the function `read_dataset()`. It represents the dataset read-in from a file. If the dataset read was completely successful, the status field will be nil. Otherwise, it will be a table detailing the failure of the operation. If successful, the specific content of the table is dependent on the input table to the function. If table read is unsuccessful, this table will not be returned.

Field	Data Type	Comments	Default
data_format	string		n/a
server_config	string	name of server config file without the .srv extension	
server_append	string	"on" or "off"	"off"
id	number	dataset number	
grid_file	string		
results_file	string	may or may not be present	
function_file	string	may or may not be present	
names_file	string	may or may not be present	
cylindrical	string	"yes" or "no", if id > 0	n/a
machine_axis	string	"X" or "Z", if cylindrical="yes"	
transient	string	"yes" or "no", if id > 0	n/a
cur_time_step	number	if transient="yes"	n/a
has_solution_times	string	if transient="yes", "yes" or "no"	n/a
cur_solution_time	number	if transient="yes" and if has solution time="yes"	n/a
changing_number_of_grids_over_time	string	if transient="yes" Note: This option applies to only some readers.	n/a
rmin	number	if cylindrical="yes"	
rmax	number	if cylindrical="yes"	
tmin	number	if cylindrical="yes"	
tmax	number	if cylindrical="yes"	
xmin	number	if cylindrical="no" or machine_axis="X"	
xmax	number	if cylindrical="no" or machine_axis="X"	
ymin	number	if cylindrical="no"	
ymax	number	if cylindrical="no"	
zmin	number	if cylindrical="no" or machine_axis="Z"	
zmax	number	if cylindrical="no" or machine_axis="Z"	
num_grids	number	total number of grids in the dataset	
grid_info	table	Table of tables, each with grid information. No. of entries depends on num_grids	
structured	string	"yes" or "no"	
num_nodes	number		
num_cells	number	if structured="yes"	
num_elements	number	if structured="no"	
imax	number		
jmax	number		
kmax	number		
q_fsmach	number	if data_format="plot3d"	

q_alpha	number	if data_format="plot3d"	
q_re	number	if data_format="plot3d"	
q_time	number	if data_format="plot3d"	
wind_gamma	number	if data_format="nparc/wind"	
wind_pinf	number	if data_format="nparc/wind"	
wind_tinf	number	if data_format="nparc/wind"	
wind_rgas	number	if data_format="nparc/wind"	
num_regions	number	total number of regions in the dataset	
region_info	table	Table of tables, each with region information. No. of entries depends on num_regions.	
name	string	name of the region	
revolutions_per_second	number		
blades_per_row	number		
period	number		
num_grids	number		
grid_list	table	A table listing grids in the region, i.e., {<number>, <number>, ...}	

Example:

```
--get dataset table
t = read_dataset(data_input_table)

--call print function with handle "t"
print_dataset_table(t)
```

Note: Function `print_dataset_table()` will produce an organized listing of your dataset's characteristics. The *dataset_info_table* can also be viewed with `dumpall(dataset_info_table)` to see what table fields are available for other scripting operations, such as `print(dataset_info_table.grid_info[1].num_nodes)` to print the number of nodes in grid 1.

Creation and Modification of Post-Processing Objects

These functions provide an automated way of accomplishing surface creation, modification and deletion for a given dataset.

The following functions are used to create surfaces and rakes from existing datasets. Each takes a table as input and outputs a handle for the created surface or rake. The handle may then be used for calls to generic functions `modify()`, `delete()` and `query()`.

```
create_boundary(boundary_table)
create_comp(comp_table)
create_iso(iso_table)
create_coord(coord_table)
create_streamline(streamline_table)
read_particle_paths(particle_path_data)
```

Functions to create Annotation objects are:

```
create_text(text_description)
create_arrow(arrow_description)
```

The following summary indicates input table fields shared by multiple functions. Individual fields are explained in subsequent sections according to their categorical use.

Common input fields for surfaces, rakes & annotation

	boundary_table	comp_table	iso_table	coord_table	streamline_table	particle_path_data	text_description	arrow_description
Field								
contours	*	*	*	*				
dataset	*	*	*	*	*	*		
display_type	*	*	*	*	**	**		
geometric_color	*	*	*	*	*	*	*	*
line_type	*	*	*	*	*	*		
number_of_contours	*	*	*	*				
scalar_func	*	*	*	*	*	*		
scalar_colormap	*	*	*	*	*	*		
scalar_range	*	*	*	*	*	*		
show_mesh	*	*	*	*				
show_legend	*	*	*	*	*	*		
show_minmax	*	*	*	*				
threshold_func	*	*	*	*				
threshold_range	*	*	*	*				
transparency	*	*	*	*	*	*		
vector_func	*	*	*	*	*			
visibility	*	*	*	*	*	*	*	*

** see [display_attributes](#)**boundary_table**

This table represents a boundary surface.

Field	Data Type	Comments	Default
dataset	number	dataset number	current dataset
types	table	This is the boundary types option. "all" or "none" or table of boundary types such as {"top_wall", "bot_wall", "inlet", "outlet"}.	"none"
flip_front_back	string	"on" or "off" - Determines which side of a boundary type composed of collocated meshes will be visible.	"off"
geometric_color	number or string	"white", "black", or number ranging from 1 to 8	
show_mesh	string	"on" or "off"	"off"
contours	string	"none", "black", "white", "scalar", "geometric" Setting contours to any value other than "none" has no effect if scalar_func is "none".	"none"
number_of_contours	number	If "Filled Contour" is OFF, max = 500 If "Filled Contour" is ON, max = 100, if value specified is higher, level reduced to 100.	16
transparency	number	Range is from 0 to 1; level is rounded to nearest value in 0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1. Query returns above values.	0
line_type	string	"thin", "medium", "thick"	"thin"
scalar_func	string	Scalar function name or "none"	"none"
scalar_range	table	double_range	n/a
min	number or string	"*" sets min to the value of abs_min (returned by query())	n/a
max	number or string	"*" sets max to the value of abs_max (returned by query())	n/a
use_local	string	"on" or "off"	"off"
scalar_colormap	table	See Scalar Colormap Specification	n/a
show_legend	string	"on" or "off" see FVX Legends	"off"
show_minmax	table	"on" or "off" see FVX Show Min Max Annotation	"off"
vector_func	string	Vector function name or "none" When vector_func has been specified for a surface, the display_type parameter will have no visible effect because the surface will be displayed as vectors.	"none"
threshold_func	string	Threshold function name or "none"	"none"
threshold_range	table	double_range	n/a
min	number or string	"*" sets min to the value of abs_min (returned by query())	n/a
max	number or string	"*" sets max to the value of abs_max (returned by query())	n/a
X_clip	table	double_range Only valid with Cartesian or RTX cylindrical coordinate systems	n/a
min	number or string	"*" sets min to the value of abs_min (returned by query())	n/a
max	number or string	"*" sets max to the value of abs_max (returned by query())	n/a
Y_clip	table	double_range Only valid with Cartesian coordinate systems	n/a
min	number or string	"*" sets min to the value of abs_min (returned by query())	n/a

max	number or string	“*” sets max to the value of abs_max (returned by query())	n/a
Z_clip	table	double_range Only valid with Cartesian or RTZ cylindrical coordinate systems	n/a
min	number or string	“*” sets min to the value of abs_min (returned by query())	n/a
max	number or string	“*” sets max to the value of abs_max (returned by query())	n/a
R_clip	table	double_range Only valid with cylindrical coordinate systems	n/a
min	number or string	“*” sets min to the value of abs_min (returned by query())	n/a
max	number or string	“*” sets max to the value of abs_max (returned by query())	n/a
T_clip	table	double_range Only valid with cylindrical coordinate systems	n/a
min	number or string	“*” sets min to the value of abs_min (returned by query())	n/a
max	number or string	“*” sets max to the value of abs_max (returned by query())	n/a
vector_options	table	See Vector Options	n/a
visibility	string	“on” or “off”	“on”
display_type	string	“constant_shading”, “faceted_shading”, “smooth_shading”, “mesh_shading”, “contour_lines”, “outline_edges”, “vertices”, “shaded_vertices”	“mesh_shading”

Example:

```

boundary_table1 = {
    scalar_func = "Normalized density [PLOT3D]",
    types = {"body", "wing"},
    display_type = "smooth_shading"
}

--create boundary surface and assign handle to
--"surface_handle1"
surface_handle1 = create_boundary(boundary_table1)

boundary_table2 = {
    vector_func = "Velocity Vectors [PLOT3D]",
    threshold_func = "X",
    threshold_range = {min=2.4,max=2.5},
    types = {"tail"}
}

--create boundary surface and assign handle to
--"surface_handle2"
surface_handle2 = create_boundary(boundary_table2)

```

A subtable named 'material' has been added to the input table for the `create_boundary()` function:

```
material = {
  name = "Glossy Paint",
  -- Some materials also need a color, and there will be a
  -- default color used for each of those materials.
  -- The 'color' subfield can be used to change that color.
  -- It will be ignored unless the material is
  --   Aluminum
  --   Glass
  --   Glossy or Matte Paint
  --   Shiny or Dull Plastic

  -- When using query(), the returned 'material' subtable will
  -- have the 'color' subfield only when applicable.

  color = {
    red   = <number>,  -- Valid range from 0 through 255
    green = <number>,
    blue  = <number>
  } -- color
} -- material
```

Create / Modify

The 'material' subtable will be accepted as valid input for boundary surface objects only, under the following conditions:

'scalar_func' is "none"

'display_type' is "faceted_shading" or "smooth_shading"

It will be an error if both of these conditions are not met.

The following is a list of material names that will be recognized:

```
"None"
"Aluminum"
"Brass"
"Bronze"
"Chrome"
"Copper"
"Gold"
"Iron"
"Silver"
```

"Steel"
 "Titanium"
 "Glass"
 "Rubber"
 "Glossy Paint"
 "Matte Paint"
 "Shiny Plastic"
 "Dull Plastic"

As **Tecplot Inc.** distributes additional materials in future releases, they will also be accepted.

The 'color' subfield of the material input table will be ignored unless the current material allows color to be specified. The materials that allow color to be specified are:

"Aluminum"
 "Glass"
 "Glossy Paint"
 "Matte Paint"
 "Shiny Plastic"
 "Dull Plastic"

Query

The 'material' field will be returned in the query table if the boundary surface is using one.

The 'color' subfield of the material input table will be present only if the current material allows color to be specified.

Environments

An **FVX** command has been introduced:

```
set_environment(input_table)
```

input_table			
The table argument to <code>set_environment()</code> contains one or more of these input tables:			
Field	Data Type	Comments	Default
window	number or string	number or "current"	1
environment	string	"Default", "Courtyard", "Street", "Sky", "Sky and sea" or "Sky and grass"	
background	table		
color	number or string	"white", "black", or number ranging from 1 to 8	
image	string	"none" or filename, including path	
position	string	"center", "stretch", or "fit"	

If not explicitly set by the user, color will be "black", image will be "none" and position will be "center". If explicitly set, the values will persist.

Example:

```
set_environment({
  { window = "current",
    environment = "Courtyard",
    background = {
      color = 4,
      image = "/home/fv-test/Pictures/img2.png",
      position = "stretch"
    }, -- background
  },
}) -- set_environment
```

The environment is a windows-level attribute, not an object attribute. Window corresponds to the number visible above split sub-windows visible in the **FieldView** graphics window. If a value for window is not provided, it defaults to 1.

It is required that the environment or background field be included in the input_table.

By taking a table of tables, set_environment() can set up multiple windows with one call.

```
output = set_environment( {
  {
    window = 1,
    environment = "Courtyard",
  },
  {
    window = 2,
    environment = "Street",
  },
  {
    window = 3,
    environment = "Sky and sea",
  },
  {
    window = 4,
    environment = "Sky and grass",
  },
},
)
```

comp_table			
This table of fields represents a computational surface.			
Field	Data Type	Comments	Default
dataset	number	dataset number	current dataset
grid	number	number of the desired grid	
axis	string	"I" or "J" or "K"	"I"
I_inc	number		
I_axis	table	if axis = "I", triple_range, else double_range	
min	number or string	"*" sets min to the value of abs_min (returned by query())	
current	number	if axis = "I"	
max	number or string	"*" sets max to the value of abs_max (returned by query())	
J_inc	number		
J_axis	table	if axis = "J", triple_range, else double_range	
min	number or string	"*" sets min to the value of abs_min (returned by query())	
current	number	if axis = "J"	
max	number or string	"*" sets max to the value of abs_max (returned by query())	
K_inc	number		
K_axis	table	if axis = "K", triple_range, else double_range	
min	number or string	"*" sets min to the value of abs_min (returned by query())	
current	number	if axis = "K"	
max	number or string	"*" sets max to the value of abs_max (returned by query())	
geometric_color	number or string	"white", "black", or number ranging from 1 to 8	
show_mesh	string	"on" or "off"	"off"
contours	string	"none", "black", "white", "scalar", "geometric" Setting contours to any value other than "none" has no effect if scalar_func is "none".	"none"
number_of_contours	number	If "Filled Contour" is OFF, max = 500 If "Filled Contour" is ON, max = 100, if value specified is higher, level reduced to 100.	16
transparency	number	Range is from 0 to 1; level is rounded to nearest value in 0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1. Query returns above values.	0
line_type	string	"thin", "medium", "thick"	"thin"
scalar_func	string	Scalar function name or "none"	"none"
scalar_range	table	double_range	n/a
min	number or string	"*" sets min to the value of abs_min (returned by query())	n/a
max	number or string	"*" sets max to the value of abs_max (returned by query())	n/a
use_local	string	"on" or "off"	"off"
scalar_colormap	table	See Scalar Colormap Specification	n/a
show_legend	string	"on" or "off" see FVX Legends	"off"

show_minmax	table	“on” or “off” see FVX Show Min Max Annotation	“off”
threshold_func	string	threshold function name or “none”	“none”
threshold_range	table	double_range	
min	number or string	“*” sets min to the value of abs_min (returned by query())	
max	number or string	“*” sets max to the value of abs_max (returned by query())	
vector_func	string	vector function name or “none” When vector_func has been specified for a surface, the display_type parameter will have no visible effect because the surface will be displayed as vectors.	“none”
vector_options	table	See Vector Options	n/a
visibility	string	“on” or “off”	“on”
display_type	string	“constant_shading”, “faceted_shading”, “smooth_shading”, “mesh_shading”, “contour_lines”, “vertices”, “shaded_vertices”	“mesh_shading”

Example:

```
--define computational surface
comp_table = {
    dataset = 3,
    grid = 2,
    axis = "I",
    I_inc = 1,

    I_axis = {
        min = 1,
        current = 1,
        max = 20,
    },

    scalar_func = "Density (Q1)",
    vector_func = "Velocity Vectors [PLOT3D]",

    threshold_func = "Density (Q1)",
    threshold_range = {
        min = 0.75,
        max = 1.01,
    },

    visibility = "on",
}

-- call function and assign handle to "c"
c = create_comp(comp_table)
```

iso_table This table represents an iso-surface.			
Field	Data Type	Comments	Default
dataset	number	dataset number	current dataset
iso_func	string or table	iso function name if data type is string or mode and subtables if data type is table, REQUIRED	
mode	string	"point_and_normal" or "points_in_plane"	
pt1	table	{ x, y, z }	
pt2	table	{ x, y, z }	
pt3	table	{ x, y, z } exists if mode = "points_in_plane"	
iso_value	table	triple_range,	
min	number or string	"*" sets min to the value of abs_min (returned by query())	
current	number		
max	number or string	"*" sets max to the value of abs_max (returned by query())	
geometric_color	number or string	"white", "black", or number ranging from 1 to 8	
show_mesh	string	"on" or "off"	"off"
unrolled	string	"on" or "off"	"off"
contours	string	"none", "black", "white", "scalar", "geometric" Setting contours to any value other than "none" has no effect if scalar_func is "none".	"none"
number_of_contours	number	If "Filled Contour" is OFF, max = 500 If "Filled Contour" is ON, max = 100, if value specified is higher, level reduced to 100.	16
transparency	number	Range is from 0 to 1; level is rounded to nearest value in 0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1. Query returns above values.	0
line_type	string	"thin", "medium", "thick"	"thin"
scalar_func	string	Scalar function name or "none"	"none"
scalar_range	table	double_range	n/a
min	number or string	"*" sets min to the value of abs_min (returned by query())	n/a
max	number or string	"*" sets max to the value of abs_max (returned by query())	n/a
use_local	string	"on" or "off"	"off"
scalar_colormap	table	See Scalar Colormap Specification	n/a
show_legend	string	"on" or "off" see FVX Legends	"off"
show_minmax	table	"on" or "off" see FVX Show Min Max Annotation	"off"
vector_func	string	Vector function name or "none" When vector_func has been specified for a surface, the display_type parameter will have no visible effect because the surface will be displayed as vectors.	"none"
vector_options	table	See Vector Options	n/a
threshold_func	string	Threshold function name or "none"	"none"
threshold_range	table	double_range	
min	number or string	"*" sets min to the value of abs_min (returned by query())	

max	number or string	"*" sets max to the value of abs_max (returned by query())	
visibility	string	"on" or "off"	"on"
display_type	string	"constant_shading", "faceted_shading", "smooth_shading", "mesh_shading", "contour_lines", "crinkle", "vertices", "shaded_vertices"	"constant_shading"

Example:

```
--define iso surface
iso_table = {
  dataset = 2,
  iso_func = {
    mode = "point_and_normal",
    pt1 = {1,0,0},
    pt2 = {1,1,0},
  },
  iso_value = {
    min = -10,
    current = 0,
    max = 20,
  },
  scalar_func = "Cp [PLOT3D]",
  vector_func = "none",
  threshold_func = "Enthalpy [PLOT3D]",
  threshold_range = {
    min = 2.4,
    max = 2.5,
  },
  visibility = "on",
}

--create iso surface and assign handle to variable "s"
s = create_iso(iso_table)
```

coord_table This table represents a co-ordinate surface.			
Field	Data Type	Comments	Default
dataset	number	dataset number	current dataset
axis	string	Cartesian: "X" or "Y" or "Z". RTZ cylindrical: "R" or "T" or "Z". RTX cylindrical: "R" or "T" or "X".	"X" or "R"
geometric_color	number or string	"white", "black", or number ranging from 1 to 8	
show_mesh	string	"on" or "off"	"off"

contours	string	"none", "black", "white", "scalar", "geometric" Setting contours to any value other than "none" has no effect if scalar_func is "none".	"none"
number_of_contours	number	If "Filled Contour" is OFF, max = 500 If "Filled Contour" is ON, max = 100, if value specified is higher, level reduced to 100.	16
transparency	number	Range is from 0 to 1; level is rounded to nearest value in 0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1. Query returns above values.	0
line_type	string	"thin", "medium", "thick"	"thin"
scalar_colormap	table	See table Scalar Colormap Specification	n/a
scalar_func	string	Scalar function name or "none"	"none"
scalar_range	table	double_range	n/a
min	number or string	"*" sets min to the value of abs_min (returned by query())	n/a
max	number or string	"*" sets max to the value of abs_max (returned by query())	n/a
use_local	string	"on" or "off"	"off"
scalar_colormap	table	See Scalar Colormap Specification	n/a
show_legend	string	"on" or "off" see FVX Legends	"off"
show_minmax	table	"on" or "off" see FVX Show Min Max Annotation	"off"
vector_func	string	Vector function name or "none" When vector_func has been specified for a surface, the display_type parameter will have no visible effect because the surface will be displayed as vectors.	"none"
vector_options	table	See Vector Options	n/a
threshold_func	string	Threshold function name or "none"	"none"
threshold_range	table	double_range	
min	number or string	"*" sets min to the value of abs_min (returned by query())	
max	number or string	"*" sets max to the value of abs_max (returned by query())	
visibility	string	"on" or "off"	"on"
X_axis	table	if axis = "X", triple_range, else double_range. Exists if Cartesian or RTX cylindrical.	
min	number or string	"*" sets min to the value of abs_min (returned by query())	
current	number	if axis = "X"	
max	number or string	"*" sets max to the value of abs_max (returned by query())	
Y_axis	table	if axis = "Y", triple_range, else double_range. Exists if Cartesian.	
min	number or string	"*" sets min to the value of abs_min (returned by query())	
current	number	if axis = "Y"	
max	number or string	"*" sets max to the value of abs_max (returned by query())	
R_axis	table	if axis = "R", triple_range, else double_range. Exists if cylindrical.	
min	number or string	"*" sets min to the value of abs_min (returned by query())	
current	number	if axis = "R"	

max	number or string	“*” sets max to the value of abs_max (returned by query())	
T_axis	table	if axis = “T”, triple_range, else double_range. Exists if cylindrical.	
min	number or string	“*” sets min to the value of abs_min (returned by query())	
current	number	if axis = “T”	
max	number or string	“*” sets max to the value of abs_max (returned by query())	
Z_axis	table	if axis = “Z”, triple_range, else double_range. Exists if Cartesian or RTZ cylindrical.	
min	number or string	“*” sets min to the value of abs_min (returned by query())	
current	number	if axis = “Z”	
max	number or string	“*” sets max to the value of abs_max (returned by query())	
display_type	string	“constant_shading”, “smooth_shading”, “mesh_shading”, “contour_lines”, “crinkle”, “vertices”, “shaded_vertices”	“mesh_shading”
sweep_steps	number	Range is from 3 to 32767.	25
vector_options	table	See Vector Options	n/a

FVX support is included for ruled grids for Coordinate surfaces created on Cartesian datasets. Two fields, `ruled_grid` and `ruled_grid_options` (a table), are included in the input and query tables for Coordinate surfaces.

ruled_grid_subtable			
This is a subtable of coord_table.			
Field	Data Type	Comments	Default
ruled_grid	string	“on” or “off”	“off”
ruled_grid_options	table		
color	number or string	“white”, “black” or number ranging from 1 to 8	
font	string	“lee”, “lee bold”, “lee italic”, “lee bold italic”, “leemono”, “leemono bold”, “leemono italic”, “leemono bold italic”, “leese”, “leese bold”, “noto”, “roman sans serif”, “roman”, “italics”, “script”	“lee”
size	number	integer to select font size, range 1 to 100	10
horizontal_axis	table		
label	string	“X” or “Y” or “Z”, READ ONLY Determined by the current value of coord_table ‘axis’ field.	
interval	number	“default”, float greater than 0 “default” is an input value only; provided as a way to get back to the default calculated value. A subsequent query() will return the default calculated value.	
grid_lines	string	“on” or “off”	“on”
tick_marks	string	“on” or “off”	“on”
labels	string	“on” or “off”	“on”

labels_parameters	table		
numerical_format	string	"floating_point" or "exponential"	"floating_point"
decimal_places	number	integer to select number of significant digits in labels, range is 0-6	3
vertical_axis	table		
label	string	"X" or "Y" or "Z", READ ONLY Determined by the current value of coord_table 'axis' field.	
interval	number	"default", float greater than 0 "default" is an input value only; provided as a way to get back to the default calculated value. A subsequent query() will return the default calculated value.	
grid_lines	string	"on" or "off"	"on"
tick_marks	string	"on" or "off"	"on"
labels	string	"on" or "off"	"on"
labels_parameters	table		
numerical_format	string	"floating_point" or "exponential"	"floating_point"
decimal_places	number	integer to select number of significant digits in labels, range is 0-6	3

Example:

```

--define coordinate surface
coord_table = {
  dataset = 1,
  scalar_func = "Pressure [PLOT3D]",
  vector_func = "Vorticity Vectors [PLOT3D]",

  threshold_func = "Density (Q1)",
  threshold_range = {
    min = 0.75,
    max = 0.82,
  },

  visibility = "on",

  axis = "X",
  X_axis = {
    min = -6,
    current = -1.1,
    max = 8,
  }
}

--create coordinate surface and assign handle to "c"
c = create_coord(coord_table)

```

streamline_table			
This table is the input argument for <code>create streamline()</code> function.			
Field	Data Type	Comments	Default
vector_func	string	Vector function name. The field is required for <code>create streamline()</code>.	n/a
dataset	number	dataset number	current dataset
geometric_color	number or string	"white", "black", or, number ranging from 1 to 8	
scalar_func	string	Scalar function name or "none"	"none"
scalar_range	table	double_range	n/a
min	number or string	"*" sets min to the value of abs_min (returned by query())	n/a
max	number or string	"*" sets max to the value of abs_max (returned by query())	n/a
use_local	string	"on" or "off"	"off"
scalar_colormap	table	See table Scalar Colormap Specification	n/a
show_legend	string	"on" or "off" see FVX Legends	"off"
line_type	string	"thin", "medium", "thick"	"thin"
transparency	number	Range is from 0 to 1; level is rounded to nearest value in 0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1. Query returns above values.	0
visibility	string	"on" or "off"	"on"
seeding	table	seeding_input_table The presence of the field (table) will re-seed the rake, deleting seeds that may have already existed.	n/a
display_seeds	string	"on" or "off"	"on"
calculation_parameters	table		
direction	string	"forward" or "backward" or "both"	"forward"
step	number	Number of steps, an integer number.	3
time_limit	string or number	"none" or positive real number.	"none"
release_interval	number	Positive integer number. How often new particles are released for streaklines.	3
duration	number	Positive integer number. For how long new particles are released for streaklines.	1

There are 6 different formats for the `seeding_input_table`. Choice of a particular format depends on the values of `seed_coord` and `mode` fields:

seeding_input_table Formats

seed_coord	mode	
	"seed_a_surface"	"add"
"IJK_real"	Format 1	Format 4
"XYZ"	Format 1	Format 5
"RTZ"	Format 1	Format 6
"IJK_int"	Format 2	Format 3

The appropriate subtables must be present for the specified value of `seed_coord`. It will not be an error if other subtables are also present; they will be ignored. For example, `seed_coord="XYZ"` requires that the subtables "x", "y", and "z" be present; if an "i" subtable is also present, it will be simply ignored.

Possible Errors:

- There are gaps in the `seeds` subtable of the `seeding_input_table`. In other words, seeds must appear in a continuous block from seed number 1 to the total number of seeds.
- The subtables contain non-numerical data.

`seeding_input_table` formats for seeding a surface are:

seeding_input_table ; Format 1			
This table is an input argument for streamline table described above.			
Field	Data Type	Comments	Default
seed_coord	string	"IJK_real", or "XYZ", or "RTZ".	none
mode	string	"seed_a_surface"	none
seeding_surface	handle	A surface handle.	none
seeds_to_add	number	How many seeds to add on a seeding surface. Positive integer number.	none

seeding_input_table ; Format 2			
This table is an input argument for streamline table described above.			
Field	Data Type	Comments	Default
seed_coord	string	"IJK_int"	none
mode	string	"seed_a_surface"	none
seeding_surface	string	A surface handle.	none
inc	number	Whether to seed on every grid point (inc=1), every other grid point (inc=2), etc. Positive integer number.	none

seeding_input_table ; Format 3

This table is an input argument for **streamline table** described above.

Field	Data Type	Comments	Default
seed_coord	string	"IJK_int"	none
mode	string	"add"	none
seeds	table	Table of seed points; I, J and K are in their respective tables within this table	none
i	table	Table of integer 'I' values	none
j	table	Table of integer 'J' values	none
k	table	Table of integer 'K' values	none
grid	table	Table of grid numbers. Optional. One grid number for each seed.	Grid 1 in the current dataset.



Note: If any *i*, *j*, or *k* values are specified as real numbers, they will be truncated to integers. For example, if *i*={1.2, 2.9}, it will be transformed to *i*={1, 2}.

seeding_input_table ; Format 4

This table is an input argument for **streamline table** described above.

Field	Data Type	Comments	Default
seed_coord	string	"IJK_real".	none
mode	string	"add"	none
seeds	table	Table of seed points; I, J and K are in their respective tables within this table	none
i	table	Table of real 'I' values	none
j	table	Table of real 'J' values	none
k	table	Table of real 'K' values	none
grid	table	Table of grid numbers. Optional. One grid number for each seed.	Grid 1 in the current dataset.



Note: Real *i*, *j*, and *k* values may be used to place a seed between grid nodes.

seeding_input_table ; Format 5

This table is an input argument for **streamline table** described above.

Field	Data Type	Comments	Default
seed_coord	string	"XYZ".	none
mode	string	"add"	none
seeds	table	Table of seed points; X, Y and Z are in their respective tables within this table	none
x	table	Table of 'X' coordinates	none

y	table	Table of 'Y' coordinates	none
z	table	Table of 'Z' coordinates	none



Note: x, y and z coordinates are treated as real values (not truncated).

seeding_input_table ; Format 6			
This table is an input argument for streamline table described above.			
Field	Data Type	Comments	Default
seed_coord	string	"RTZ".	none
mode	string	"add"	none
seeds	table	Table of seed points; R, T and Z are in their respective tables within this table	none
r	table	Table of 'R' coordinates	none
t	table	Table of 'T' coordinates	none
z	table	Table of 'Z' coordinates	none



Note: r, t and z coordinates are treated as real values (not truncated).

Example:

```
seeding_input_table = {
  seed_coord = "XYZ",
  mode = "add",
  seeds = {
    x = {1., 2.0, 3},
    y = {4.5, 5, 6.25},
    z = {7, 8, 9}
  }
}

streamline_table = {
  vector_func = "Velocity Vectors [PLOT3D]",
  scalar_func = "Density (Q1)",

  dataset = 1,
  visibility = "on",

  seeding = seeding_input_table,
```

```

display_seeds = "on",

calculation_parameters = {
    direction = "both",
    step = 5,
    time_limit = 20,
    release_interval = 10,
    duration = 3
}
}

rake_handle = create_streamline(streamline_table)

```

Notes:

- Individual seeds may be rejected, usually because the seed is outside the bounds of the dataset. A seed could also be rejected if the maximum number of seeds is already reached, or **FieldView** could not allocate memory to store the seed. Under the "add" mode, these rejections will be silent. Therefore, the user is strongly urged to `query()` after seeding to verify the list of seeds that were accepted.
- Modifying the `seed_coord` field will delete all existing seeds.
- FVX** seeding is not incremental. The presence of the `seeding` field (subtable) in the `streamline_table` for a rake will re-seed the rake, deleting any existing seeds.
- When seeding a surface, it will be an error if the specified surface is not appropriate for the current `seed_coord` value.

```
set_streamlines_display (display_attributes)
```

This command allows choice of how to display the streamlines. In general, it works like the **DISPLAY TYPE** option in the GUI. `set_streamlines_display` takes `display_attributes` table as an input and applies attributes specified in the table to all rakes on all datasets in memory.

display_attributes			
The table is an input argument for <code>set_streamlines_display</code> and <code>set_particle_paths_display</code> .			
Field	Data Type	Comments	Default
<code>display_type</code>	string	"complete", "filament", "filament_arrows", "filament_spheres", "growing", "dots", "spheres_and_lines", "spheres", "polyspheres"+, "lines of spheres", "lines of dots", "ribbons"*	** "complete"
<code>ribbon_width</code>	number	Used for "ribbons" display type. Positive integer number less than or equal to 1024.	32
<code>scale</code> <code>sphere_scale</code> (deprecated)	number	Sphere (or arrow) scale. Used for display types using spheres or arrows. Positive floating point number.	1.0
<code>scalar_sizing</code> <code>scalar_sphere_sizing</code> (deprecated)	string	Used to control whether spheres (or arrows) will be sized according to the current Scalar Function. "on" or "off".	"off"

length_fraction	number	Parameter used to control length of longest streamline used in calculation of filament length.	0.9
animate	string	This parameter specifies if the animated particle paths move forward or backward in time. "up", "down", "off".	"off"
animate_divs	number	The number of segments that the longest streamline will be divided into. Used for "filament" display type. Integer number between 1 and 1000.	25

* "ribbons" display_type is invalid for set_particle_paths_display()

** Default type is "spheres" for transient Particle Paths.

+ Type "polyspheres" applies only to Particle Paths. The only valid types for transient particle paths are "spheres", "polyspheres" and "dots".

Example:

```
display_attributes = {display_type="spheres"}
set_streamlines_display (display_attributes)
```

Notes:

- scale and scalar_sizing have an effect only if display_type is one of the sphere (or arrow) types; no error is produced if either is specified with no effect.
- ribbon_width has an effect only if display_type is "ribbons"; no error is produced if specified with no effect.
- An error will be produced if set_streamlines_display() is called when no rakes exist on any dataset in memory.

query_streamline_display()

This command returns the global display settings for streamline rake(s). This command returns a display attributes table.

particle_path_data			
This table is an input argument for read_particle_paths command.			
Field	Data Type	Comments	Default
dataset	number	dataset number	current dataset
visibility	string	"on" or "off"	"on"
geometric_color	number or string	"white", "black", or, number ranging from 1 to 8	
format	string	"fv particle path", "fv particle path direct", "fvp", "fvp direct", "star-cd trk", "star-cd 33", "fidap FDPART", "cfx-4 trk", "xdb import"	none
filename	string	The name of the particle path file to be read.	
scalar_colormap	table	See Scalar Colormap Specification	n/a
show_legend	string	"on" or "off" see FVX Legends	"off"

scalar_func	string	Use this to specify the name of a flow-field scalar function. Do not specify a value for ppath_func if this is already specified.	'none'
ppath_func	string	Use this function to specify the name of a path variable, or "Emission Time" or "Path Tag Number". Do not specify a value for scalar_func if this is already set.	'none'
scalar_range	table	table containing absolute, local and specified range for either the scalar or ppath function	
max	number or string	Specified maximum value for the scalar or ppath function "*" sets max to the value of abs_max (returned by query())	
min	number or string	Specified minimum value for the scalar or ppath function "*" sets min to the value of abs_min (returned by query())	
local	string	"on" or "off", when specified "on", the local range for the scalar ppath or function is used.	
path_variables	table	Returned following a query	
tags	table	Returned following a query	
transparency	number	Range is from 0 to 1; level is rounded to nearest value in 0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1. Query returns above values.	0
select_by	table	Particle path thresholding methods	
initial_value_variable	string	'none' or the name of a ppath variable	'none'
initial_value_range	table		
max	number or string	Specified maximum for the initial value variable "*" sets max to the value of abs_max (returned by query())	
min	number or string	Specified minimum for the initial value variable "*" sets min to the value of abs_min (returned by query())	
value_on_path_variable	string	'none' or the name of a ppath variable	'none'
value_on_path_range	table		
max	number	Specified maximum for the value on path variable "*" sets max to the value of abs_max (returned by query())	
min	number	Specified minimum for the value on path variable "*" sets min to the value of abs_min (returned by query())	
tags	string or table	'none', 'all', a valid tag name or a table of valid tag names	'none'
line_type	string	"thin", "medium", "thick"	"thin"

Notes on the particle path file format

The format definition, "fvp", is synonymous with "fv particle path" and is provided as an abbreviated form for user convenience.

Reading particle path data with **FieldView** running in Client-Server mode can become confusing since the dataset can be read remotely while the particle path data can be read either remotely or locally. If the dataset has been read locally (also known as using

Direct Mode), the particle path data is expected to be read locally as well. So, if the dataset was read using a Server, then by default, **FieldView** will read particle path data from the same Server. In this case, using “fv particle path” or “fvp” will direct **FieldView** to read the FV particle path data from that Server.

However, there is an additional option available if the dataset was read from a server. You may also import an FV particle path file from the Client (also known as using local mode). Only the **FieldView** particle path format files can be read in this way. To read **FieldView** particle path data when the dataset has been read from a Server, use either “fv particle path direct” or “fvp direct”.

FieldView currently allows only one particle path file per dataset. With this **FVX** command, each call to `read_particle_paths()` will silently replace the dataset's current particle paths, if any.

Particle path subsetting provides powerful features to modify the visual content of the data from any particle path file. The selection criteria based on either the `initial_value_variable` or the `value_on_path_variable` will only accept path variable names. They are not able to accept "Emission Time" or "Path Tag Number". One or more "Path Tag Number" can be used for subsetting using the 'Select by Tag' field on the GUI. The equivalent subsetting features expressed in **FVX** would look similar to:

```
select_by = {
    initial_value_variable = "Density (Q1)",
    initial_value_range = {max = 1.1},
    tags = {"Path-2", "Path-1"},
},
```

In some cases, particle path files may contain locations which lie outside the bounds of the dataset. When `scalar_func` has been set to a flow-field scalar function, it is not possible to obtain data for these points which lie outside the dataset volume. When this happens, **FieldView** generates this warning:

```
WARNING
One or more points from the particle path data
lie outside the bounds of the dataset.
```

Normally, this warning is displayed in a popup. However, when running an **FVX** script, the warning will simply be printed to console window.

Example:

```
local particle_paths_1 = read_particle_paths(
{
    scalar_colormap = {
        name = "spectrum",
        invert = "off",
```

```

        filled_contour = "off",
    }, -- scalar_colormap
    ppath_func = "Duration",
    format = "xdb import",
    number_of_contours = 16,
    geometric_color = 4,
    line_type = "thin",
    scalar_range = {
        min = 0,
        local_min = 0,
        max = 0.2000000029802322,
        local_max = 0.2000000029802322,
        use_local = "off",
    }, -- scalar_range
    select_by = {
        initial_value_variable = "none",
        value_on_path_variable = "none",
        tags = "none",
    }, -- select_by
    visibility = "on",
    scalar_func = "none",
    dataset = 1,
    filename = "/usr3/xdb/spitfire.xdb",
    }
) -- particle_paths_1

```

set_particle_paths_display(display_attributes)

This function is similar to the set_streamlines_display command; refer to the [display_attributes](#) input table.

Example: Read in a STAR-CD .trk file, and set the display type to spheres.

```

my_ppath = read_particle_paths({
    dataset = 2,
    visibility = "on",
    format = "star-cd trk",
    filename = "testcase.trk",
    ppath_func = "Diameter",
})

set_particle_paths_display({
    display_type = "spheres",
    sphere_scale = 0.25,
    scalar_sizing = "on"
})

```

Example: Use the `select_by` options to modify the display of a particle path file.

```
my_ppath = read_particle_paths({
  dataset = 1,
  visibility = "on",
  format = "fidap FDPART",
  filename = "bloodcell.FDPART",
  ppath_func = "Diameter",
})
```

The first step is to read the particle path data.

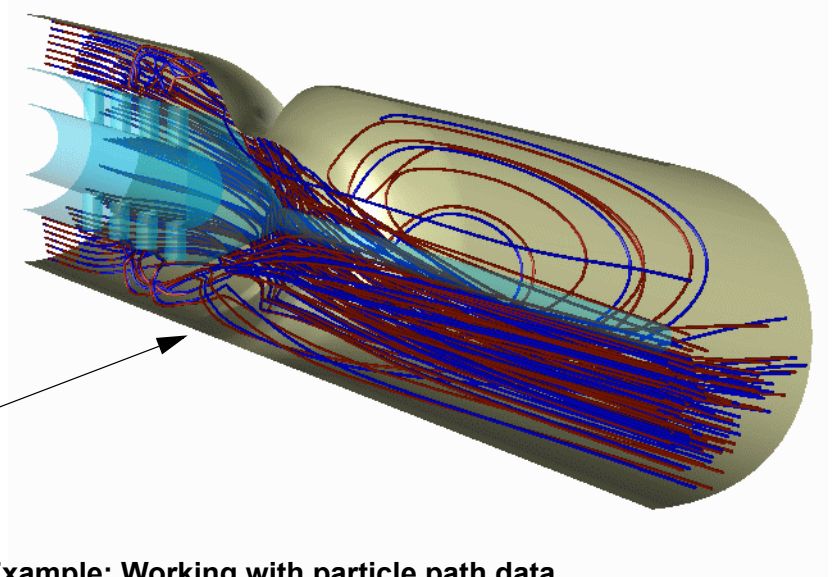


Figure 87 FVX Example: Working with particle path data

```
modify (my_ppath, {
  select_by = {
    initial_value_variable =
      "Diameter",
    initial_value_range = { min = 2 },
  }
})
```

Use the `select_by` option to select only larger particles.

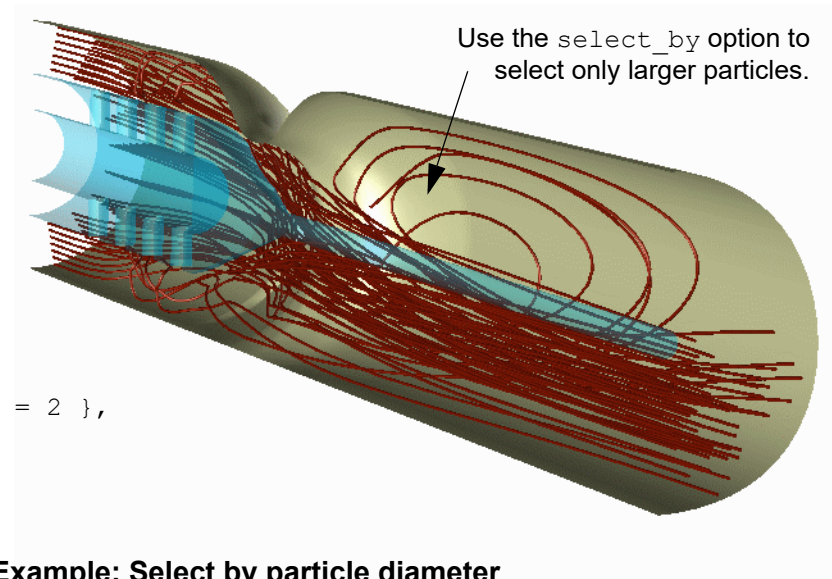


Figure 88 FVX Example: Select by particle diameter

```

modify (my_ppath, {
  select_by = {
    value_on_path_variable =
      "wp_velocity",
    value_on_path_range = { max = 0.0 },
  },
  ppath_func = "TIME"
})

```

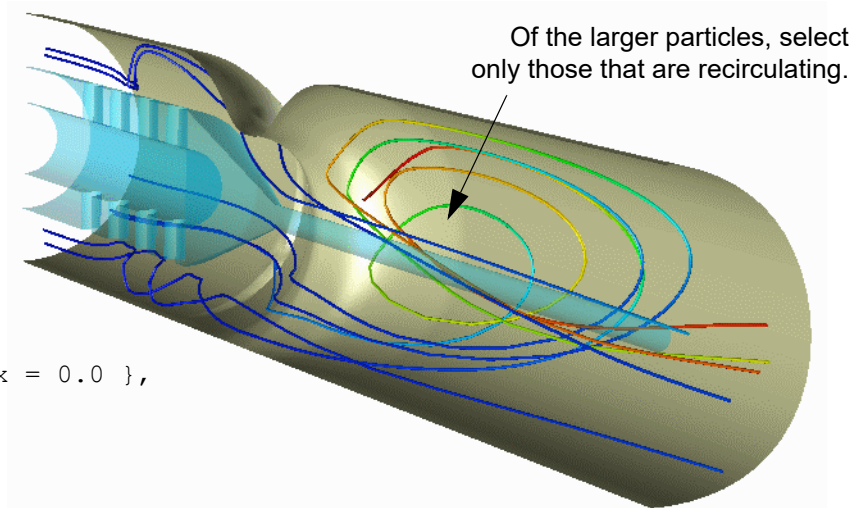


Figure 89 FVX Example: Select only recirculating trajectories

```

set_particle_paths_display({
  display_type = "filament_spheres",
  sphere_scale = 0.5,
})

```

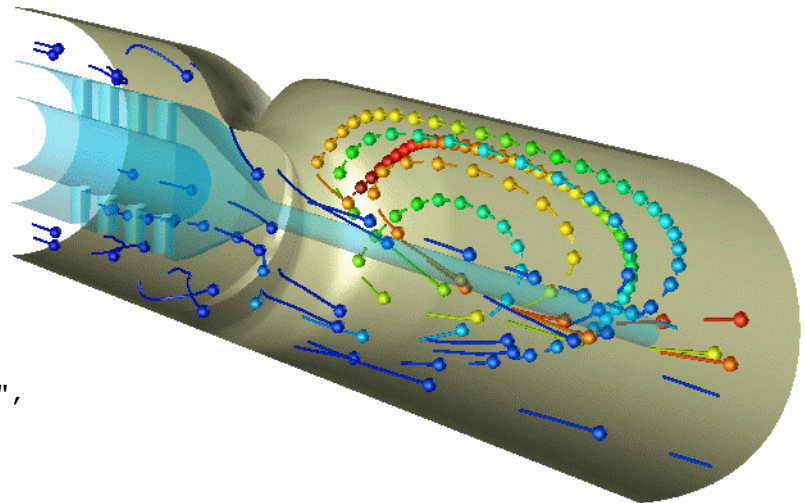


Figure 90 FVX Example: Modify path display type

```
query_particle_paths_display()
```

This command will return the global display settings for particle path(s). This command returns a display attributes table.

```
modify(handle, table)
```

This function is used to modify a previously created surface (boundary, computational, iso or coordinate), rake or annotation. The input includes the handle for the object to be modified and a table containing the modifications appropriate for the object. No explicit

output is provided. For the `table` argument for this command, please refer to tables for the surface, rake or annotation object that you want to modify.

Example:

```
coord_table = {  
    dataset = 1,  
    scalar_func = "Pressure [PLOT3D]",  
    visibility = "on",  
    axis = "X",  
    X_axis = {  
        min = -6,  
        current = -1.1,  
        max = 8,  
    }  
}  
  
--create coordinate surface and assign handle to  
--"coord_handle"  
coord_handle = create_coord(coord_table)  
  
modify(coord_handle, {scalar_func = "Density (Q1)"} )
```



Note: It is not possible to modify the dataset for an existing object. It is also not possible to modify the grid number for a computational surface. The dataset number and grid number are fixed at the time of creation of the object.

```
delete(handle)
```

This function is used to delete a previously created surface, streamline or user defined GUI (created with the `make_panel` command; see [GUI Functions](#)). The input is the handle for the surface, streamline or panel to delete. No explicit output is provided and the handle is invalid afterwards.

Example:

```
--the variable "b" holds the handle returned by the function  
--create_boundary()  
delete(b)
```

FVX Show Min Max Annotation

FVX support includes graphical markers and annotation for **local** Min and Max values for a given scalar on Computational, Coordinate, Iso and/or Boundary surfaces, capturing interactive and RESTART functionality from the Scalar Min/Max option on surface panels. (see [Scalar min/max Sub-panel](#) in **Working with FieldView** Ch. 2, **FieldView** Interface). Three fields, `show_minmax_options` and both `min` and `max` (tables), are included in the input and query tables for Comp, Coord, Iso, and Boundary surfaces. The **FVX** field `show_minmax`, analogous to the *Show min/max* check box on the Surface tab, is off by default. When `show_minmax_options` is on, a points and annotation are displayed for the current Scalar Function, defined by the `scalar_func` field. If `scalar_func` is "none", an error will result (see [Scalar Colormap Specification](#)).

FVX_Show Min/Max			
This table describes FVX support for location and annotation of scalar Min and Max values.			
Field	Data Type	Comments	Default
<code>show_minmax_options</code>	string	"on" or "off"	"off"
<code>font</code>	string	'leemono italic' , 'roman sans serif' , 'lee bold' , 'leemono' , 'italics' , 'leese' , 'script' , 'roman' , 'noto sans regular' , 'lee italic' , 'leemono bold italic' , 'leese bold' , 'noto' , 'lee bold italic' , 'lee' , 'leemono bold'	"lee"
<code>size</code>	number	integer to select font size, range 1 to 100	25
<code>min</code>	table	parameters to control display of min and max locations for current scalar	
<code>show_location</code>	string	"on" or "off"	"on"
<code>show_text</code>	table	"on" or "off"	"on"
<code>color</code>	number or string	"white", "black", number ranging from 1 to 8	"black" or "white"
<code>text</code>	string	Strings and/or escape sequences	"Min: %%SCALAR_FUNC = %%SCALAR_MIN"
<code>max</code>	table	parameters to control display of min and max locations for current scalar	
<code>show_location</code>	string	"on" or "off"	"on"
<code>show_text</code>	table	"on" or "off"	"on"
<code>color</code>	number or string	"white", "black", number ranging from 1 to 8	"black" or "white"
<code>text</code>	string	Strings and/or escape sequences	"Min: %%SCALAR_FUNC = %%SCALAR_MIN"

If the `show_minmax` alone is set "on", all other fields will default as shown. The graphical point marker and corresponding text are **shown** by default for both the minimum and maximum scalar values. To turn either off, set `show_location` to "off", and `text` to nil, respectively.

Also, for field `text`, numerical formatting as described in ["Special Numerical Annotation \(Escape Sequences\)"](#) on page 229 are also allowed, eg. `%%SCALAR_MINF12.6`.

FVX Legends

FVX support includes legends, capturing interactive and RESTART functionality from the Legend tab on surface and rake visualization panels (see [Legend Tab Controls](#) in **Working with FieldView** Ch. 2, **FieldView** Interface). Two fields, `show_legend` and `legend` (a table), are included in the input and query tables for Comp, Coord, Iso, Boundary, Streamlines and Particle Paths. The **FVX** field `show_legend`, analogous to the Show Legend check box on the Legend tab, is off by default. When `show_legend` is on, a legend is displayed for the current Scalar Function, defined by the `scalar_func` field. If `scalar_func` is "none", an error will result (see [Scalar Colormap Specification](#)).

FVX_Legends			
This table describes FVX support for legends, applicable to <code>create_coord</code> , <code>create_boundary</code> , <code>create_comp</code> , <code>create_iso</code> , <code>create_streamline</code> , <code>read_particle_path</code> .			
Field	Data Type	Comments	Default
<code>show_legend</code>	string	"on" or "off"	"off"
<code>legend</code>	table		
<code>type</code>	string	"spectrum" or "contour"	n/a
<code>spectrum</code>	table	parameters to control the spectrum display	
<code>border</code>	string	"on" or "off"	"off"
<code>colorbar</code>	string	"on" or "off"	"on"
<code>horizontal</code>	string	"yes" or "no"	"no"
<code>num_labels</code>	number	integer number corresponding to number of labels in the legend, range is 2 to 52	2
<code>contour</code>	table		
<code>labels_per_line</code>	string	"single" or "multi" contour subtable is ignored if type is "spectrum"	"single"
<code>labels</code>	string	"on" or "off"	"on"
<code>labels_parameters</code>	table		
<code>size</code>	number	integer to select font size, range 1 to 100	10
<code>coloring</code>	number or string	"white", "black", number ranging from 1 to 8, or scalar	
<code>decimal_places</code>	number	integer to select number of significant digits in legend labels, range is 0-6	3
<code>numerical_format</code>	string	"floating_point", "exponential" or "powers_of_ten" (Note: "powers_of_ten" is a valid value only if Log Scale is being used)	"floating_point"
<code>font</code>	string	'leemono italic' , 'roman sans serif' , 'lee bold' , 'leemono' , 'italics' , 'leese' , 'meteorology' , 'math upper case' , 'script' , 'cyrillic' , 'roman' , 'helvetica' , 'greek' , 'math lower case' , 'noto sans regular' , 'lee italic' , 'leemono bold italic' , 'leese bold' , 'noto' , 'lee bold italic' , 'lee' , 'leemono bold'	"lee"
<code>relative_position</code>	table	normalized display coordinates with X and Y between -1 and 1 for upper left corner placement of legend	0.85 for both
<code>scale_height</code>	number	real value used to specify vertical scaling of legend	1
<code>scale_width</code>	number	real value used to specify horizontal scaling of legend	1

annotation	string	“on” or “off”	“on”
annotation_parameters	table		
position	string	“top”, “bottom”, “left”, “right”	“top”
title	table		
size	number	integer to select font size, range 1 to 100	10
color	number or string	“white”, “black”, or, number ranging from 1 to 8	
text	string	content for legend main title	%%SCALAR_FUNC
font	string	'leemono italic' , 'roman sans serif' , 'lee bold' , 'leemono' , 'italics' , 'leese' , 'meteorology' , 'math upper case' , 'script' , 'cyrillic' , 'roman' , 'helvetica' , 'greek' , 'math lower case' , 'noto sans regular' , 'lee italic' , 'leemono bold italic' , 'leese bold' , 'noto' , 'lee bold italic' , 'lee' , 'leemono bold'	“lee”
subtitle	table		
size	number	integer to select font size, range 1 to 100	8
color	number or string	“white”, “black”, or number ranging from 1 to 8	
text	string	content for legend sub-title	%%CURRENT
font	string	'leemono italic' , 'roman sans serif' , 'lee bold' , 'leemono' , 'italics' , 'leese' , 'meteorology' , 'math upper case' , 'script' , 'cyrillic' , 'roman' , 'helvetica' , 'greek' , 'math lower case' , 'noto sans regular' , 'lee italic' , 'leemono bold italic' , 'leese bold' , 'noto' , 'lee bold italic' , 'lee' , 'leemono bold'	“lee”
frame	string	“on” or “off”	“off”
background	string	“on” or “off”	“off”

If the legend type is "spectrum" (default), a spectrum subtable determines whether a colorbar is on (default) or off; whether a border is on or off (default); whether the legend display is horizontal (default no); and the number of labels, num_labels (default 2, range 2-52). For legend type "contour", labels_per_line may be "single" (default) or "multi"; note that the display_type must be contour_lines, so contour legends are only practical for surfaces and not rakes.

The legend position is controlled by the relative_position field, which is specified in normalized display coordinates with X and Y between -1 and 1 (default 0.85 for both). The normalized display coordinates are consistent with those used in RESTART files (legend_xpos & legend_ypos). Note that this on screen position differs from Annotation objects, which are defined in absolute pixel coordinates.

The scaling factors scale_width and scale_height scale the legend width and height, respectively; both default to 1 to correspond to the legend size calculated by **FieldView**. Note that the legend size can also be changed interactively and is affected by most legend attributes, except those related to color. Scaling conditions for legend type spectrum are determined using the following rules:

- With colorbar on, scale_width and scale_height both apply;

- With `colorbar` off, the `horizontal` field determines scaling: if set to `no`, `scale_height` applies and `scale_width` is ignored; if set to `yes`, `scale_width` applies and `scale_height` is ignored.

For legend type `contour`, `scale_height` applies but `scale_width` is ignored.

If the `labels` field is on (default), a `labels_parameters` subtable determines whether coloring is scalar, white (default), black or a number 1-8 (see [Geometric Color and Scalar Colormap Specification](#)) and which font is used (see table above) as well as the size (default 10, range 1-100), `numerical_format` (default `floating_point`; or `exponential`) and number of `decimal_places` (default 3, range 0-6) of the numerical labels.

If the `annotation` field is on (default), an `annotation_parameters` subtable allows specification of a title and/or subtitle. The annotation position may be set to the `top` (default), `left`, `right` or `bottom` of the legend. Default text for title is `%%SCALAR_FUNC`. Default subtitle text is `%%CURRENT` for Comp, Coord and Iso. An empty string can also be used to delete the title or subtitle from the legend. Default size for title is 10, for subtitle is 8, range 1-100 for both. For title and subtitle, color may be white (default when the background color is not white), black (default when the background color is white) or a number 1-8 (see [Geometric Color and Scalar Colormap Specification](#)).

The frame surrounding the legend and the background behind it may be turned on or off (default).

When a complete RESTART is saved interactively, the Guide **FVX** file will contain the full syntax for any legends displayed with any of the supported visualization objects. An example listing follows below:

```
create_coord(
    { show_legend = "on",
      legend = {
        type = "spectrum",

        relative_position = {
          0.85,
          0.85,
        }, -- relative_position

        scale_height = 1,
        scale_width = 1,

        frame = "off",
        background = "off",

        spectrum = {
```

```

        border = "off",
        colorbar = "on",
        horizontal = "no",
        num_labels = 2,
    }, -- spectrum

    labels = "on",
    labels_parameters = {
        size = 10,
        coloring = "white",
        decimal_places = 3,
        numerical_format = "floating_point",
        font = "lee",
    }, -- labels_parameters

    annotation = "on",
    annotation_parameters = {
        position = "top",
        title = {
            size = 10,
            color = "white",
            text = "%%SCALAR_FUNC",
            font = "lee",
        }, -- title
        subtitle = {
            size = 8,
            color = "white",
            text = "%%CURRENT",
            font = "lee",
        }, -- subtitle
    }, -- annotation_parameters

}, -- legend

display_type = "constant_shading",
contours = "none",
scalar_range = {
    min = 355.9788818359375,
    max = 1907.152587890625,
}, -- scalar_range
dataset = 1,
scalar_func = "temperature",
axis = "X",
show_mesh = "off",
X_axis = {
    current = -0.01,

```

```

        }, -- X_axis
    }
) -- coord_surfs[1]

```

FVX Support to Return Object Handles

Object handles bridge the gap between **FVX** and script commands, restarts and interactive use of the **FieldView** GUI. In order for a surface, rake or annotation to be accessible, a handle for that object must be created. The **FVX** commands `get_current_object_handle` and `get_all_object_handles` perform this task.

```
my_handle = get_current_object_handle()
```

This retrieves the handle of the current object on the current dataset.

If the current dataset has no current object, `get_current_object_handle()` returns `nil`.

```
handles_table = get_all_object_handles(dataset_number)
```

where

```

handles_table = {
    boundary_handles = {},
    comp_handles = {},
    coord_handles = {},
    iso_handles = {},
    streamline_handles = {},
    particle_path_handles = {},
    text_handles = {},
    arrow_handles = {}
}

```

If `dataset_number` has no objects, `get_all_object_handles()` returns `nil`.

For each of the subtables, if there are no objects of that type on the dataset, the subtable will not appear in the output table (ie, the subtable is `nil`).

Note that the table of handles returned is a "snapshot" of the state of **FieldView** at the moment the command is executed.

For each non-`nil` subtable, indexing starts at 1.

Except for the `comp_handles`, `text_handles`, and `arrow_handles` subtables, the index would also correspond to the "number" of the object on its panel in the GUI. For example, `handles.coord_handles[2]` would be the **FVX** handle of the 2nd surface on the dataset.

Computational Surface Handles

Since computational surfaces belong to grids on the dataset, the subtable, `comp_handles`, will be a 2D table, indexed first by grid, and then by surface on the grid.

For example,

`comp_handles[1]` would be a table of handles of surfaces on Grid 1.

`comp_handles[1][1]` would be the handle of the 1st surface on Grid 1.

`comp_handles[1][2]` would be the handle of the 2nd surface on Grid 1.

`comp_handles[2][1]` would be the handle of the 1st surface on Grid 2.

etc.

If there are no surfaces on Grid `n`, `comp_handles[n]` will be `nil`.

Text and Arrow Handles

Although text and arrows are created in **FVX** with separate commands (see [Creation and Modification of Post-Processing Objects](#)), they are created interactively by the same visualization panel (Annotation) in the **FieldView** GUI.

Therefore, the indices into the `text_handles` and `arrow_handles` subtables are not guaranteed to match the number of that object on the Annotation panel, where text and arrows can be intermixed, according to the order in which they were created.

But, `text_handles[n]` would be the `n`th text object and `arrow_handles[n]` would be the `n`th arrow object found on that panel.

Known Limitations and changes in behavior

In order to avoid having a `nil` handle returned, a surface should be current. If there is no current surface, as would be the case for when a visualization panel is closed, then no handle is returned.

Geometric Color and Scalar Colormap Specification

A field to specify geometric color (`geometric_color`) is available for input tables to the creation commands outlined in [Common input fields for surfaces, rakes & annotation](#). The geometric color setting will be used when the `scalar_func` table entry is either not set or set to "none". For Annotation objects, `geometric_color` is the only coloring option.

The color specifications for "white" and "black" are the only colors which will be referenced by their actual names. Since any of the other 8 colors can be changed using the color mixer, numbers instead of names are assigned to reference them. These color specification numbers are designed to map onto the current color mixer chip. The numbers ranging from 1 to 4 refer to the colors on the top row (excluding "white"), and the numbers ranging from 5 to 8 refer to colors defined on the bottom row (excluding "black"). The color "white" can be considered to be color number 0, but cannot be specified in this way. In the same respect, the color "black" can be considered to be color number 9.

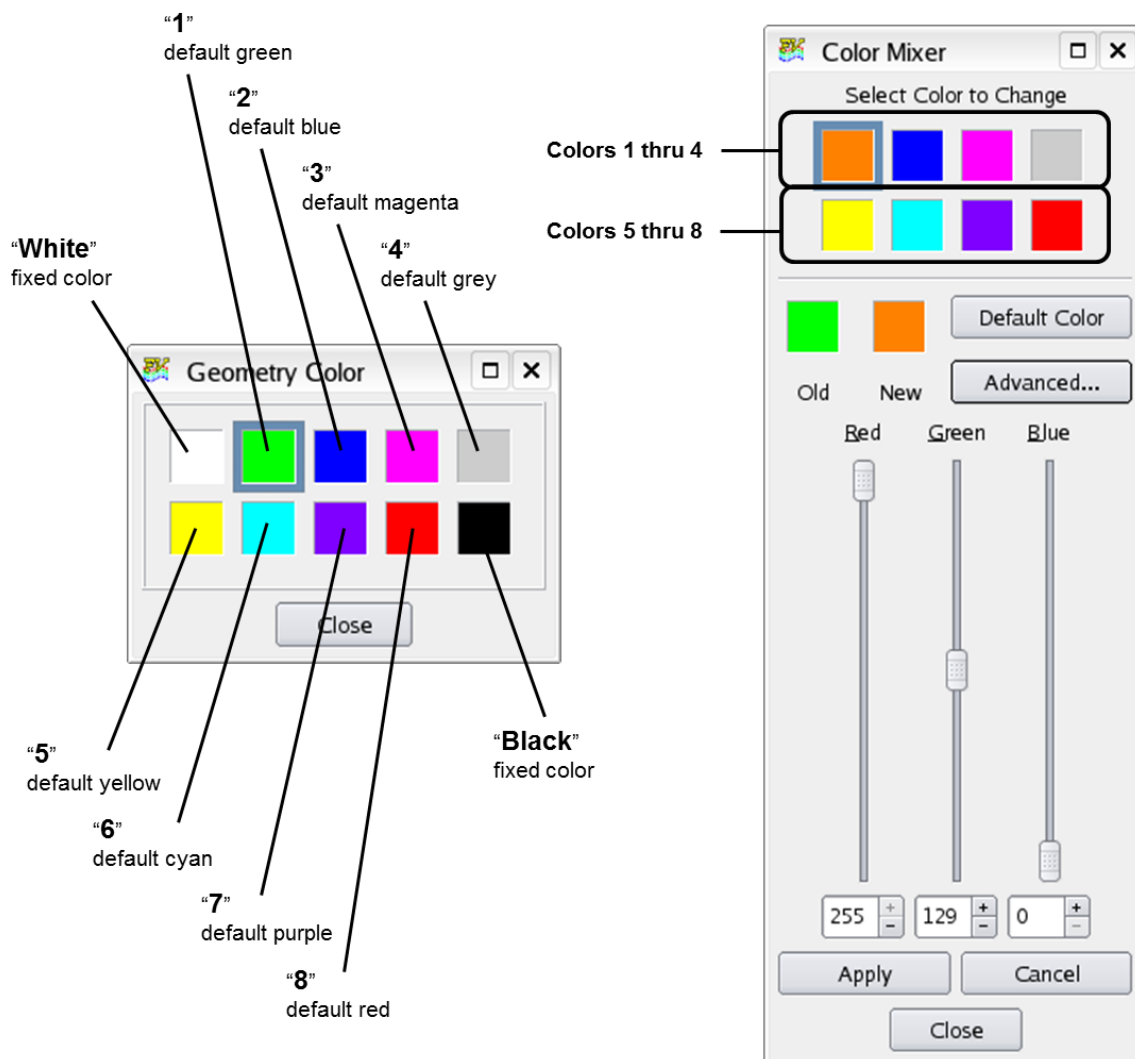


Figure 91 Geometric Color Specification

When a query is performed on a surface or rake, the field for the geometric color will be returned, showing one of either the strings "white" or "black" or, a number ranging from 1 to 8.

Example: Create a geometric colored boundary surface.

```
boundary_surface1 = create_boundary({
  types = {"body", "wing"},
  display_type = "smooth_shading",
  geometric_color = "white"})
```

```

boundary_surface2 = create_boundary({
    scalar_func = "none",
    geometric_color = 8,          -- 8 = red (default color)
    threshold_func = "X",
    threshold_range = {min = 2.4, max = 2.5},
    types = {"tail"}
})

```

```
set_colortable(color_table_specification)
```

This function lets you customize the standard palette of 8 colors, following the same interactive functionality provided with the color mixer.

color_table_specification			
The table is an input argument for set_colortable command.			
Field	Data Type	Comments	Default
1	table		
red	number	Valid range is from 0 to 255	0
green	number	Valid range is from 0 to 255	255
blue	number	Valid range is from 0 to 255	0
name	string	optional	"green"
2	table		
red	number	Valid range is from 0 to 255	0
green	number	Valid range is from 0 to 255	0
blue	number	Valid range is from 0 to 255	255
name	string	optional	"blue"
3	table		
red	number	Valid range is from 0 to 255	255
green	number	Valid range is from 0 to 255	0
blue	number	Valid range is from 0 to 255	255
name	string	optional	"magenta"
4	table		
red	number	Valid range is from 0 to 255	204
green	number	Valid range is from 0 to 255	204
blue	number	Valid range is from 0 to 255	204
name	string	optional	"gray"
5	table		
red	number	Valid range is from 0 to 255	255
green	number	Valid range is from 0 to 255	255
blue	number	Valid range is from 0 to 255	0
name	string	optional	"yellow"
6	table		
red	number	Valid range is from 0 to 255	0
green	number	Valid range is from 0 to 255	255
blue	number	Valid range is from 0 to 255	255
name	string	optional	"cyan"

7	table		
red	number	Valid range is from 0 to 255	127
green	number	Valid range is from 0 to 255	0
blue	number	Valid range is from 0 to 255	255
name	string	optional	“purple”
8	table		
red	number	Valid range is from 0 to 255	255
green	number	Valid range is from 0 to 255	0
blue	number	Valid range is from 0 to 255	0
name	string	optional	“red”

Just one or all of the colors can be re-specified to your custom settings. Note: Definitions for all 8 colors must be supplied in order to change the color table with this command.

```
query_colortable(color_table_specification)
```

This function will return a table of 8 color specification tables, as defined above with the previous command. The practical utility of this command is that it provides a simple way to generate a full table which can then be modified and sent back to the `set_colortable` command described previously. Any changes made to the color palette, either interactively, or by reading a COLOR RESTART prior to this query will be correctly returned.

```
query_default_colortable()
```

This function will return the original 8 color settings in table form. The returned table can then be passed to `set_colortable()` to restore the color palette to its original default settings.

```
set_color(color_specification)
```

This function can be used to change a single color within the color palette. It differs from the `set_colortable` command which is used to set ALL of the colors in the color palette.

color_specification			
The table is an input argument for <code>set_color</code> command.			
Field	Data Type	Comments	Default
1..8	number	number matches order in colortable	
	table	color specification table	
red	number	Valid range is from 0 to 255	
green	number	Valid range is from 0 to 255	
blue	number	Valid range is from 0 to 255	
name	string	optional	

```
get_default_color(color_number)
```

This function is used to obtain the original default color specification table for any one of the 8 colors in the color palette. The `color_number` corresponds to an integer which represents a single entry in the color table specification table.

Example:

Change the original entry for “green” in the default color palette to “brown” as illustrated in the Color Mixer panel, shown previously.

```

brown = { red = 164, green = 129, blue = 0 }

-- Method 1
set_color (1, brown)

-- Method 2
geom_colors = query_colortable()
geom_colors[1] = brown
set_colortable(geom_colors)

-- Method 3
-- replace "green" chip with "brown"
geom_colors = query_colortable()
geom_colors[1] = {red=164, green=129, blue=0, name="brown"}
set_colortable(geom_colors)

```

Note that the **FVX** script would continue to specify this new color by the number 1, eg: `geometric_color = 1`, and not by the name, “brown”.

Scalar Colormap Specification

If the `scalar_func` field is present, fields to specify colormap (`scalar_colormap`) and range (`scalar_range`) will be present in the query table and available for input tables to the creation commands outlined in [Common input fields for surfaces, rakes & annotation](#). Support for the “Local” check button on the Colormap GUI is also provided. This will let you automatically scale the colormap to the local range for the surface or rake being created (or modified). Support for the “Log Scale” check button on the Colormap GUI is also provided (see [Log Scale for Colormaps](#) in [Working with Field-View](#) for information).

```

scalar_colormap = {
  name = "spectrum" | "nasa-1" | "nasa-2" | "gray scale" |
  "color striped" | "black & white" | "striped" | "zebra" |
  "achromatic vision 1" | "achromatic vision 2" |
  "banded blue to red dark" | "banded blue to red light" |
  "banded grayscale" | "big difference" | "bio spectrum 1" |
  "bio spectrum 2" | "bloodflow doppler" | "blue chrome" |
  "camouflage" | "ccm blue red" | "ccm cool warm" |
  "ccm high contrast" | "ccm spectrum" | "cd spectrum" |

```

```

"cd striped" | "chrome" | "cold" | "dark radiation" |
"dark spectrum" | "dark green gradient" |
"flame transition" | "gold" | "high contrast" | "hot" |
"hot to cold diff" | "indigo flame" | "inferno" |
"leaf color" | "magma" | "plasma" | "radiation colors" |
"red to blue diff" | "red to purple diff" | "relief map" |
"relief map land" | "relief map ocean" | "simple flux" |
"small difference" | "spectrum diff gray" |
"spectrum diff white" | "steel blue" |
"teal black gradient" | "viridis"
-- a string that is the filename (including path) of a
-- colormap file.
[default: "spectrum"],
invert = "on" | "off" [default: "off"],
filled_contour = "on" | "off" [default: "off"],
log_scale = "on" | "off" [default: "off"]
}

```

If `scalar_colormap.name` does not match a defined colormap name, **FieldView** assumes the string represents a filename, including path, of a colormap file.

The scalar range field for any surface or rake contains the following:

```

scalar_range = {

-- the global range for a given scalar function
abs_max = number,
abs_min = number,

-- the local range for a given scalar function
local_max = number,
local_min = number,

-- user specified range to control the colormap for a scalar
max = number,
min = number,

-- simple setting to toggle local scaling of the colormap
use_local = "on/off"

}

```

By setting the `min` and `max` fields within the `scalar_range` input table, you can directly specify the range for the colormap. The `use_local` field setting can also have an impact on the scaling of the colormap.

- If "use_local" is set to "on" and the min and/or max fields are also specified, then the min and max fields will be used, overriding the "use_local" setting.
- If "use_local" is set to "on" and neither of the min or max fields are specified, then the colormap scaling will be based on the local min/max range of the scalar function. This is the same as turning the "local" button on.
- If "use_local" is set to "off", or is not present, and the min and/or max fields are specified, then the min and max fields will be used to set the colormap range.
- Following a create or modify command, if "use_local" is not present in the scalar_range input table, it is considered to be set to "off".

The min and max values in the scalar_range input table may be set to values that exceed the bounds defined by abs_min and abs_max, under the condition that "use_local" is set to "off", or is not present. Additionally, the min and max values in the scalar_range input table may be set to values that exceed the bounds defined by local_min and local_max, under the condition that "use_local" is set to "on".

Example: Create a theta coordinate surface, and modify the scalar range, temperature.

```
my_coord = create_coord({
    dataset = 1,
    axis = "T",
    scalar_func = "temperature",
    scalar_range = { min = -72, max = 1500 },
    display_type = "smooth_shading"
})

-- override previous specification for scalar range
modify(my_coord, { scalar_range = { use_local = "on" } })
```

Example: Create a series of computational surfaces, with the local scalar range turned on.

```
my_comps = {}  
  
for i=1,8 do  
  
    my_comps[i] = create_comp({  
        grid = 1,  
        axis = "K",  
        K_axis = {  
            min = 1,  
            current = 1,  
            max = 2 },  
        J_axis = {  
            min = 1+(i-1)*8,  
            max = 1+i*8 },  
        scalar_func = "Y",  
        scalar_range =  
            { use_local = "on"},  
        display_type = "constant_shading",  
    })  
  
end
```

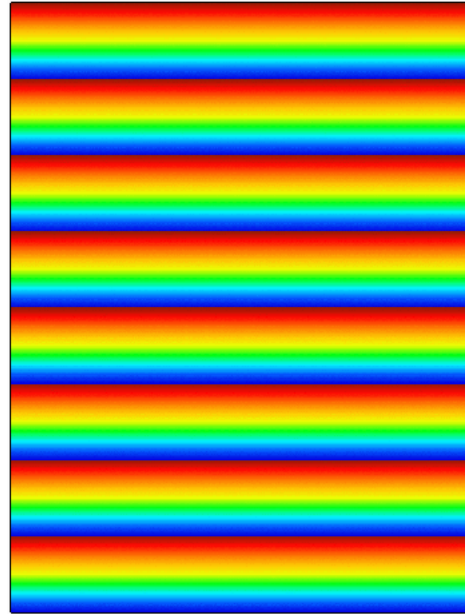


Figure 92 FVX Example: Controlling local scalar range

Example: Create a series of coordinate surfaces with scalar contours showing a range of transparency values.

```

my_coords = {}

for i=1,8 do

    my_coords[i] = create_coord({
        axis = "Z",
        Z_axis = { current = 1 },
        scalar_func = "X",
        contours = "scalar",
        number_of_contours = 9,
        line_type = "thick",
        display_type = "smooth_shading",
        transparency = (i-1)/8,
        threshold_func = "Y",
        threshold_range = {
            min = (i-1)*8,
            max = i*8
        }
    })
end

```

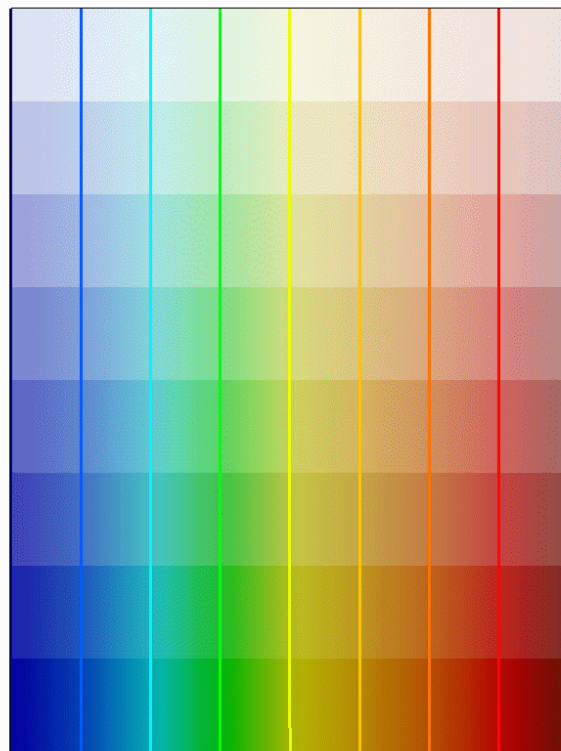


Figure 93 FVX Example: Controlling local scalar range

`xcolor_defines.fvx` *Utility*

A utility for handling colors, `xcolor_defines.fvx`, is located on the **FieldView** DVD in the `fvx_and_restarts` subdirectory. This utility contains a table with definitions for all of the standard X Colors and is compatible with the `set_color` and `set_color_table` **FVX** commands. There are also two **FVX** functions included in this utility: `set_xcolor` and `match_colors`. The former function can be used to modify an existing color definition with one of the X Colors. The latter function can be used to query the X Color table and return a subset of the colors matching a selection criterion.

Example: Change the coordinate surfaces from the preceding example to be geometric colored, using the first eight X colors that have the word 'blue' in their name. This example will show how to use the `xcolor_defines.fvx` functions.

```

dofile("xcolor_defines.fvx")

just_blue = match_colors("blue")
my_colors = query_colortable()

for i = 1,8 do
    my_colors[i] = just_blue[i]
end
set_colortable(my_colors)

my_color_names = {}

for i = 1,8 do
    modify(my_coords[i],
        { geometric_color = i })

    my_color_names[i] = create_text({
        text = my_colors[i].name,
        font = "helvetica",
        size = 16,
        position = { 60, 750 - 95*(i-1) },
        geometric_color = "white",
    })
end

```

**Figure 94 FVX Example: Controlling local scalar range**

Vector Options

FVX support is included for curved and skip vector specifications. A field, `vector_options`, of type `table`, is included in the input and query tables for `Comp`, `Coord`, `Iso` and `Boundary` surfaces. This field will always be allowed for `create()` and `modify()` but will not be returned by `query()` if `vector_func` is "none".

vector_options			
This is a subtable of <code>comp_table</code> , <code>coord_table</code> , <code>iso_table</code> and <code>boundary_table</code> .			
Field	Data Type	Comments	Default
<code>shaft_type</code>	string	"straight" or "curved" Only Coord surfaces can set <code>shaft_type</code> to "curved", otherwise it will be an error.	"straight"
<code>head_type</code>	string	"2D" or "3D"	"2D"
<code>head_scaling</code>	string	"on" or "off"	"off"

head_scaling_value	number	float; range: greater than or equal to 0.0 head_scaling_value will be present in query() table only if head_scaling is "on"	1.0
type	string	"total", "yz", "xz", "xy" or "projected" The value "projected" is not allowed on Comp surfaces; it will be ignored if specified, silently reverting to its previous value. The value "projected" is not allowed if shaft_type is "curved". If a Coord surface sets type to "projected" when its current or input shaft_type is "curved", it will silently revert to "total".	"total"
skip	string	"on" or "off"	"off"
skip_value	number	float greater than or equal to 0.0; range: 0 to 1, inclusive, rounded to nearest of: 0.0, 0.125, 0.250, 0.375, .5, .625, .75, .875, 1. An input value of 0.0 is the same as setting skip to "off". skip_value will be present in query() table only if skip is "on".	
uniform_sampling	string	"on" or "off" Only Coord surfaces can set uniform_sampling to "on", otherwise it will be an error.	"off"
number_of_samples	table	For Coord surfaces, number_of_samples will be present in the query() table only if uniform_sampling is "on". For Comp, Iso & Boundary surfaces, number_of_samples will be ignored on create()/modify() and will not be present in the query() table. Fields x, y and z are expected for Cartesian; r, t and z or x for non-Cartesian.	
x or r	number	range 1-999, inclusive	10
y or t	number	range 1-999, inclusive	10
z or x	number	range 1-999, inclusive	10
vector_scale	number	floating point number Range: greater than 0.0 and less than or equal to 1e+10 NOTE: for shaft_type "straight"	1.0
time_limit	number	floating point number The default value for time_limit is calculated. Range: greater than 0.0 and less than or equal to 1e+10 NOTE: for shaft_type "curved"	
display_type	string	"complete", "filament" or "growing" NOTE: for shaft_type "curved"	"complete"

Annotation

```
create_text(text_description)
```

This function lets you create a text string with control over its font, color, size and position within the graphics window. **FVX** `modify` and `query` calls are supported for this data structure.

text_description			
This table is an input argument for <code>create_text</code> command.			
Field	Data Type	Comments	Default
text	string	Specify the content of the text string	
visibility	string	"on" or "off"	"on"
geometric_color	number or string	"white", "black", or, number ranging from 1 to 8	
font	string	"lee", "lee bold", "lee italic", "lee bold italic", "leemono", "leemono bold", "leemono italic", "leemono bold italic", "leese", "leese bold", "noto", "roman", "roman sans serif", "italics", "script", "helvetica", "math lower case", "math upper case", "greek", "cyrillic", "meteorology"	"roman"
size	number	number ranging from 1 to 100	10
direction	string	"horizontal", "vertical"	"horizontal"
position	string	"left", "center", "right", "top", "middle", "bottom"	
	table		
	number	X position in pixels (0 is left side of window)	
	number	Y position in pixels (0 is top of window)	

Example: Create titles to show the solution time and time step for a transient dataset. In this case, we'll use the **escape sequence** to show the Solution Time and Time Step.

```
create_text({
    text = "Soluton Time: %%T1",
    font = "roman sans serif",
    size = 10,
    position = { 12, 625 },
    geometric_color = "white"
})
```

```
create_text({
    text = "Time Step: %%N1",
    font = "roman sans serif",
    size = 10,
    position = { 300, 625 },
    geometric_color = "white"
})
```

A simple illustration of the legacy fonts available is shown in the following figure:

Roman	AaBbCcDdEeFfGg..
Roman San Serif	AaBbCcDdEeFfGg..
Roman Italics	<i>AaBbCcDdEeFfGg..</i>
Helvetica	AaBbCcDdEeFfGg..
Script	<i>AaBbCcDdEeFfGg..</i>
Math Lower Case	a\$b√c√dce∪f⊃gn..
Math Upper Case	A\$B√C√DCE∪F⊃GN..
Greek	ΑαΒβΓγΔδΕεΖζΗη..
Cyrillic	АЯБаВбГвДгЕдЁе..
Meteorology	AaBbCcDdEeFfGg..

Figure 95 Legacy Font illustration

To consistently and more easily place text within the graphics window, please note the following.

If the text is horizontal, the left side of the text rectangle passes through the X position. The Y position specifies the baseline of the text. Note that descenders in letters such as "g", "j", "p", "q", and "y" lie below the baseline. Therefore specifying a Y position of 0 for a horizontal string means that only the descenders would be visible in the graphics window. Specifying a `position="top"` would work better in this case. Similarly, specifying an X position equal to the maximum width of the window would result in the string rendered off screen. Specifying a `position="right"` would work better in this case.

If the text is vertical, the Y position lies on the capline of the first character in the string. The X position lies midway between the left and right sides of the text rectangle. Therefore, specifying a Y position equal to the maximum height of the window would result in the string rendered off screen. Specifying `position="bottom"` would work better in this case. Similarly, specifying an X position equal to 0 or max screen width would result in only half the string being visible. Specifying `position="left"` or `position="right"` would work better.

An illustration of text layout using the various specifications is shown in the following figure:

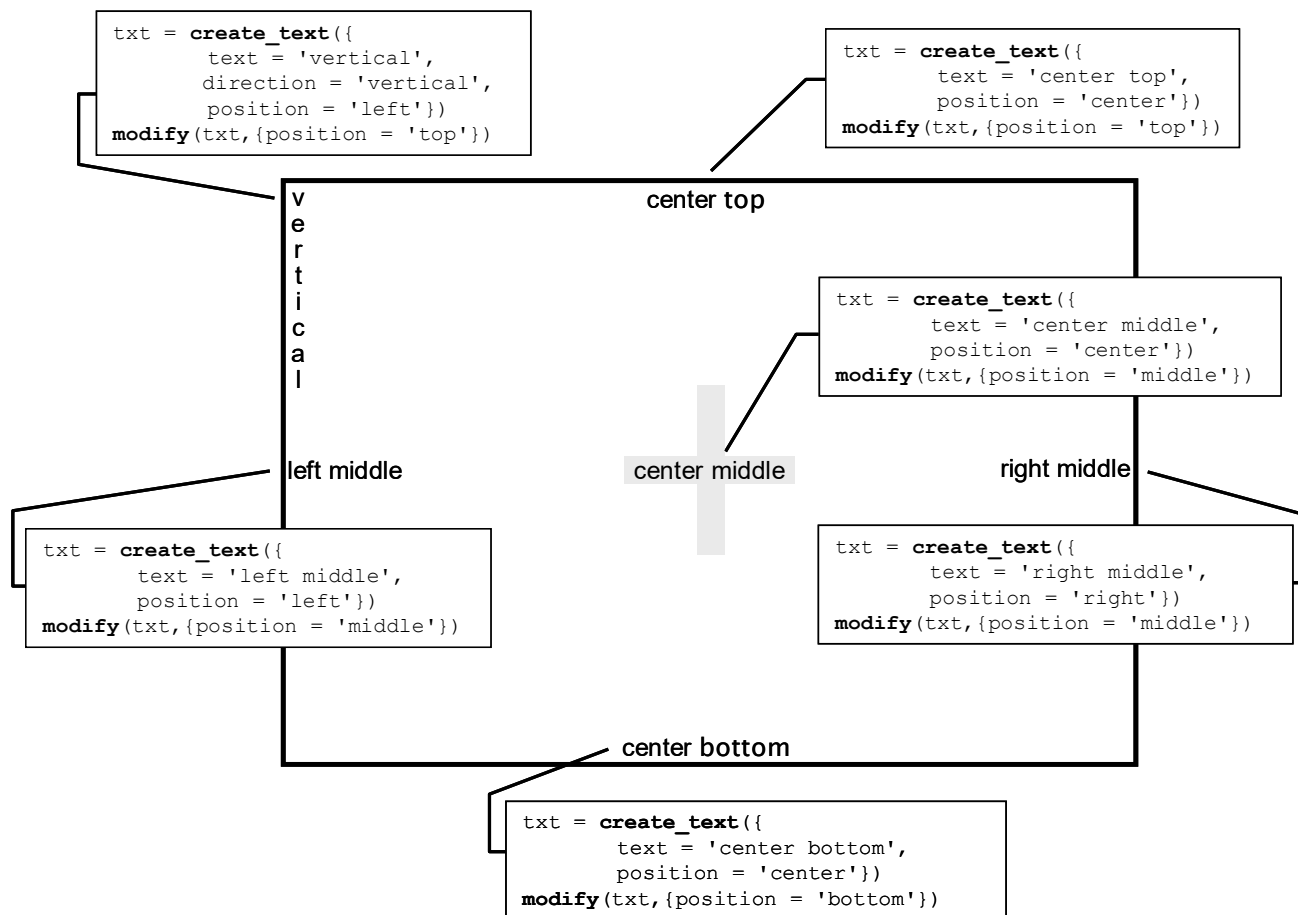


Figure 96 FVX Example: Text placement

`create_arrow(arrow_description)`

This function lets you create an arrow with control over its starting and end point, color, size and position within the graphics window. Alternate specifications are available to choose the arrow start point and its angle. The **FVX** `modify` and `query` calls are supported for this data structure.

arrow_description This table is an input argument for <code>create_arrow</code> command.			
Field	Data Type	Comments	Default
visibility	string	"on" or "off"	"on"
geometric_color	number or string	"white", "black", or, number ranging from 1 to 8	
from	table		
	number	X position in pixels (0 is left side of window)	

	number	Y position in pixels (0 is top of window)	
to	table		
	number	X position in pixels (0 is left side of window)	
	number	Y position in pixels (0 is top of window)	
angle	number	angle in degrees, 0 corresponds to a horizontal orientation in the graphics window	
length	number	length of arrow in pixels	
width	number	width of arrow in pixels	

Example: Create a series of arrows of different colors, angles and lengths.

```

arrow = {}

for i=1,19 do
  arrow_color = mod(i,8)
  if mod(i,8) == 0 then
    arrow_color = 8
  else
    arrow_color = mod(i,8)
  end

  arrow[i] = create_arrow ({
    geometric_color = arrow_color ,
    angle = (i-1)*10,
    length = 200 + (i-1)*8,
    width = i,
    from = { 400, 350 }
  })
end

```

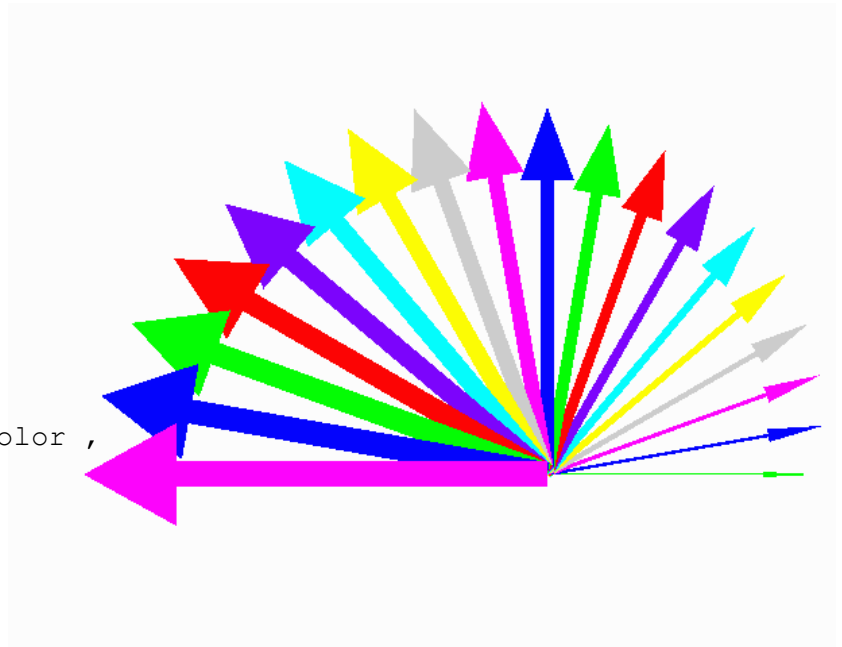


Figure 97 FVX Example: Drawing arrows

Quantify and Query

```
integrate_all(scalar_function [, dataset_number])
```

This function is used to integrate a current scalar function across all visible surfaces in the entire dataset. The input is the `scalar_function` and the optional `dataset_number`. If `dataset_number` is not provided, it defaults to the current dataset. The output is the `integration_result` table.

integration_result			
This table contains the data returned by the integration function.			
Field	Data Type	Comments	Default
integral_type	string	integral_type	

scalar_function	string	scalar_name	
area	number	area of the integral	
sum	number	sum of scalar in area	
average	number	sum divided by area	

Example:

```

scalar_function = "Temperature [PLOT3D]"
-- dataset no. not provided
result_a = integrate_all(scalar_function)
-- dataset no. is provided
result_b = integrate_all(scalar_function, 3)

```

```
integrate_surface(surface_handle)
```

This function is used to integrate the current scalar function across a previously defined surface. The input is the handle for the surface to be integrated. The output is the `integration_result` table. The scalar function should have been loaded in a prior step, i.e., during surface creation or modification. Otherwise, the first function in the list of scalar functions is automatically selected.

integration_result			
This table contains the data returned by the integration function.			
Field	Data Type	Comments	Default
integral_type	string	integral_type	
surface	string	surface_type	
scalar_function	string	scalar_name	
area	number	area of the integral	
sum	number	sum of scalar in area	
average	number	sum divided by area	
has_surface_normals	string	"yes" or "no"	
fields below exist if has_surface_normals = "yes"			
vector_function	string	"none" or name of function	
sum_Nx	number	not present when integrating a coordinate surface	
sum_Ny	number	not present when integrating a coordinate surface	
sum_Nz	number	not present when integrating a coordinate surface	
sum_V_dot_N	number		

Example:

```

--define iso surface
iso_table = {
    dataset = 4,
    mode = "point_and_normal",
    pt1 = {1,0,0},
    pt2 = {1,1,0},
    iso_value = {
        min = -3,

```

```

        current = -1,
        max = 0,
    },
    scalar_func = "Cp [PLOT3D]",
    vector_func = "Velocity Vectors [PLOT3D]",
    threshold_func = "Entropy [PLOT3D]",
    threshold_range = {
        min = -0.03,
        max = 0.32,
    },
    visibility = "on",
}

--create iso surface
iso_handle = create_iso(iso_table)

--integrate scalar function over iso surface and put results
--in table result
result = integrate_surface(iso_handle)

```

`integrate_partial_surface(surface_handle, point_table, selection_tolerance)`

This command is used to integrate over a portion of a coordinate surface or iso-surface when the surface consists of more than one piece. The integration is performed on the connected portion of the surface that touches the point specified by `point_table`. Adjacent polygons of a surface are considered connected if they share at least one node, or have two nodes that are closer than `selection_tolerance`. The integration results are returned in the form of an `integration_result` table.

point_table This is a table with numbers at index values 1, 2 and 3.			
Field	Data Type	Comments	Default
[1]	number	'x' coordinate	
[2]	number	'y' coordinate	
[3]	number	'z' coordinate	

`selection_tolerance` is a real number. Two pieces of the surface are assumed to be connected to each other if they have nodes that are closer to each other than this number.

See `integrate_surface` command above for detail on the `integration_result` table.

Example:

```
point_table={1.1,2,3.}
```

```
integrate_partial_surface(iso_handle,point_table,1e-05)
```



Note: A `selection_tolerance` of zero will return a single polygon on what appears to be a contiguous surface. It is recommended that the default value of `1e-5` be used.

```
get_all_boundary_types(dataset_number or surface_handle)
```

This function is used to get a list of boundary types available for a given dataset. The input is a one-based `dataset_number` or a `surface_handle`. A single argument is required. The output is a table with all the boundary types available for the specified dataset or the dataset of the surface.

Example:

```
output_table = get_all_boundary_types(<dataset|handle>)
```

```
get_scalar_functions(dataset_number)
```

Returns a table of (volume) scalar functions for the dataset whose number is the input argument.

Example:

```
output_table = get_scalar_functions(dataset_no)
```

```
get_vector_functions(dataset_number)
```

Returns a table of (volume) vector functions for the dataset whose number is the input argument.

Example:

```
output_table = get_vector_functions(dataset_no)
```

```
get_surface_scalar_functions(dataset_number)
```

Returns a table of surface-based scalar functions for the dataset whose number is the input argument.

Example:

```
output_table = get_surface_scalar_functions(dataset_no)
```

```
get_surface_vector_functions(dataset_number)
```

Returns a table of surface-based vector functions for the dataset whose number is the input argument.

Example:

```
output_table = get_surface_vector_functions(dataset_no)
```

```
match_one_entry(tbl, substr)
```

Match `substr` in table `tbl` and return the full string of the matching entry in `tbl`. Returns a nil string and error message if no matches or more than one match. Matching is case-insensitive with no regular expressions allowed. The value of `err_str` is nil when there are no errors. Otherwise, it is assigned a text string consisting of the error message. This function will operate on numerically indexed tables only, and not tables indexed by other data types.

Syntax:

```
single_match = match_one_entry(tbl, substr)
```

```
match_multiple_entries(tbl, substr1 [, substr2...substrN] )
```

Match one or more `substrs` in table `tbl` and return a table of the matching entries in table. Returns a nil table and error message if no matches. Matching is case-insensitive with no regular expressions allowed. The value of `err_str` is nil when there are no errors. Otherwise, it is assigned a text string consisting of the error message. This function will operate on numerically indexed tables only, and not tables indexed by other data types.

Syntax:

```
multi_match = {}
multi_match = match_multiple_entries(tbl, "substr1" [, "substr2"... "substrN"])
```

```
query(handle)
```

This function is used to query for information on a previously created surface or streamline. The input is the handle for the surface or the streamline. The output is a table whose content will depend on the type of the surface. Refer to [Creation and Modification of Post-Processing Objects](#) for the table definition of each surface type.

Example:

```
handle = create_comp(comp_table)
--returned table is assigned to out_table
out_table = query(handle)
--debugging dump showing the contents of out_table
dumpall(out_table)
```

Thus, for surfaces `query()` returns values that would be in the input table. For streamlines,

```
query_table = query(rake_handle)
```

returns a table similar to the [streamline_table](#).

In addition, `query(rake_handle)` returns the following fields:

- `number_of_seeds`
- `duration` – a table of real numbers. `duration[i]` is the duration (length) of the streamline for seed `i`.
- `duration_longest_path (on rake)` – real number. Maximum of `duration[i]`.

These fields are read-only; if present in an input table to `create_streamline()` or `modify()`, they will be ignored.

The seeding subtable of the `query_table` returned by the command will be of the form:

query from streamline rakes			
This seeding subtable of a table returned by <code>query(rake_handle)</code> command.			
Field	Data Type	Comments	Default
seed_coord	string	"IJK_int", "IJK_real", "XYZ", "RTZ"	none
mode	string	"add"	none
seeds	table	Table of seed points. I, J and K; or X, Y, Z; or R, T, Z are in their respective tables within this table.	none
i	table	Table of 'I'	none
j	table	Table of 'J'	none
k	table	Table of 'K'	none
x, or r	table	Table of 'X', or 'R' coordinates	none
y, or t	table	Table of 'Y', or 'T' coordinates	none
z	table	Table of 'Z' coordinates	none
grid	table	Table of grid numbers. One grid number for each seed.	Grid 1 in the current dataset.

`query(rake_handle)` will not return i, j, k seed subtables if the dataset is unstructured.

Example:

```
streamline = create_streamline(streamline_table)
-- returned table is assigned to query_table
query_table=query(streamline)
-- debugging dump showing the contents of query_table
dumpall(query_table)
```

The output from the **FVX** query command, for any surface or rake, will be modified to include the local minimum and maximum range for the scalar, if it has been specified. The returned local minimum and maximum values will be consistent with any thresholding, subsetting and/or dynamic clipping which may have been applied (through **FVX** commands) to the surface or rake in question.

Updated Query Return for FVX Local Min and Max			
This table contains the data returned by a normal query.			
Field	Data Type	Comments	Default
scalar_function	string	scalar_name	
scalar_range	table	table containing absolute and local range for the scalar function	

abs_max	number	global maximum for the scalar function	
abs_min	number	global minimum for the scalar function	
max	number	specified maximum for the scalar function	
min	number	specified minimum for the scalar function	
use_local	string	"on" or "off", depending on use_local setting	
local_max	number	local maximum for the scalar function	
local_min	number	local minimum for the scalar function	

```
probe_current_functions(point [, dataset_number])
```

This function is used to return the values of functions at a given point for a given dataset. The input arguments are the table `point` and the optional `dataset_number`. If `dataset_number` is not provided, it defaults to the current dataset. The output is the table `probe_cf_result`. The coordinate system can be Cartesian or cylindrical.

point This is a table with numbers at index values 1, 2 and 3.			
Field	Data Type	Comments	Default
[1]	number	'x' or 'r' coordinate	
[2]	number	'y' or 'theta' coordinate	
[3]	number	'z' coordinate	

point_cf_result This table contains data that is returned by the function <code>probe_current_functions()</code> .			
Field	Data Type	Comments	Default
point	table	table of numbers for point that is probed	
region	number	region of point (only if regions are defined)	
grid	number	number of the grid that the point belongs to	
IJK	table	table of I, J and K values; only present if the grid is of type 'structured'	
scalar	table	table with the scalar function's information	
func	string	name of the scalar function; nil if no function	
value	number	value of the scalar function; nil if no function	
iso	table	table with the iso function's information	
func	string	name of the iso function; nil if no function	
value	number	value of the iso function; nil if no function	
threshold	table	table with the threshold function's information	
func	string	name of the threshold function; nil if no function	
value	number	value of the threshold function; nil if no function	
vector	table	table with the vector function's information	
func	string	name of the vector function; nil if no function	
value	table	table of x, y and z components of the vector function; nil if no function	
1	number	value of vector component	
2	number	value of vector component	
3	number	value of vector component	

Example:

```
point = {10, 15, 32}
probe_current_functions(point) -- without specifying dataset
probe_current_functions(point, 3) -- with dataset specified
```

```
probe_IJK_current_functions(probe_IJK_input, grid_number [, data-
set_number])
```

This function is specific to structured grids. It returns a point given three fixed axis-value pairs, or for a line of values, two fixed axis-value pairs and one range. The first input argument is the table `probe_IJK_input`. The second input argument is the `grid_number`. The optional third argument is the `dataset_number`. If `dataset_number` is not provided, it defaults to the current dataset. The output is the table `probe_IJK_result`.

probe_IJK_input			
This table is the input argument for the function <code>probe_IJK_current_functions()</code> .			
Field	Data Type	Comments	Default
I	number or table		
min	number	If datatype of I is table, this field exists.	
max	number	If datatype of I is a table, this field exists.	
J	number or table		
min	number	If datatype of J is table, this field exists.	
max	number	If datatype of J is a table, this field exists.	
K	number or table		
min	number	If datatype of K is a table, this field exists.	
max	number	If datatype of K is a table, this field exists.	

probe_IJK_result			
This table is returned by the function <code>probe_IJK_current_functions()</code> .			
Field	Data Type	Comments	Default
n	number	number of results returned	
[n]	table	table of values of the nth result	
IJK	table	table of I, J and K values; only present if the grid is of type 'structured'	
[1]	number	value of index I	
[2]	number	value of index J	
[3]	number	value of index K	
point	table	table of x, y and z coordinates of point 'n'	
[1]	number	value of x-coordinate of point 'n'	
[2]	number	value of y-coordinate of point 'n'	
[3]	number	value of z-coordinate of point 'n'	
grid	number	number of the grid that the point belongs to	

region	number	region at point (only if regions are defined)	
scalar	table	value of scalar at point 'n'	
func	string	name of the scalar function; nil if no function	
value	number	value of the scalar function; nil if no function	
iso	table	value of iso function at point 'n'	
func	string	name of the iso function; nil if no function	
value	number	value of the iso function; nil if no function	
threshold	number	value of threshold function at point 'n'	
func	string	name of the threshold function; nil if no function	
value	number	value of the threshold function; nil if no function	
vector	table	table of vector function at point 'n'	
func	string	name of the vector function; nil if no function	
value	table	table of vector function components	
[1]	number	value of the first component of the vector function	
[2]	number	value of the second component of the vector function	
[3]	number	value of the third component of the vector function	

Example:

```

probe_IJK_input = {
    I=10,
    J=10,
    K=14
}
-- call function with dataset no. specified
probe_IJK_result = probe_IJK_current_functions(probe_IJK_in-
put, 1, 2)

-- print scalar function name
-- Note: Although we have just one table, we still need to
-- specify its number, [1].
print(probe_IJK_result[1].scalar.func)

-- print scalar function value
print(probe_IJK_result[1].scalar.value)

```

Example:

```

probe_IJK_input = {
    I = 10,
    J = 10,
    K = {
        min=10,
        max=14
    }
}
-- function is called with dataset not specified. Also,
-- more points are returned since a range of K values is

```

```
-- provided as input
probe_IJK_result = probe_IJK_current_functions(probe_IJK_in-
put, 1)

-- dump all five tables (for K=10,11,12,13,14)
dumpall(probe_IJK_result)

-- print vector function name for the second table (for K=11)
print(probe_IJK_result[2].vector.func)

--print the value of third component of the vector function
--for the second table (for K=11)
print(probe_IJK_result[2].vector.value[3])
```

```
probe_IJK_current_scalar(probe_IJK_scalar_input, grid_number [,
dataset_number])
```

This function is similar to `probe_IJK_current_functions()`. Its performance is far greater as it does not return region, iso, threshold and vector values.

probe_IJK_scalar_input This table is the input argument for the function <code>probe_IJK_current_scalar()</code> .			
Field	Data Type	Comments	Default
I	number or table		
min	number	If datatype of I is table, this field exists.	
max	number	If datatype of I is table, this field exists.	
J	number or table		
min	number	If datatype of J is table, this field exists.	
max	number	If datatype of J is table, this field exists.	
K	number or table		
min	number	If datatype of K is table, this field exists.	
max	number	If datatype of K is table, this field exists.	

probe_IJK_scalar_result This table is returned by the function <code>probe_IJK_current_scalar()</code> .			
Field	Data Type	Comments	Default
n	number	number of results returned	
[n]	table	table of values of the nth result	
IJK	table	table of I, J and K values; only present if the grid is of type 'structured'	
point	table	table of x, y and z coordinates of point 'n'	
grid	number	number of the grid that the point belongs to	
scalar	table	table with scalar settings at point 'n'	
func	string	name of the scalar function; nil if no function	

value	number	value of the scalar function; nil if no function	
-------	--------	--	--

Surface to Surface Sampling for Dataset Comparison

Surface Sampling allows sampling of a results target surface using all of the nodes of a grid target surface. This is many times faster than Dataset Sampling for looking at comparisons on one surface with fewer restrictions (see [Dataset Sampling](#) in **Working with FieldView**).

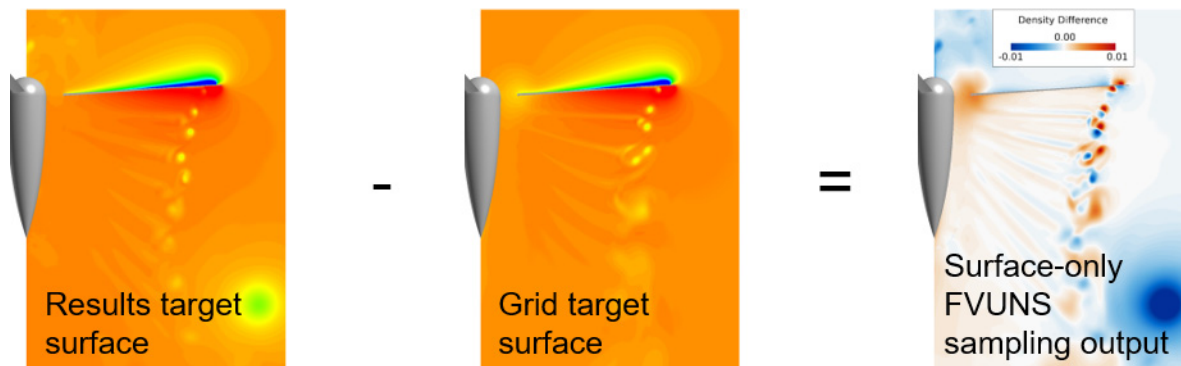


Figure 98 Surface to Surface Sampling for Dataset Comparison

Surface Sampling is done with an **FVX** command using **FVX** surface handles for the grid and results surfaces. **FVX** surface handles are returned by all **FVX** surface creation commands, and can be fetched for surfaces created outside of **FVX** by using the **FVX** `get_current_object_handle` command.

Surface Sampling creates an FV-UNS dataset made of the geometry of the grid target surface, stored as a boundary, and containing results from the grid target (variables are tagged with "G:") and also from the results target (tagged with "R:").

The "R:" and "G:" variables can then be compared using differences or ratios by creating a user-defined formula.

All solver variables from the grid target dataset and the results target dataset are output (with a few exceptions noted below under [Limitations](#)). If a user is interested in comparing fewer variables, lists of variables to be sampled from the results target and written from the grid target can be specified optionally to reduce compute and I/O time.

The FV-UNS files are created by the **FieldView** client, even if the datasets are on servers. The FV-UNS files can be read in "direct" mode, or by any server that can see the file location. Some versions of **FieldView** require that you use the Data Input option "Surface Sampled Data". This input option is also available for **FVX** with the option "surface_sampled_data", as described for the function `read_dataset()` in this chapter.

The original grid target dataset and results target dataset do not need to be present when you read these FV-UNS file outputs, making it very quick to load the sampling results for comparison.

The FV-UNS files have one grid, so using a parallel server to read them will not be any faster.

There are no restrictions on the datasets. Datasets can be a volume dataset or an XDB file or a boundary-only dataset, in any combination. One dataset can be client-server and the other can be "direct". One dataset can be from a server, and the other from a different server, or a parallel server.

The two surfaces must be in different datasets.

It is not necessary that either surface belongs to the current dataset. The current dataset and current surface is unchanged by surface sampling.

The two surfaces do not have to be the same type of surface. The intent is that the two surfaces have the same (or at least overlapping) geometry, but that is up to the user.

If the grid target surface is a boundary surface, the FV-UNS boundary types will match the grid target. For other types of grid target surfaces, the FV-UNS file will have a single boundary type whose name gives basic information about the grid target surface, such as "Coord Surf: R = 2.5" for an R = 2.5 cylindrical coordinate surface.

The preprocessing used for Surface Sampling has been decorrelated from the overall Grid Processing performed on the entire dataset. The recommended option for performing Surface Sampling is to set Grid Processing to "Less" at read time, which will save both time and memory.

FVX Syntax

```
sampling_output_table = surface_sampling(surface_sampling_table)
```

Input table `surface_sampling_table`:

```
surface_grid_target = FVX handle of grid target surface (required)
surface_results_target = FVX handle of results target surface (required)
surface_file = name (path) of combined grid & results or split grid-only FV-
               UNS
surface_results_file = name (path) of split results-only FV-UNS
surface_grid_variables = list of variables to be written from the grid target
surface_results_variables = list of variables to be sampled from the
                           results target
```

At least one of the two file names must be specified.

If only "surface_file" is specified, a combined (grid and results) FV-UNS file is created.

If both "surface_file" and "surface_results_file" are specified, then surface_file is an FV-UNS grid-only file, and surface_results_file is an FV-UNS results-only file.

If only "surface_results_file" is specified, then an FV-UNS results-only file is created, but not a grid-only file. You must generate a grid-only file at least once in order to use this results-only file. This is intended for cases where the same grid target surface is being used for comparison with multiple results target surfaces and the same grid-only output file can be reused.

The vertices of the grid target surface are used to sample (probe) the results target surface.

If a vertex from the grid surface lies on the results surface, then the sampling probe for that point will succeed, and **FieldView** will return scalar and vector values from the results surface for that point.

If a vertex from the grid surface does not lie on the results surface, then the sampling probe for that point will fail, and **FieldView** will return non-finite values (magenta) for all scalars and vectors at that point.

Output table (sampling_output_table):

```
matched_vertices = number of grid surface vertices that were
                    located inside the results surface
unmatched_vertices = number of grid surface vertices that were
                    not located inside the results surface
```

If a serious error occurred, the output table is nil and no FV-UNS files are created. An error message should be displayed as a pop-up or in the console.



Note: You can sample multiple surfaces at once by writing them into an XDB file. For example, you can write an XDB file containing a coordinate surface sweep with 25 sweep steps. The 25 sweep steps appear as 25 boundary types in the XDB file. Create a boundary surface containing all 25 sweep steps, and use this surface as the grid target surface. Surface sampling will produce an FV-UNS file with the same 25 boundary types. You can then look at the sampling results for the individual sweep steps (one boundary type at a time), or all at once.

Example:

```
-- Select the grid surface target
fv_script("SELECT DATASET 1")
```

```
fv_script("SELECT BOUNDARY 1")
surface_1 = get_current_object_handle()
-- Select the results surface target
fv_script("SELECT DATASET 2")
fv_script("SELECT BOUNDARY 1")
surface_2 = get_current_object_handle()

surface_sampling({
    surface_grid_target = surface_1,
    surface_results_target = surface_2,
    surface_file = "surface_sample.uns"
})
```

Limitations

Computational surfaces and structured boundary surfaces are not supported.

Dynamic clipping of either surface is not supported.

Face-based boundary variables are not supported.

PLOT3D/OVERFLOW-2 Q variables are not supported. PLOT3D and OVERFLOW-2 quantities derived from these are not supported.

Formulas and user-defined toolkit functions are not supported.



Note: All limitations above with the exception of face-based boundary [BNDRY] variables can be overcome by exporting surfaces and variables to XDB format and importing them back into **FieldView** prior to performing Surface Sampling.

Normals-based surface integrals are not supported on the resulting sampling output FV-UNS.

Grid target surface polygons with more than 256 vertices are skipped with a console warning.

Transient Data Handling

```
set_transient(transient_table [, dataset_number])
```

Causes **FieldView** to move to the specified time step or solution time on `dataset_number` or the current dataset if `dataset_number` is not provided. The input argument `transient_table` specifies the time step or solution time. There is no explicit output. The function returns an error if the dataset is not transient.

transient_table			
This table represents the information on a specific transient step. Transient steps do not have handles since they may not be deleted, only selected. Handles are not necessary since they have inherent identifiers, i.e., <code>time_step</code> or <code>solution_time</code> values.			
Field	Data Type	Comments	Default
<code>time_step</code> or <code>solution_time</code>	number	Only one of these fields should be specified. If both fields are provided, <code>time_step</code> overrides <code>solution_time</code>	

Example:

```
transient_table = {
    time_step = num,
}
set_transient(transient_table)
```

```
query_transient([transient_step], [dataset_number])
```

This returns a `transient_table` for a transient step. It returns a table described below for the current transient step if no argument is specified or a specific step when specified as input. The optional second argument is the `dataset_number`. If `dataset_number` is not provided, it defaults to the current dataset.

transient_table			
This table represents output from <code>query_transient</code> function.			
Field	Data Type	Comments	Default
<code>time_step</code>	number	Current time step.	n/a
<code>solution_time</code>	number	Current solution time.	n/a
<code>total_time_steps</code>	number	Total number of time steps in the current dataset. Read Only.	n/a
<code>time_step_range</code>	table	Read Only.	n/a
min	number	Minimal time step number.	n/a
max	number	Maximal time step number.	n/a
values	table	Table of time step numbers. It has <code>total_time_steps</code> entries.	n/a
<code>solution_time_range</code>	table	Read Only.	n/a
min	number	Minimal solution time in the dataset.	n/a
max	number	Maximal solution time in the dataset.	n/a
values	table	Table of solution time step values. It has <code>total_time_steps</code> entries.	n/a

Example:

```
--set the transient for transient step "a"
set_transient(a)
--retrieve the values of the current transient, i.e., "a"
query_transient()
--retrieve the values of transient step "b"
query_transient(b)
```

```
sweep_time (input_table [, dataset_number])
```

The command performs sweeping in time for transient data. If `dataset_number` is not provided, it defaults to the current dataset.

NOTE: The below `input_table` provides for use of either 'step' or 'solution time', but only one format can be used. For example, a table containing both a `from_time_step` field and a `to_solution_time` field will produce an error.

input_table; Format 1			
This table is an input table for sweep time command.			
Field	Data Type	Comments	Default
<code>from_time_step</code>	number*	'first', or an integer number. Required field	n/a
<code>to_time_step</code> *(see note)	number*	'last', or an integer number. Required field	n/a
<code>from_solution_time</code>	number	Real number. Required field	n/a
<code>to_solution_time</code>	number	Real number. Required field	n/a
<code>loop</code>	number*	Positive integer number.	1
<code>skip</code>	number*	Positive integer number. <code>skip=1</code> means no skip, <code>skip=2</code> means skip every other step, etc.	1
<code>cycles</code>	number*	Positive integer number.	1
<code>delta_time</code>	number	Positive real number. If specified, it will override any solution times for the dataset (to recover the original solution times, reissue the <code>sweep_time</code> command without a <code>delta_time</code> field).	none
<code>streaklines_filename</code>	string	If this field is not present and if the transient sweep produces streaklines, they will be exported to a file in the current directory with the default name: <code>streak<loginname_date_H_M_S>.fvp</code> For example: <code>streakuser_12Jun99_12_14_29.fvp</code>	
<code>extracts_database_name</code> ** (see note)	string	If provided, the transient sweep will export an XDB database. Note: no streaklines are exported when this field is present.	
<code>export_surfaces</code>	table	A table of tables, each specifying the export of a specific surface; see below (following Format 2) for details.	

* In general, in **FVX**, if a real number is supplied for a field that requires an integer number it will be truncated to integer. For example, if `loop=1.9`, it will be transformed to `loop=1`. However, `from_time_step` and `to_time_step` *must* be integers. For example, `from_time_step=1.9` will be rejected.

** The **Working with FieldView** section [Function Selection for XDB Export](#) describes the location and use of files ***xdb_vars*** and ***root.xfn*** which are **required** to specify the functions to be stored in your XDB extract when using `extracts_database_name`.

Subtable for transient export of visualization objects:

export_surfaces This is a subtable of the <code>input_table</code> to the <code>sweep_time</code> command. In FieldView 17 , only the first surface subtable will be used and any others will be ignored.			
Field	Data Type	Comments	Default
surface	table	<handle of a surface> In FieldView 17 , the handle must be that of a Coordinate or unstructured Boundary surface.	
basename	string		
type	string	"text", "csv" or "mat-file"	"text"

Graphing

`graph(graph_input)`

This function is used to create a 2D plot based on input criteria and an existing dataset. The input argument is the table `graph_input`. The output is a handle to a 2D plot which is displayed. The `graph` command is currently not supported on the MAC.

graph_input			
This table is used as the input argument to the graph function. The fields in this table specify the parameters of the graph to be drawn.			
Field	Data Type	Comments	Default
title	string	Example: "Graph Title"	
grid	table	<code>grid_table = {}</code> . When this empty table is present, the graph background will have a grid on it.	
background	string	<code>color_spec</code> - a string with the name of a known color or the hexadecimal value of the color, i.e., "#rrggbb" where "rr", "gg" and "bb" are hexadecimal representations of red, green and blue respectively.	
plotbackground	string	see <code>color_spec</code> above	
line	table	table with one or more entries of <code>line_table</code>	
bar	table	table with one or more entries of <code>bar_table</code>	
x_axis	table	table with one <code>axis_table</code> entry for the x-axis	
y_axis	table	table with one <code>axis_table</code> entry for the left y-axis	
y2_axis	table	table with one <code>axis_table</code> entry for the right y-axis	

line_table			
This table is used to represent properties of a line that will be drawn on a 2D plot. It is a field within a table which is in the <code>graph_input</code> table.			
Field	Data Type	Comments	Default
title	string	title for the line (optional).	"lineN"
xdata	table	A table with numerical indices and corresponding number values (required).	
ydata	table	A table with numerical indices and corresponding number values (required).	
option	table	<code>line_option_table</code>	

line_option_table			
This table is used to represent properties of a line that will be drawn on a 2D plot. It is a field within the table <code>line_table</code> .			
Field	Data Type	Comments	Default

color	string	This field is used to specify the color of the line as <code>color_spec</code> - a string with the name of a known color or the hexadecimal value of the color, i.e., “#rrggbb” where “rr”, “gg” and “bb” are hexadecimal representations of red, green and blue respectively.	
pixels	string	This field refers to the size of the markers in “N” - no. of pixels or “Nc” - distance in centimeters or “Ni” - distance in inches or “Nm” - distance in millimeters or “Np” - distance in printer points. <u>Note:</u> In all previous options, replace “N” with the numerical value.	
fill	string	This field is used to specify the fill color of the markers; see <code>color_spec</code> above.	
symbol	string	“square” or “circle” or “diamond” or “plus” or “cross” or “splus” or “scross” or “triangle”	
mapy	string	this specifies which y axis to use, i.e., “y” or “y2”. “y” is on the left side while “y2” is on the right side.	
smooth	string	smoothing for drawing line "linear" "step" "natural" "quadratic"	"linear"

bar_table

This table is used to represent properties of a line that will be drawn on a bar chart. It is a field within a table which is in the `graph_input` table.

Field	Data Type	Comments	Default
title	string	title for the line (optional)	“barN”
xdata	table	A table with numerical indices and corresponding number values (required).	
ydata	table	A table with numerical indices and corresponding number values (required).	
option	table	<code>bar_option_table</code> .	

bar_option_table

This table is used to represent properties of a bar that will be drawn on a 2D plot. It is a field within the table `bar_table`.

Field	Data Type	Comments	Default
barwidth	number	specifies the width of the bars in units along the X axis.	
foreground	string	<code>color_spec</code> - a string with the name of a known color or the hexadecimal value of the color, i.e., “#rrggbb” where “rr”, “gg” and “bb” are hexadecimal representations of red, green and blue respectively.	
background	string	see <code>color_spec</code> above	
mapy	string	this specifies which y axis to use, i.e., “y” or “y2”. “y” is on the left side while “y2” is on the right side.	

axis_table

This table is used to represent properties of an axis that will be drawn on a 2D plot. It is a field within a table which is in the `graph_input` table.

Field	Data Type	Comments	Default
<code>title</code>	string	Axis title	
<code>titlecolor</code>	string	<code>color_spec</code> - a string with the name of a known color or the hexadecimal value of the color, i.e., “#rrggbb” where “rr”, “gg” and “bb” are hexadecimal representations of red, green and blue respectively.	
<code>logscale</code>	string	turn “on” or turn “off” logarithmic scaling of axis	“off”

Example:

```

position = {1,2,3,4,5}
pressure = {1,5,6,2,.5}

graph_table = {
  title = "Axial Variation of Pressure",
  grid = {},
  background = "white",
  plotbackground = "gray",

  line = {
    {
      title = "Pressure",
      xdata = position, --table of x-axis positions
      --table of corresponding y-axis positions
      ydata = pressure,
      option = {
        fill = "red",
        symbol = "diamond"
      },
    },
  },

  x_axis = {
    title = "Axial Position",
    titlecolor = "black"
  },

  y_axis = {
    title = "Pressure",
    titlecolor = "black"
  }
}

```

```
graph_handle = graph(graph_table) -- create 2D plot
```

```
postscript_output(graph_handle, output_file_name, postscript_output_options_table)
```

This function allows a graph to be saved as a postscript file. The input arguments to this function are the following: the variable holding the handle for the graph, the name of the output file and the table that specifies the postscript output options.

postscript_output_options_table			
This table will be used by the function <code>postscript_output()</code> for specifying postscript options.			
Field	Data Type	Comments	Default
center	string	"on" or "off". Centers graph on page.	"on"
colormode	string	"color" or "gray" or "mono"	"color"
decorations	string	"on" or "off". Option to draw background and border. Note: set to "off" to force white background for printing.	"on"
landscape	string	"on" or "off". If "off" is selected, the output is in "portrait" orientation.	"off"
maxpect	string	"on" or "off". If "on", the graph will scale to the largest size that will fit on the page while maintaining aspect ratio.	"off"
width	string	"N"- no. of pixels or "Nc" - distance in centimeters or "Ni" - distance in inches or "Nm" - distance in millimeters or "Np" - distance in printer points. Note: In all previous options, replace "N" with the numerical value.	size on screen
height	string	"N"- no. of pixels or "Nc" - distance in centimeters or "Ni" - distance in inches or "Nm" - distance in millimeters or "Np" - distance in printer points. Note: In all previous options, replace "N" with the numerical value.	size on screen

Example:

```
graph_handle = graph(graph_table)

--set output specifications
postscript_output_options_table = {
    colormode = "gray",
    landscape="off",
    maxpect="off",
    width="6i", --set width to six inches
    height="8i" --set height to eight inches
}

--set output file name
outfile = "graph.ps"
```

```
--output postscript file
postscript_output(graph_handle,outfile, postscript_out-
put_options_table)
```

GUI Functions

`make_panel(make_panel_input)`

This function allows users to create their own panels. It takes as its input argument a table describing the widgets that the user wishes to create. The user may add an arbitrary number of widgets. Each widget may be used zero or more times. This function returns the handle to the panel object. Widgets within this panel are assigned numbers starting from one, in the order that they are specified in the input table. This allows specific access to the widget since the returned handle is a table with number indices. The `make_panel` function calls `set_preserve_globals(1)` to preserve global variables since panels can be used after a script completes. The `make_panel` command is currently not supported on the MAC.

make_panel_input			
This table will hold the information needed by the <code>make_panel()</code> function to create the panels as desired by the user.			
Field	Data Type	Comments	Default
title	string	panel title	
xloc	number	horizontal location of the top left corner of the panel in pixels	
yloc	number	vertical location of the top left corner of the panel in pixels	
width	number	width of the panel in pixels	
height	number	height of the panel in pixels	
text widget spec table	table	zero or more entries of table	
type	string	"text" - an area that accepts text input	
title	string	title for the panel	
action	function	function definition or variable pointing to function	
button widget spec table	table	zero or more entries of table	
type	string	"button" - a labeled button which may be pressed	
title	string	title for the slider	
action	function	function definition or variable point to function	
slider widget spec table	table	zero or more entries of table	
type	string	"slider" - a slider whose value may be set by slider or type-in entry. The 'min' and 'max' values of the sliders are displayed as read-only entries.	
title	string	title for the panel	
action	function	function definition or variable point to function - called when the user stops dragging the slider	
drag_action	function	function definition or variable point to function - called while the user drags the slider	
value	table	The initial value for the slider	
abs_min	number		

current	number		
abs_max	number		
digits	number	An integer specifying how many significant digits should be retained when displaying the slider value. If the number is less than or equal to zero, then the scale picks the smallest value that guarantees that every possible slider position prints as a different string.	

```
self:set()
```

This function can be used in the `action` fields of the input table for the `make_panel()` function. It sets the input argument as the value of the widget where this function is used.

```
self:get()
```

This function can be used in the `action` fields of the input table for the `make_panel()` function. It gets the value of the relevant field in the widget where this function is used.

Example:

```
--define functions to be used with panel
function button_test()
    print ("Result of button test.")
end

function slider_test(new_pos)
    print ("Current slider is ",new_pos)
end

--set panel specifications
make_panel_input = {
    title = "demo panel",
    xloc=700,yloc=100 ;

    { --button widget
        type = "button",
        title = "Button Test",
        action = function()
            button_test()
        end
    },

    { --text widget
        type = "text",
        title = "Enter text followed by carriage return:",
        action = function(self)
            print("text is "..self:get())
        end
    }
}
```

```

    },

    { --slider widget
      type = "slider",
      title = "Slider Value:",
      value = {
        abs_min = 1,
        current = 50,
        abs_max = 100,
      },
      drag_action = function(self)
        slider_test( self:get().current )
      end,
      action = function(self)
        slider_test( self:get().current )
      end
    }
  }

  --create panel
  panel_handle = make_panel(make_panel_input)

```

Other Functions

`fv_script()`

This function is used to execute **FieldView** script commands that are provided as text input. The input consists of one string representing **FieldView** script. There is no explicit output. See [Chapter 5](#) in this **Reference Manual** for full documentation on **FieldView** Script Language Commands.

Example:

```

--execute the specified restart file
fv_script("RESTART BOUNDARY aerospace.bnd")

```

`redraw()`

This function is used to refresh the **FieldView** screen when changes are made to surfaces. There is no input provided. There is no explicit output. This function is not operational while **FieldView** is in batch mode.

Example:

```

redraw()

```

`set_auto_redraw([arg])`

If the argument is not set, or is `nil`, then graphics updates will not occur unless the **FVX** script encounters a `redraw()` command (above). If `set_auto_redraw([arg])` is set with a non-`nil` argument, the default behavior will be restored, where **FieldView**

updates graphics normally. This function is useful to control what images are captured with the command `fv_script("record on")`.

Example:

```
set_auto_redraw() --turns off auto redraw
```

```
set_preserve_globals([arg])
```

If the input argument is non-nil, global variables are preserved across script invocations. The default behavior is to not preserve global variables. However if the function `make_panel()` is used, it calls `set_preserve_globals(1)`. This is needed since panels can be used after a **FieldView** script completes.

Example:

```
set_preserve_globals() --turns off global preservation
```

```
dump(variable)
```

This function accepts a variable as input and outputs detailed information on the variable such as its value and type. In the case of functions and complex tables (i.e., with tables and functions as fields), value does not apply and the variable's memory address in hexadecimal format is output instead.

Example:

```
a = "hello"
b = "2"
c = { {a=1}, {"hello"} }
function d() print("temp") end
dump(a) --outputs:  string      hello
dump(b) --outputs:  number      2
dump(c) --outputs:  1 table      table: 046CEAD0
                  2 table      table: 046CEAB0
dump(d) --outputs:  function     function: 046CF2F0
```

```
dumpall(variable)
```

This function is similar to the `dump` command but outputs more detailed information on the input variable by fully expanding tables.

The following example compares output from the two commands following a query.

```
coord_table = query(my_surface)
```

```
dump(coord_table)
```

```
table: 0FF822F0
threshold_range  table      table: 0FE50D18
display_type     string      constant_shading
scalar_func      string      Density (Q1)
```

X_axis	table	table: 0FE50E38
axis	string	X
Z_axis	table	table: 0FE50E10
visibility	string	on
Y_axis	table	table: 0FE50D40
threshold_func	string	$(Y^2+Z^2)^{.5}$
dataset	number	1
vector_func	string	none

```
dumpall(coord_table)
```

```
table: 13AB4320
```

threshold_range	table	table: 0FE50D18
abs_max	number	6.021609783172607
max	number	0.6000000238418579
abs_min	number	0
min	number	0
display_type	string	constant_shading
scalar_func	string	Density (Q1)
X_axis	table	table: 0FE50E38
abs_max	number	7.829783916473389
current	number	0.9049365520477295
max	number	7.829783916473389
abs_min	number	-6.01991081237793
min	number	-6.01991081237793
axis	string	X
Z_axis	table	table: 0FE50E10
abs_max	number	6.019911289215088
max	number	6.019911289215088
abs_min	number	-3.059648224734701e-005
min	number	-3.059648224734701e-005
visibility	string	on
Y_axis	table	table: 0FE50D40
abs_max	number	6.019956111907959
max	number	6.019956111907959
abs_min	number	-6.019910335540772
min	number	-6.019910335540772
threshold_func	string	$(Y^2+Z^2)^{.5}$
dataset	number	1
vector_func	string	none

```
stop()
```

This function allows for an **FVX** program to be paused during execution. Its usefulness is for the purpose of debugging code. This function stops the program at the line where it is called, then brings up the debugger prompt in the terminal window from which **FieldView**

was executed (Linux) or the console window (Windows). This function is called without any input arguments. See also [FVX Debugger](#) commands [stop at \[line\]](#) and [stop in \[function\]](#).

Dynamic Clipping

A dynamic clip group can be created by specifying a series of clip definitions:

```
dynamic_clip_table={
  dataset = 1,
  name = "clipping_line",
  clip_definitions = {
    { point = {3.27324,    5.43,    4.6723  },
      normal = {3.27324, 18.105,  22.874  }
    },
    { point = {3.27324,    4.1625,  2.85213 },
      normal = {3.27324,    2.8950,  1.03196 }
    },
  },
  }, -- end of definitions
  active = "on|off",
}
create_dynamic_clip(dynamic_clip_table)
```

dynamic clip table			
Field	Data Type	Comments	Default
dataset	number	dataset number	current
name	string	must have a name (explicit definition required)	
clip_definitions	table	table with one or more entries of clip definition table	
active	string	"on" or "off"	"off"

clip definition table			
Field	Data Type	Comments	Default
point	table	{point1, point2, point3}	
normal	table	{point1, point2, point3} ordered as X, Y, Z in dataset space	

FieldView automatically checks for active **FVX**-defined dynamic clips and GUI-defined dynamic clips. If an **FVX** defined dynamic clip is turned ON, then ALL the other **FVX**- and GUI-defined dynamic clips are turned off. Similar to other surfaces, the user can create, delete, modify, and query the `create_dynamic_clip` function via the handle. These **FVX** commands can only be applied to **FVX**-created dynamic clips, and not to GUI-created dynamic clips. The dynamically created GUI must also be maintained. For more information on Dynamic Clipping, see [Chapter 14](#).

FVX View Controls

set_view

An **FVX** utility is provided on the **FieldView** DVD (see /fvx_and_restarts/set_view.fvx) to allow for manipulation of the view parameters which will permit user defined orientation of a given dataset. This utility creates a temporary View Transform restart file (*.vct) and then applies the restart to the current dataset. To illustrate its use, a simple example program is also provided (see /fvx_and_restarts/set_view_TEST.fvx).

set_view(set_view_table)

Sets view properties for datasets in **FieldView**. The input is the set_view_table. Note: This utility requires a dataset to have been read in by the read_dataset() **FVX** function.

set_view_table This table of fields represents FieldView 's view settings.			
Field	Data Type	Comments	Default*
dataset_info_table	table	Output table handle of function read_dataset()	
Zoom	number		1
angle_axis	table		
at	table	X, Y, Z point coordinate of "look at" view point	
from	table	X, Y, Z point coordinate of "look from" view point	
up	table	Vector designating upward direction	{0, 1, 0}
translation	table		
x	number		
y	number		
z	number		
outline	string	"on" or "off"	"on"
perspective**	table		
angle	number	Measured in degrees	40
z	number	Value depends on scaling of dataset	-2.75
axis_marker	string	"on" or "off"	"on"
mouse_mode	string	"track" or "running"	"track"
light_direction	table	Table of X, Y, and Z vector components	
x_light	number		0.577350
y_light	number		0.577350
z_light	number		0.577350
rotation_center**	table	Table of X, Y, and Z vector components	
x_rot	number		0
y_rot	number		0

<code>z_rot</code>	number		0
<code>quick_transparency</code>	string	"on" or "off"	"off"

*Default settings for these fields are applied if and only if there are no previous field settings detected.

**Excluding these fields completely from the input table will set their toggle status to "off" (field equals nil); setting these fields equal to an empty table {} initiates the default options.

FVX Debugger

The **FVX** debugger enables **FieldView** users to track down errors in their code in a systematic fashion. The debugger is triggered whenever an error occurs during program execution. The commands given below may then be used to trace the source of the error(s). The syntax is similar to the 'dbx' debugger used on UNIX platforms. The debugger may be started forcibly during **FVX** execution by pressing the ESCAPE key, while the cursor is in the graphics window or by inserting a `stop()` command within an **FVX** file.

`assign`

`assign exp1 = exp2`

`call procedure(args)`

evaluate an assignment or function call immediately

`print procedure(args)`

`cont/c`

continue program execution

`cont to [line]`

continue program execution until specified line is reached

`delete [breakpoint_number]`

delete specified breakpoint

`delete all`

delete all breakpoints

`down`

move down one stack level (argument not supported - always 1)

`dump (tbl)`

displays all entries in tbl

`dumpall (tbl)`

displays all entries in all nested tables

`find ref`

print all locations which contain a reference to ref

`list`

list source code

`level num`

set stack to specified level

`next/n`

execute next line (argument not supported - always 1)

`print/p`

print expressions

`quit/q`

quit the debugger and return to **FieldView**

`return`

execute until function return (argument not supported - always current function)

`step`

execute next line, stepping into function calls (argument not supported - always 1)

`stop at [line]`

set a breakpoint at the specified line

`stop in [function]`

set a breakpoint at the specified function

`up`

move up one stack level (argument not supported - always 1)

`where`

display the current call stack (argument not supported - always prints the entire stack)

Access to FVX Programs from the Tools Menu

FVX utilities are located in the `fvx_and_restarts` subdirectory of the standard **FieldView** installation. For information on **FVX** utilities directly accessible from the **FieldView** Tools menu, see [FVX Utilities](#) in [Chapter 14](#) of **Working with FieldView**.

Python-Enabled FVX

Although the **FVX** programming language in **FieldView** is based on the LUA scripting language, **FieldView** is also capable of working with Python and can be used with either of the two main Python versions (2 or 3). **FVX** functions can be called from Python, and Python data structures can be passed in and out of **FVX**. The implementation involves integrating a Python environment and custom programming into **FieldView**. The structure is sketched below in [Figure 99 Python-Enabled FVX Scheme](#):

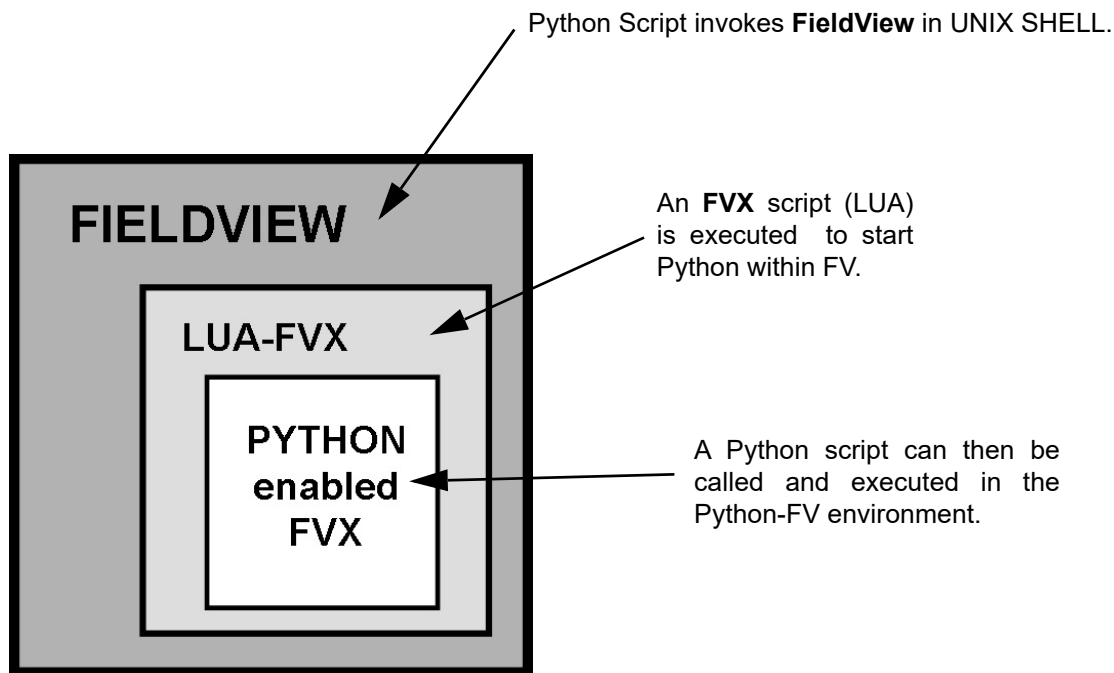


Figure 99 Python-Enabled FVX Scheme

Our design implements the **FVX** post-processing calls such as reading data and creating, modifying, querying and deleting surfaces and streamlines. This is done in a manner that allows the **FVX** data structures to be accessed as though they were native Python structures.

To configure your version of Python, please consult [Python Installation \(Optional\)](#) in the **Installation Guide**.

The choice of namespace for Python will have an impact on the command syntax. For example, choose either of:

```
import fv
from fv import *
```

```
FVX cmd is fv.create_boundary
FVX cmd is create_boundary
```

There are additional implications on the commands themselves. A valid **FVX** boundary surface input table is:

```
boundary_table = { scalar_func = 'Normalized density [PLOT3D]',  
                    types = {'body', 'wing'},  
                    display_type = 'smooth_shading' }
```

The same input table, in Python would appear as:

```
boundary_table = { scalar_func : "Normalized density [PLOT3D]",  
                    types : {1 : 'body', 2 : 'wing'},  
                    display_type : 'smooth_shading' }
```

Support for Tkinter

To provide more Graphical User Interface (GUI) customization options, **FieldView** supports the execution of Python scripts which include Tkinter function calls. Tkinter is a Python binding to the Tk GUI toolkit. With support for a wide range of GUI widgets, it has become the Python standard for building GUIs.

Tkinter comes with the Python installation provided with **FieldView**, but if you wish to link **FieldView** to your own Python installation, see [Python Installation \(Optional\)](#) in the **Installation Guide**.

The figure below shows a Tk panel obtained from the execution of a Python script in **FieldView**. This panel shows some examples of the various widgets supported by Tkinter.

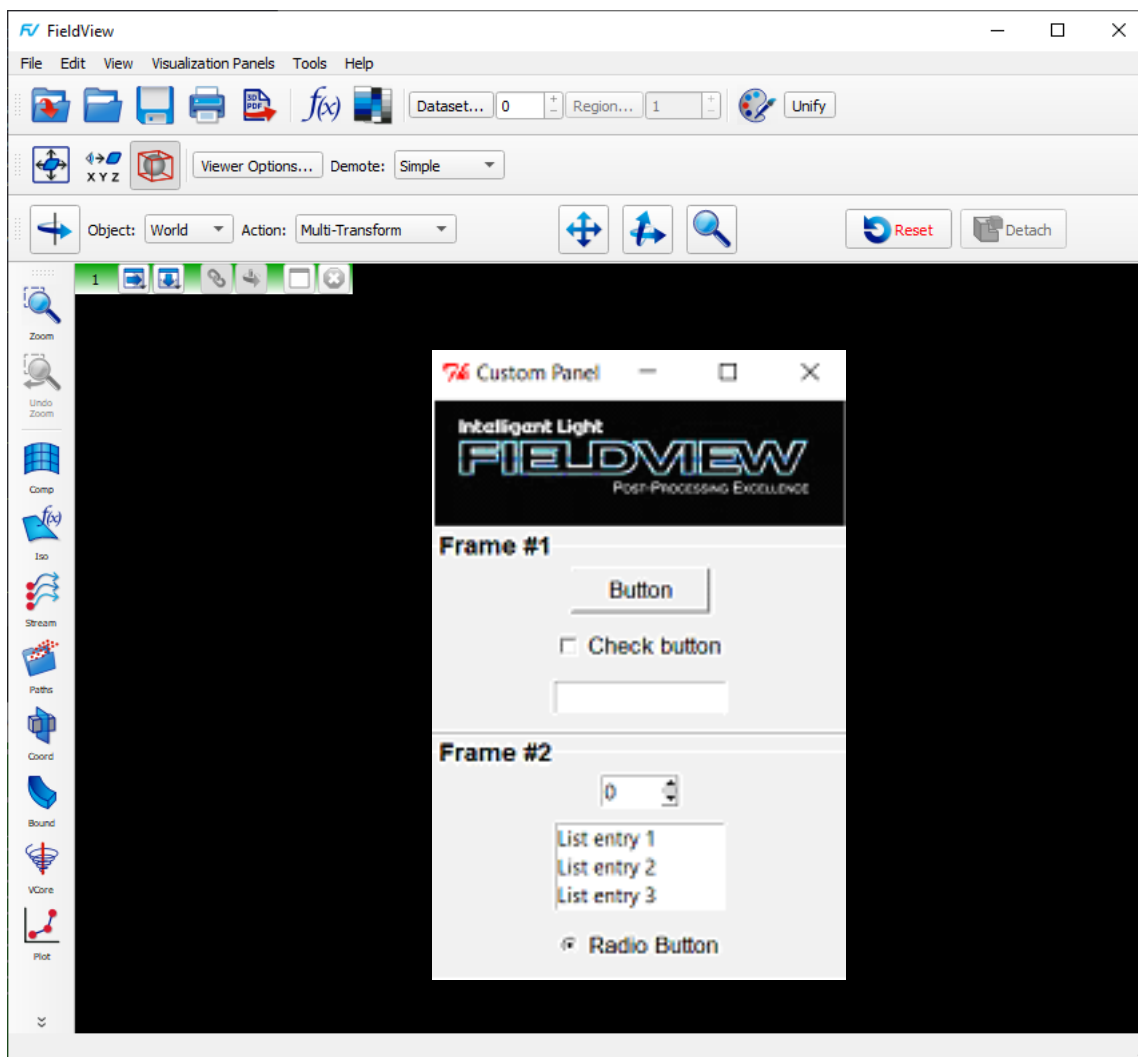


Figure 100 Custom GUI panel showing some Tkinter widget examples

Such widgets can be used for calling Python-enabled **FVX** functions. Tkinter support has been added to the **FieldView** event loop, allowing two-way interactions between Tk panels and **FieldView**, which means that user-defined panels can behave exactly like original **FieldView** panels.

Tkinter can be imported in a Python script using the import command:

```
import Tkinter
```

Limitations for Tkinter

Tkinter is not supported on the Mac.

Tkinter scripts run from **FieldView** must use the default root window.

Known Limitations for Python-Enabled FVX

There is no debugging support. If your Python script has an error in it, it is likely to crash **FieldView**.

The **FVX** `make_panel()` command is not supported.

FVX functions not enabled for Python:

```
appendto(filename)
closefile(handle)
dostring(cmd_string)
dump(tbl)
dumpall(tbl)
execute(cmd)
format(string, arg, ...)
getn(tbl)
openfile(filename, mode)
read(handle, format...)
readfrom(filename)
remove(filename)
rename(oldname, newname)
self:get()
self:set()
set_view({tbl})
stop()
strfind(string, pattern, ...)
tinsert(tbl, pos, value)
tonumber(string[, base])
tostring(number)
tremove(tbl, pos)
type(arg)
write(handle, arg, ...)
writeto(filename)
```

FVX Learning Tools

To better educate users on the versatility and ease of using **FVX**, several tutorials and utilities are included on the **FieldView** DVD.

FVX Tutorial Scripts

FVX scripts have been generated for most of the **FieldView** tutorials. The scripts are intended to demonstrate the basic use of **FVX** to read datasets, create surfaces, and perform calculations similar to those actions performed in the **FieldView** tutorials. In some cases, **FieldView** restarts have been used to recreate steps illustrated in the tutorials.

To test out these **FVX** scripts, we recommend that you create a subdirectory or folder. Put the tutorial dataset into this folder, and copy the **FVX** scripts and associated restart files. **FVX** scripts for each tutorial are located within the respective tutorial folder on the **FieldView** DVD and are named according to the following naming convention:

```
sampleXXXXXXfvx.fvx
```

where "XXXXXX" indicates the associated tutorial name.

To run these scripts, you do not need to load the data first: the script will do this for you. Several break points have been introduced into these scripts so that the results of each intermediate step can be viewed. When a break point is reached, the **FVX** script is paused. To advance the **FVX** script from a break point, you need to have access to the console window or the window where you launched **FieldView**. Press Enter to continue through to the next break point. Since some of these scripts will generate files or animations, it is important that you have write permissions to these directories.

Note: For maximum utility, try opening an **FVX** tutorial script in a text editor while running the same **FVX** tutorial in **FieldView** to observe how the **FVX** syntax and commands create each tutorial step.

FVX Templates

As the use of **FVX** has become more widespread, the need for faster script construction has been addressed. A set of **FVX** Templates is included on the **FieldView** DVD and `FVX_Templates.pdf` is located in the `fvx_and_restarts` subdirectory of the standard **FieldView** installation. **FVX** example commands can be copied from the templates and pasted into working **FVX** scripts. Templates for most of the large **FVX** functions are included with all possible features printed, allowing the user to customize surfaces, rakes, feature extractions, plots, GUI panels, output files, etc. with any or all available options. Unwanted options can be easily deleted or commented out of the **FVX** script.

Guide FVX saved with Restarts

Whenever you save a Complete; Complete, Current Window; or Current Dataset Restart, a Guide **FVX** program is also saved with it automatically. This Guide **FVX** program contains the **FVX** commands needed to re-create the surfaces, rakes and other post-processing objects that are on the screen at the time that the Restart is written. Our intent with this feature is to show you the equivalent **FVX** commands for creating the surfaces, rakes and annotation objects on the screen. This gives you a way to learn **FVX** and compare it with what is being saved with your Restarts.

This Guide **FVX** program has the same rootname as the restart. Not all the functionality within a full restart can be captured using **FVX**. For this feature, the Guide **FVX** program recreates, as best it can, the same visual representation of the data. For a Complete Restart, the Guide **FVX** program contains **FVX** command language for:

- Data Input
- Boundary Surfaces
- Computational Surfaces
- Coordinate Surfaces
- Iso-surfaces
- Streamlines
- Particle Paths
- Annotation
- Dynamic Clipping

Several query functions are available in **FVX** to return a list of all the boundary surfaces, scalar and vector functions and surface scalar and surface vector functions. These queries are provided for each dataset in the Guide **FVX** output.

For the case of transient data, **FVX** commands to query the transient dataset and do a transient sweep are written.

Several additional queries to return information on the color table, the streamline display and particle path display are also included.

For a Complete Restart, the Guide **FVX** program uses the `fv_script` command to launch the component restarts for:

- Formula (`.frm`)
- Vortex Cores (`.vtx`)
- View (`.vct`)
- Colormap (`.map`)
- 2-D Plots (`.lpt`)
- Presentation (`.prd`)

Although the **FVX** implementation does not cover everything that can be done with Restarts, the original restart matching the Guide **FVX** program is saved and can be used for comparison.

In some instances, it may not be desirable to automatically save the Guide **FVX** program with a restart. One example might be a case in which many streamline rakes have been created - the **FVX** listing for these rakes is verbose. Therefore, an environment variable, `FV_NO_FVX_RESTART` has been implemented to turn this behavior off.

When **FieldView** writes a Complete Restart, there are some situations in which a Guide **FVX** program is not written. These instances include restarts saved during transient sweeps, XDB building sweeps, keyframe restarts, and restarts saved when **FieldView** terminates unexpectedly. If **FieldView** is in a multi-window state, a Guide **FVX** file will not be saved with a Complete Restart (returning to a single window will allow Guide **FVX** saving).

Chapter 5

Restart Files and Script Language

5

Restart files allow the current state of each individual panel, or the state of the entire system, to be saved and read in at a later time. In this way, you can restart the system and bring up a previous view of a particular dataset with only one command. Note that except for the Complete Restart and Data File options, the names of the dataset input files are not saved. Thus, by using the individual panel options of the Restart menus, a new dataset may be viewed in the same way as a previous dataset quite easily. The Current Dataset option and the Complete, No Data Read option under Open Restart also facilitate this.

Another Open Restart menu option is Script. **FieldView** will attempt to interpret the text in this file as a series of commands, as defined in the **FieldView** Script Language section. You are responsible for creating the script files, with a text editor or a script generator program, according to the rules described in that section. While executing the script, command errors are reported to you. **FieldView** will beep when script execution is completed or canceled.



Note: **FieldView** is 100% backward compatible with all Restart Files for all releases. Every attempt is made to ensure forward compatibility as well.

Restart Files Menu

The current state of your **FieldView** visualization can be saved or restored with the use of Restart Files.

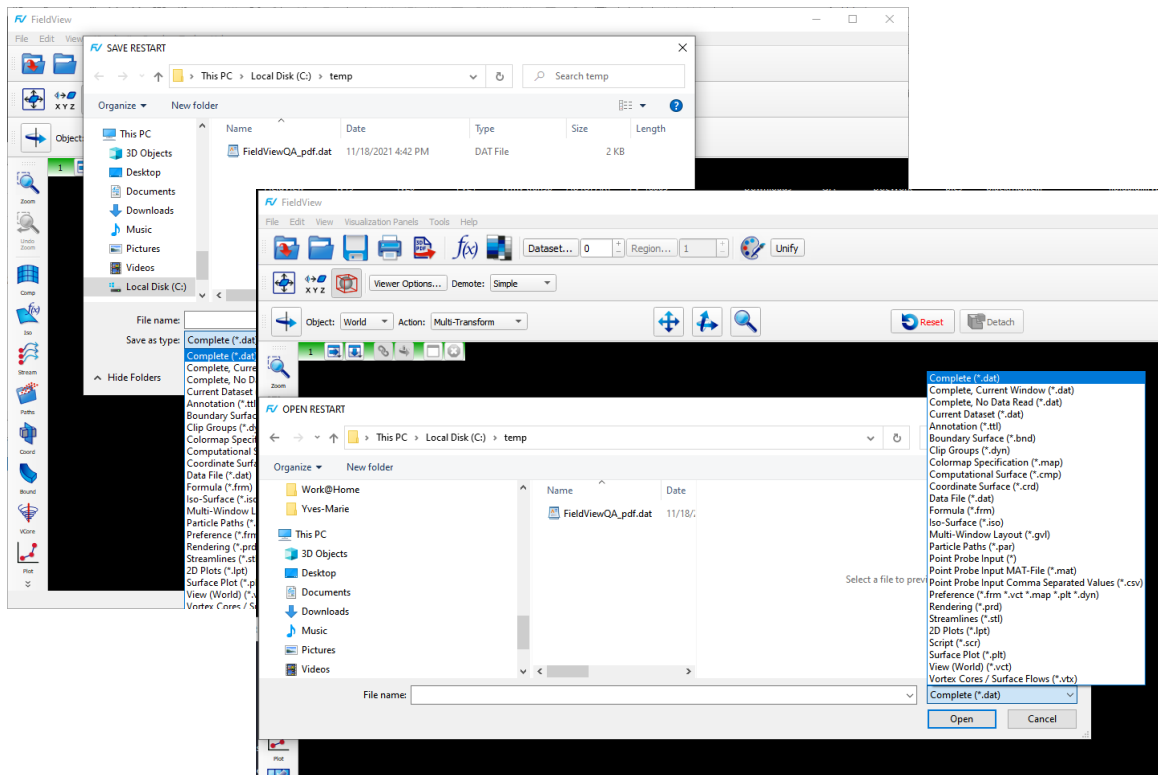


Figure 101 Save/Open Restart menu options

The **Save File** and **Open File** buttons on the Main Menu always open with the Data Restart (.dat) restart type selected, but can be changed. Also, using the **File.. Open Restart** or **File.. Save Restart** options will produce a file dialog which allows you to choose the desired restart type, and will default to the last type of restart you had used.

File Naming Convention

The general form for Restart filenames is: `filename.extension`

File Type	Extension
Formula	.frm
Data File Input	.dat
Computational Surface	.cmp
Iso-Surface	.iso
Streamlines	.stl
Particle Paths	.par
Annotation (titles & arrows)	.ttl
View (World)	.vct
Colormap Specification	.map
Surface Plot	.plt
Script File	.scr
Boundary Surface	.bnd
Vortex Cores / Surface Flows	.vtx
Coordinate Surface	.crd
2-D Plots	.lpt
Presentation	.prd
Point Probe	.prb
Clip Groups (Dynamic Clipping)	.dyn

These extensions are automatically appended when saving or reading, except when saving a Script File.

Automatic Restart

FieldView can read a particular Preference or Complete Restart file during startup by using a command line argument (`-f filename`) or by naming the restart file `fv`. In addition, **FieldView** may startup and automatically read in a script file (`-s filename`). See [Chapter 1](#) of the **User's Guide** for more information.

Restart Flexibility

Certain restarts are more flexible for ease of use in multiple datasets. That is, a visualization created for one dataset and saved to a Complete Restart or Current Dataset Restart can be used on other datasets, whether the other datasets use the same grid (but different results) or are completely different. The “flexible” restarts are:

- Boundary Surface
- Clip Groups
- Computational Surface
- Coordinate Surface

- Iso-Surface
- Particle Paths
- Streamlines
- Vortex Cores/Surface Flows

If an Iso-Surface, Coordinate Surface, or Computational Surface in a flexible restart falls outside of the range of the `Min/Max` for the dataset for which it was created, then the surface will be created at the closest possible value (`Min` or `Max`) but the Visibility of the surface will *not* be `ON`. For example, if the restart includes a `Z=-0.5` Coordinate Surface but the current dataset has a `Z` range of `0.0` to `1.0`, then a surface will be created at `Z=0.0` but its Visibility will be `OFF`.

Streamline seeds and surfaces that were created on a grid that belonged to dataset `N` will be deleted if that grid does not exist in dataset `N` in memory. For example, if a flexible restart is created with three 9-grid datasets in memory and later read in with only 2 single-grid datasets in memory, any surface/rake that is encountered in the restart files that refer to non-existent grids or datasets will be ignored. For best results, you should have the same number of grids and datasets for restarts to work in this manner. Another possible restart mismatch is incompatible coordinate systems (Cartesian vs. Cylindrical) via the region file (see [Chapter 3](#) of this **Reference Manual**).

In general, **FieldView** will ignore surfaces or seeds in any flexible restart in which any one or more of the following are true:

- Surface or seed refers to a grid number that is greater than the number of grids for the corresponding dataset.
- Object refers to a dataset number that does not exist for the current session of **FieldView**. This is true for all flexible restarts.
- Subsetting is different for the datasets. This applies to Iso-Surfaces, Coordinate Surfaces and Computational Surfaces.

Restart saved on Exit from FieldView

Upon exit, **FieldView** will save a complete restart named "`fv_backup_restart`". The '`fv_backup_restart`' files will be overwritten without confirmation.

For Linux and Mac, **FieldView** will save the backup restart in:

The directory specified by the environment variable `HOME`, or
The directory specified by the environment variable `TMPDIR`, or
`/tmp`

For Windows, **FieldView** will save the backup restart in:

"My Documents", or
The directory specified by the environment variable `HOME`

An environment variable, `FV_NO_BACKUP_RESTART`, can be used to disable the saving of the backup restart. The Guide **FVX** restart, `fv_backup_restart.fvx` will also be saved as well unless the environment variable `FV_NO_FVX_RESTART` is set.

When the exiting is due to calling `exit()` from an **FVX** or Python script, **FieldView** will not save this restart on exit.

Restart Files Operation

The **Save File** and **Open File** buttons on the Main Menu, as well as the **File.. Open Restart** or **File.. Save Restart** options, will produce the following panel.

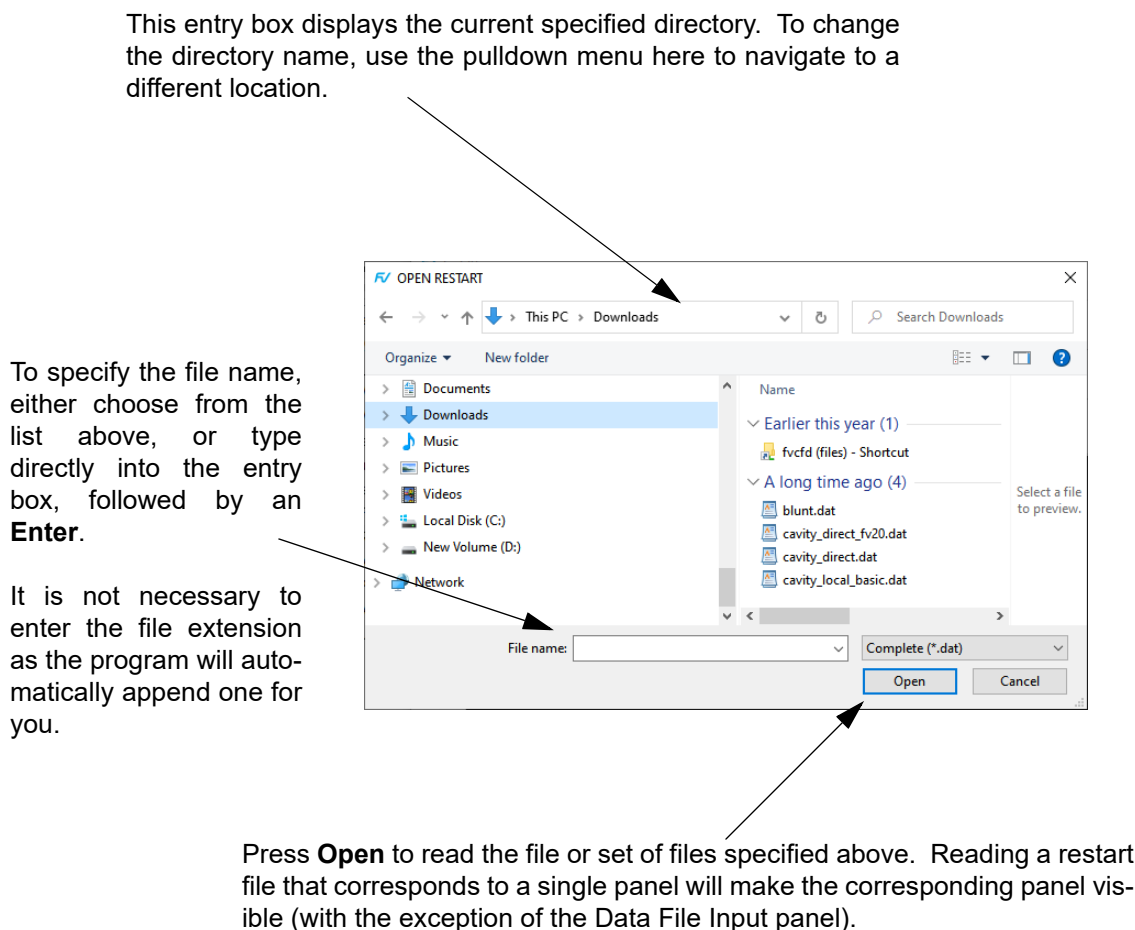


Figure 102 Restart File Panel



Note: Some read-only restart file types, such as Script Restart, do not provide a Save button on the restart panel.

Note: When saving a file, if you give the file the same name as a file that already exists, you will be given the option to overwrite the existing file (select Overwrite) or cancel the save operation (select Cancel).

Complete Restart

A Complete Restart is a *set* of files consisting of all **FieldView** panels that contain data, essentially combining the function of all restart types. The Complete Restart panel is used to specify a base filename (i.e., without an extension). **FieldView** will automatically append required file extensions (see table [File Naming Convention on page 262](#)) and save or read all component restarts. Minimally, the following component files are saved with a Complete Restart:

- Data File Input (*.dat)
- Colormap (*.map)
- View (World) (*.vct)
- Presentation (*.prd)
- Surface Plot (*.plt)

Script Restart and Point Probe Input are *not* saved as part of a Complete Restart.

When a Complete Restart is saved interactively, either from the File → Save Restart → Complete... fly-out menu (see [Figure 101](#), right) or by pressing the Save Restart icon on the Main toolbar, the contents of all the windows and the multi-window layout information is saved. When a Complete Restart is read interactively, either from the File → Open Restart → Complete... fly-out menu (see [Figure 101](#), left) or by pressing the Open Restart icon on the Main toolbar, the contents of all the windows and the multi-window layout information is restored.

Complete Restarts saved prior to **FieldView** Release 14 will restore all the visualization objects for all datasets within a single window.



Note: Pathnames to saved Complete Restart files must be preserved when referencing them in future **FieldView** sessions.

For information on reading Complete Restarts with 2D Plots, see [2D Plots Restart on page 279](#).

Complete, Current Window...

The Complete, Current Window... Restart entry permits you to save or restore a complete restart from or into the current window of a multi-window layout. Use this to read

single-window restarts, one window at a time, into any multi-window session. In [Figure 103](#), we start with a multi-window layout containing four windows. Restarts for some of the **FieldView** tutorial datasets are used to read the data and create the visualizations shown.

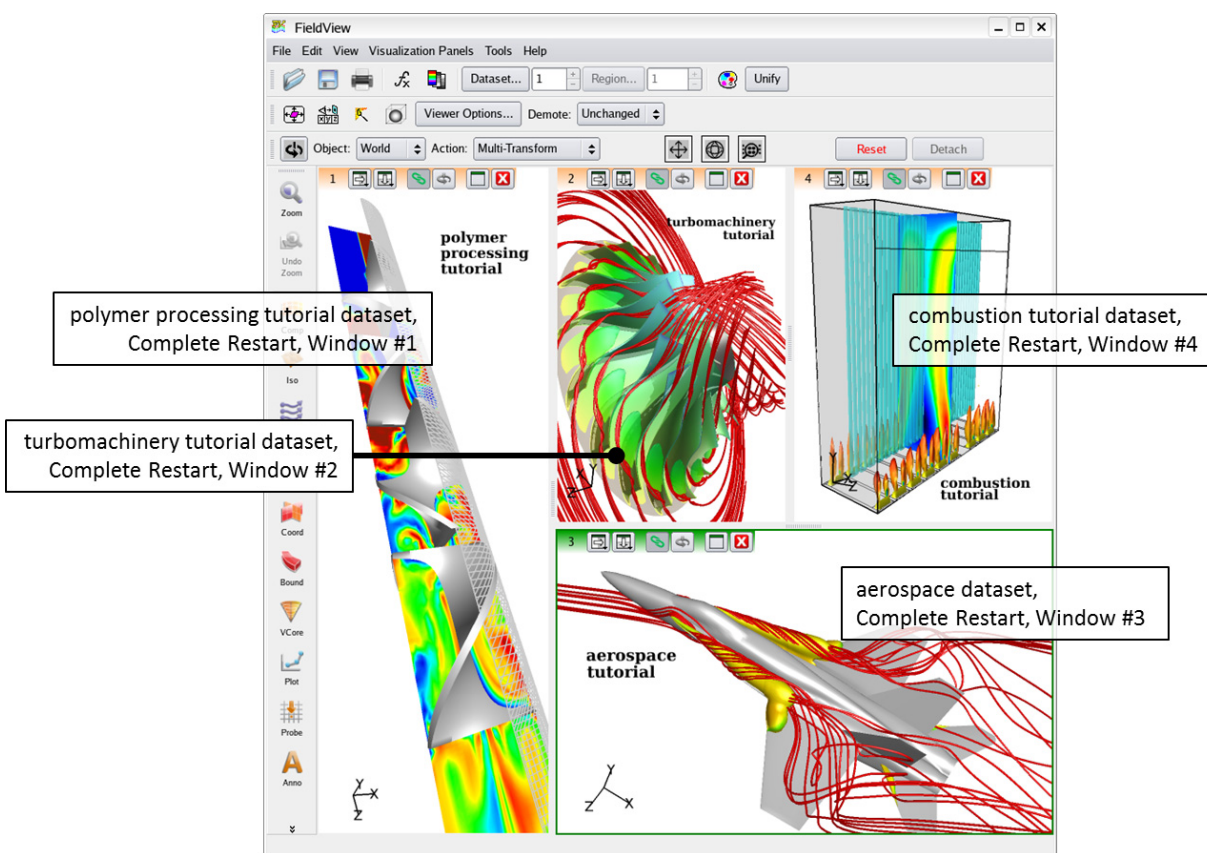


Figure 103 Complete Restarts for some Tutorial Datasets

Important Note: Dataset numbering is window relative. In the preceding illustration, we have four datasets in total. However, in each window, the dataset number for each of the cases is equal to 1.

Complete Restart, No Data Read

Complete Restart, No Data Read is a read-only restart option that cannot be saved. This allows a Complete Restart saved for one dataset to be read and used with a *different* dataset to create the “same” Computational, Coordinate and Iso-Surfaces and stream-line rakes. It gives the user an ability to create visualizations for one dataset, and then use them with a different dataset without recreating all of the surfaces and rakes for the new dataset. The new dataset may merely be a different solver run on the same grid, a run on a scaled-up or scaled-down version of the grid or may involve a completely different set of grids and results.

This particular restart is meant to be used in the following fashion:

1. Read in the first dataset.
2. Make the visualization you want.
3. Save a Complete Restart.
4. Read in a different dataset.
5. Read a Complete Restart, No Data Read which will attempt to create the same surfaces and rakes (listed above) that existed in the first dataset on the second dataset.

This restart fully reads all files associated with the `complete-restart-name` with the exception of the Data Input restart (`.dat`), for which **FieldView** will only obtain dataset orientation, duplication, scaling, region visibility, and clipping parameters. In addition, any formulas defined in the Formula Restart (`.frm`) will be appended to the current set of formulas.

A benefit of this restart is that an existing visualization can be used with different datasets. It can be applied repeatedly to a sequence of datasets to generate a series of illustrations (pictures and/or 2D plots) having identical properties (i.e. same surfaces, with the same scalar ranges and legends shown).

If multiple windows are present, this Complete, No Data Read restart will be applied to the dataset(s) in the current window only. The multi-window layout will not be affected.

Current Dataset Restart

The Current Dataset Restart reads or saves a set of restart files for the current dataset only. It lets you apply restarts from one dataset among multiple datasets in a **FieldView** session, to another dataset, in that same session. So, for example, you can read in several similar datasets, create the desired visualization on the first dataset, and then make that same visualization on each of the remaining datasets in **FieldView**.

A Current Dataset Restart can be written for one dataset in one window and read onto another dataset in a different window. This makes it possible for you to create your desired visualization on any dataset within one window and then apply it to any dataset in another window.

The Current Dataset Restart reads or saves only the following component restarts:

- Data File (`.dat`)
- Computational Surface (`.cmp`)
- Boundary Surface (`.bnd`)
- Coordinate Surface (`.crd`)
- Iso-Surface (`.iso`)
- Streamlines (`.stl`)
- Particle Paths (`.par`)
- Feature Extraction (`.vtx`) (Vortex Cores / Surface Flows)

When a Current Dataset Restart is saved, it saves information about the current dataset only, regardless of the number of datasets in memory. The `.dat` file only contains information for the current dataset, and component files will only be written for objects that exist on the current dataset. For example, if there are no Boundary Surfaces present when a Current Dataset restart is saved, then no Boundary Surface restart will be saved.

When a Current Dataset Restart is read, datasets are *not* reloaded, nor are any dataset viewing transforms that may be in the file. First, all objects on the current dataset are deleted, even objects not part of a Current Dataset Restart, such as 2D Plots. Then, only the duplication and region parameters from the Data File component restart (`.dat`) are read, and the objects in the restart are created on the current dataset.

The Current Dataset Restart differs from the Complete Restart in the way surfaces are retained or deleted. The Current Dataset Restart applies to objects on the current dataset only, while preserving objects created on other datasets. By contrast, reading a Complete Restart deletes all surfaces/rakes on all datasets, then applies the contents of the restart.

Objects not connected with datasets which are supported by "global" restarts will not be affected when reading a Current Dataset Restart, including:

- Annotation objects (titles or arrows) (`.ttl`)
- Formulas (`.frm`)
- Colormaps (`.map`)
- View (World) (`.vct`)
- Presentation Rendering Settings (`.prd`)

See [Restart Flexibility on page 262](#) for more information.

Multi-Window Layout...

The Multi-Window Layout Restart lets you save your arrangement of multiple windows. This allows you to easily re-use a previously created layout without also having to read the data that was saved with the layout.

Within the Multi-Window Layout Restart, the size and placement of each window is stored. The window attributes (see [Multi-Window Operation](#) in **Working with Field-View**) are also stored for each window. The Multi-Window Layout Restart also stores context which associates datasets, based on the order in which they have been read or created via a Copy action, with specific windows.

The Multi-Window Layout restart is written when saving a Complete Restart, and is automatically read when opening a Complete Restart.

Layouts are not saved as part of your **FieldView** preferences on exit. As a result, **FieldView** will continue to start with a single window unless a Complete Restart is read at start-up.

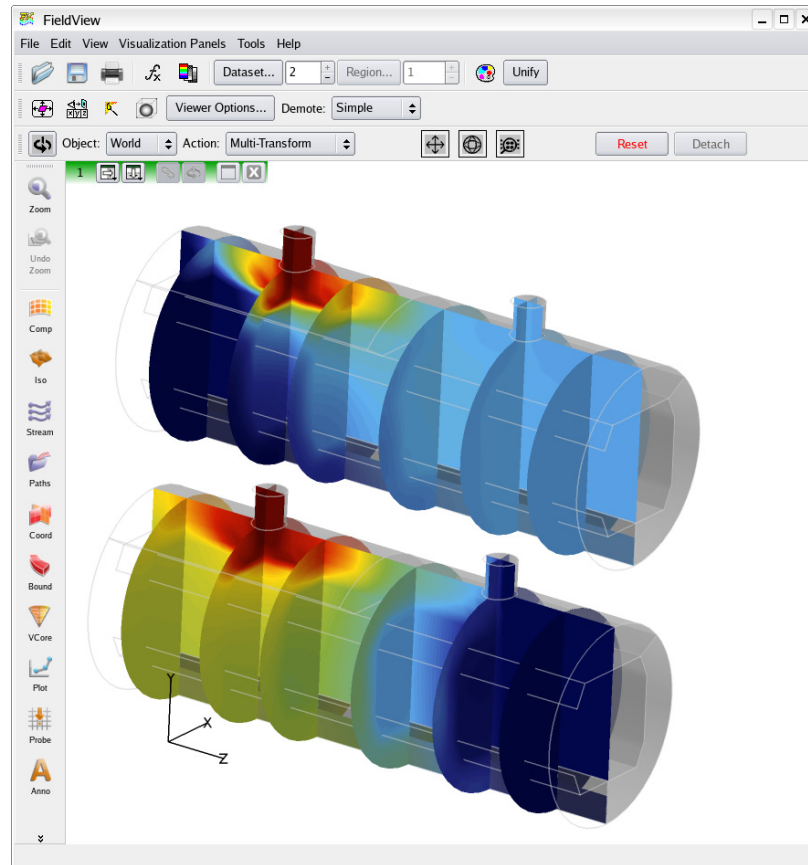


Figure 104 Two datasets in a single window

To see how a Multi-Window Layout restart works, consider the starting example in [Figure 104](#) above. To re-distribute the two datasets into different windows, we need to read a Multi-Window Layout restart that contains two windows.

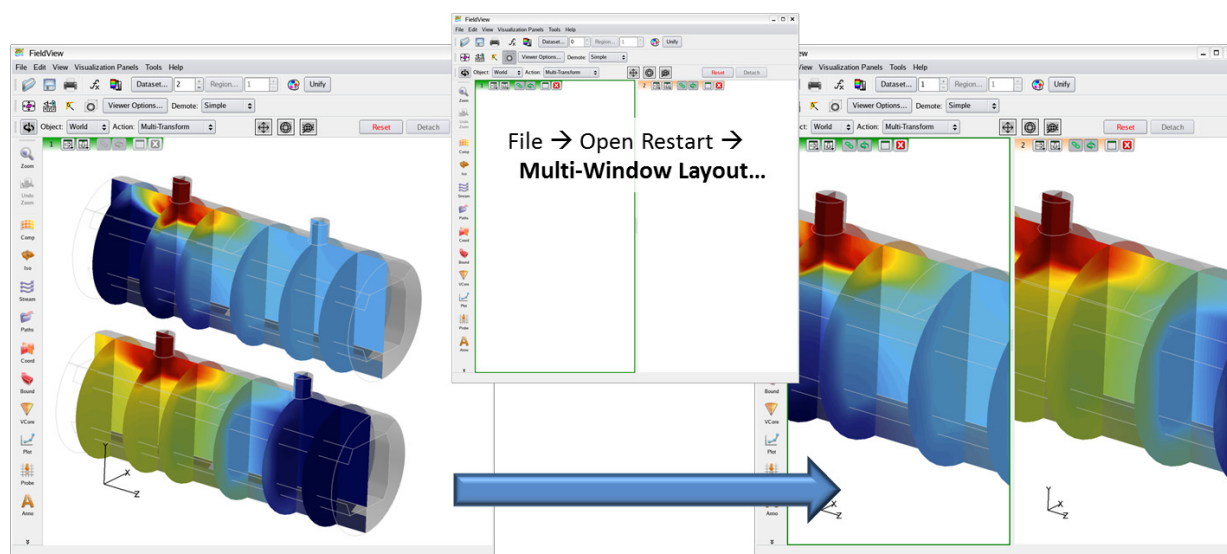


Figure 105 Applying a Multi-Window Layout Restart

In [Figure 105](#) above, a previously saved Multi-Window Layout describing two windows, side by side in a vertical orientation is used to re-distribute the two starting datasets. When a re-distribution of datasets is going to occur as a result of reading a Multi-Window Layout Restart, the following warning (see [Figure 106](#)) is displayed giving you the opportunity to cancel the operation.

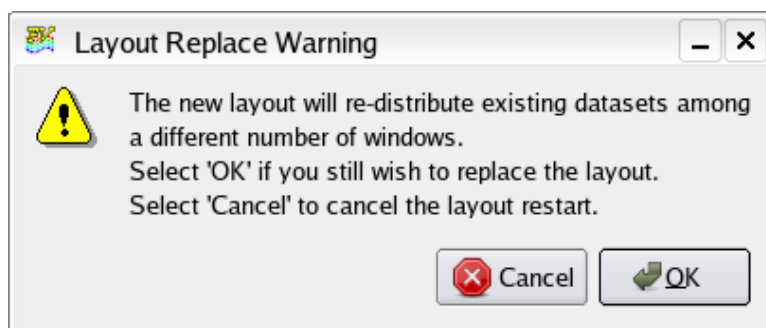


Figure 106 Layout Replace Warning

To reposition the datasets into a more suitable layout, such as a top and bottom horizontal orientation, another Multi-Window Layout Restart can be applied as shown in [Figure 107](#).

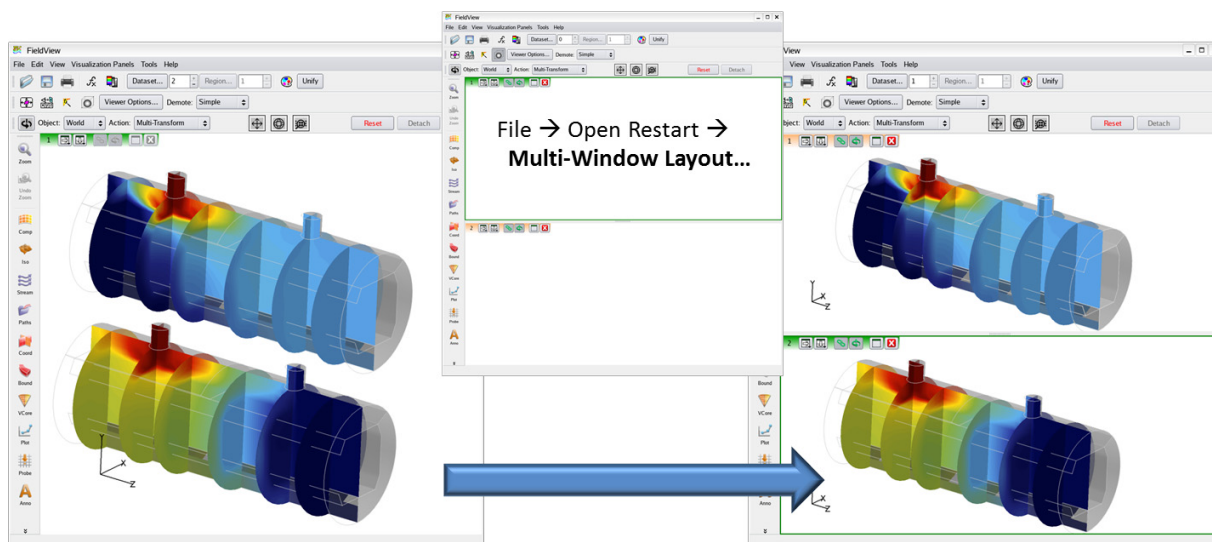


Figure 107 Number of Datasets equals Number of Windows

The preceding two cases illustrate how datasets are re-distributed when the number of datasets is equal to the number of windows. When the number of datasets is greater than the number of windows, datasets are distributed one per window until the last window is reached. At that point, all remaining datasets are placed into the last window, as illustrated in [Figure 108](#).

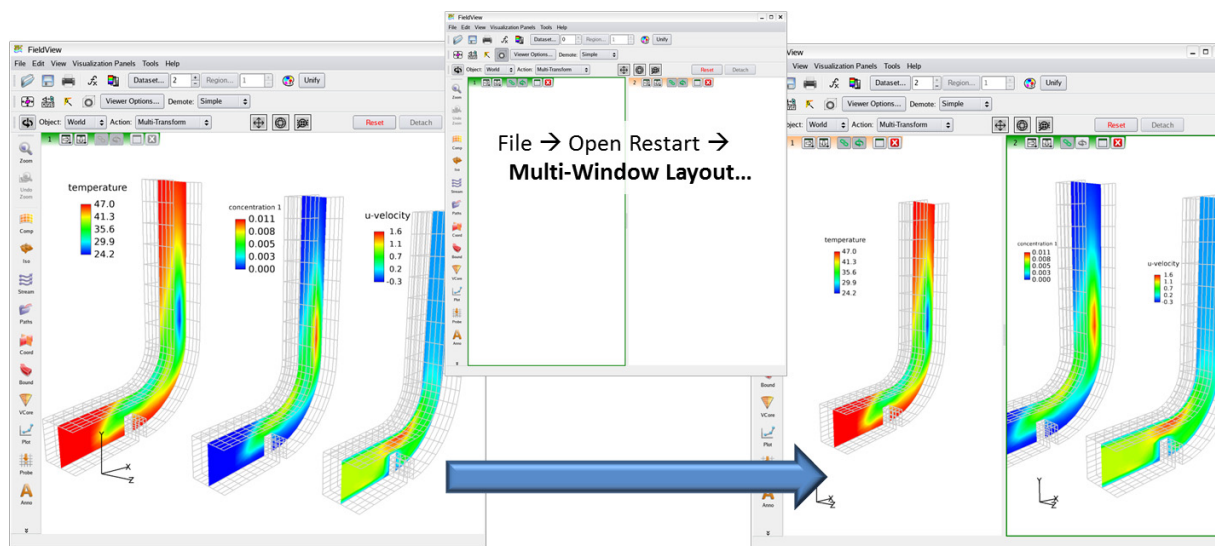


Figure 108 Number of Datasets greater than Number of Windows

In this example, the datasets sent to the last window will need to be transformed (at the Dataset level), since they will be placed one on top of the other.

When the number of datasets is less than the number of windows, datasets are simply distributed one per window and some windows in the layout will be empty, as illustrated in [Figure 109](#).

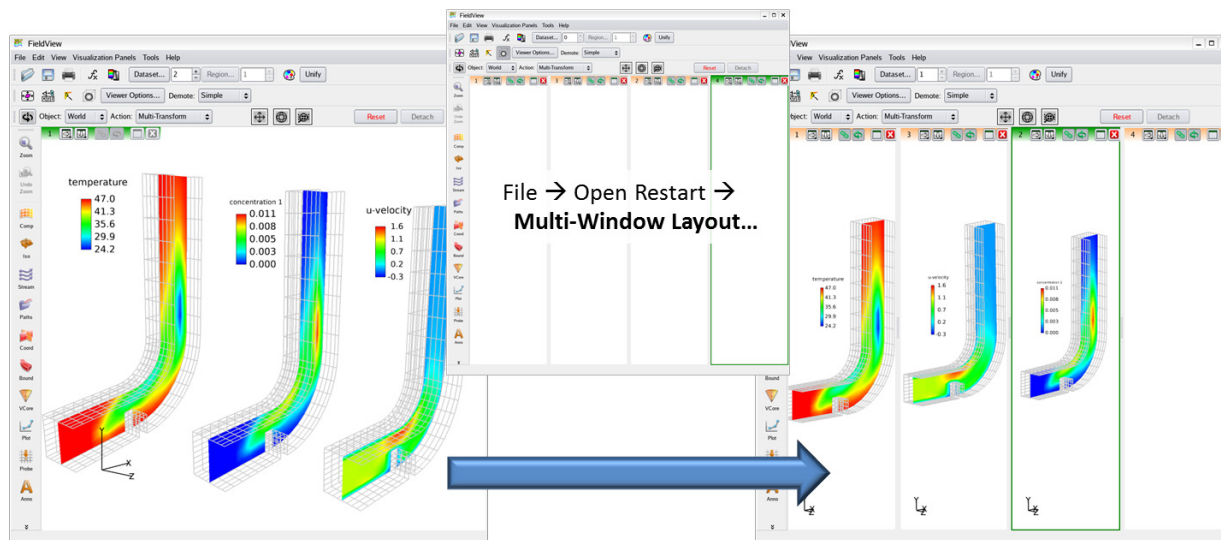


Figure 109 Number of Datasets less than Number of Windows

Layout Restart Files

Starting with **FieldView 14**, **FieldView** has the ability to display multiple windows. The number and relative size of these windows can be saved to a Multi-Window Layout Restart file. This layout can then be applied to a **FieldView** session by opening the corresponding Restart.

Since defining complex layouts can be tiresome, especially when trying multiple windows of the exact same size, the **FieldView** installation directory now provides a number of default Layout Restart files, under

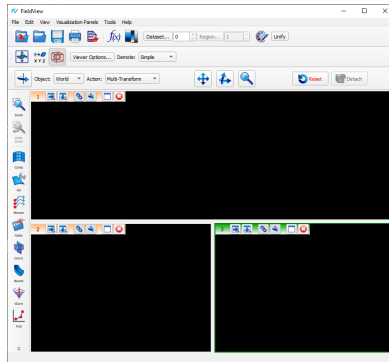
```
$FV_HOME/fvx_and_restarts/layout_restarts
```

These Restart files can be applied to a **FieldView** session through the following menu:

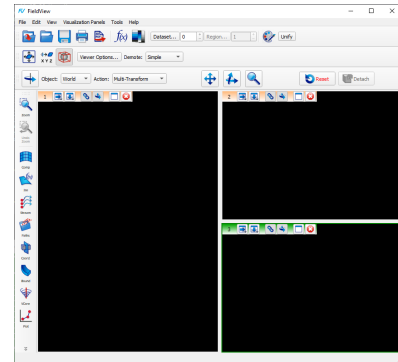
File > Open Restart > Multi-Window Layout

The following table summarizes available Layout Restart files.

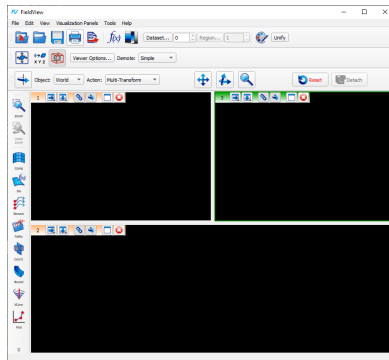
1 x 2h



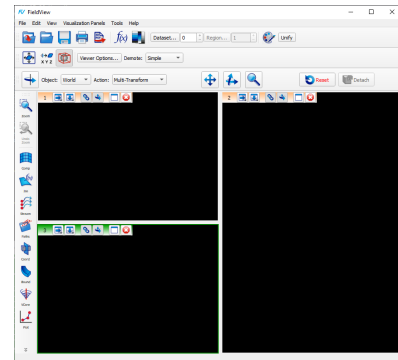
1 x 2v



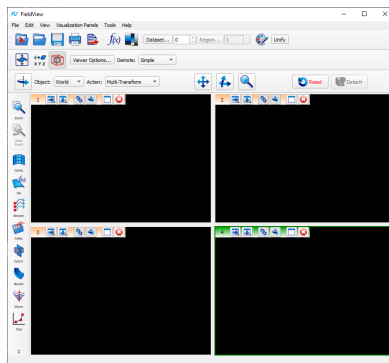
2h x 1



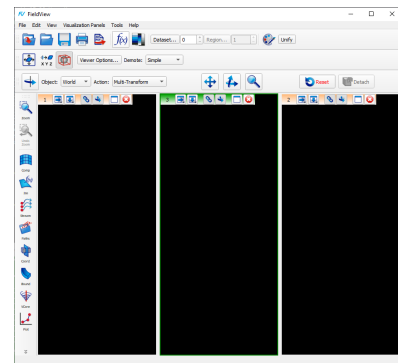
2v x 1



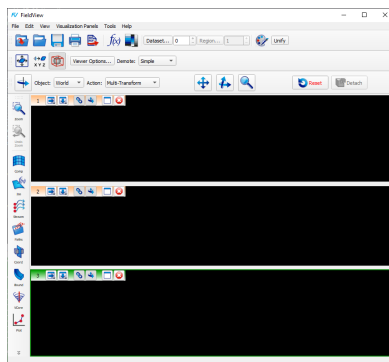
2 x 2



3 horiz.



3 vert.



Preference Restart

The Preference Restart reads or saves a set of five restart files:

- View (World) (`fv.vct`)
- Colormap (`fv.map`)
- Surface Plot (`fv.plt`)
- Formula (`fv.frm`)

Preference Restarts are written to the current directory using a default filename of `fv` to match the default behavior of Automatic Restart (see [Preference Restart on page 12](#) of the **User's Guide** for information on where they can be placed to load **automatically**.)

If a Preference Restart file contains information irrelevant to the current dataset, it will be ignored, usually silently. For example, if the Surface Plot current slice value is `I=16` and the current dataset has an `Imax=10`, then the Surface Plot slice value will be its default value of 1, not 16.

Script Restart

The Script Restart option is used to execute files containing Script Language commands. See the section [FieldView Script Language Commands on page 280](#) for details.

Formula Restart

The Formula Restart will save all formulas that have been defined by the Function Formula Specification panel. The names of the functions that were used in the formula will be saved exactly, and the formula will only be displayed when all of the necessary information is present to calculate it. For example, a formula created using the variable `Pressure` only appears when the variable `Pressure` is defined. This allows the Formula Restart to be used as a library of previously defined formulas that only have to be created once.

If the Formula Restart file contains formulas which require variables which do not exist in the current dataset, the creation of those formulas will fail silently and will not exist in the Function Selection panel.

When a Formula Restart is read in, if other formulas already exist in memory, you will be given the option of appending the new formulas to the existing ones, or replacing the formulas in memory with the ones from the file.

Note that formulas must have unique names. Thus when reading in a Formula Restart file, if a name in that file matches a name that already exists (from a solver file, for example), the formula in the restart file will be skipped, and a warning message will be issued.

The format of a Formula Restart is as follows:

The first line is a header that should read:

```
formula_restart_version: 1
```

Next the formulas are defined in two lines. The first line is the name of the formula, and the second line is the formula itself. Any math functions used will appear in the Formula Restart exactly as they appear on the Formula Specification panel. Any function names used will appear in double quotes. For example, a formula named Velocity calculated as Density divided by Momentum would appear in the restart as:

```
Velocity  
"Density"/"Momentum"
```

A function named Vorticity equal to the curl of the Velocity would appear as follows:

```
Vorticity  
curl("Velocity")
```

Data File Input

The Data File Input Restart saves all information of datasets, including transformations (the result of any Rotate / Zoom / Translate operations) for each dataset and any Duplication information (mirroring or rotating). The Visibility of each dataset is NOT saved.

The Data File Input Restart File includes the Function File pathname, the subsetting information, and the names decided on for each selected variable (including vector names). It does not include the name file pathname, since the names are stored directly in the restart file. Thus, changes to the Name File after the Restart File is created will have no effect on the restart.



Note: **FieldView** expects all data files referenced to have the identical pathnames as when the restart was saved.

Note: If Data Restart files are hand-edited and the path removed from the filename field, **FieldView** will not properly recognize a transient dataset upon read-in.

Computational Surface

The Computational Surface Restart File saves all Computational Surfaces across all datasets. It includes information about each surface and any associated legends.

Iso-Surface

The Iso-Surface Restart File saves all Iso-Surfaces across all datasets. It includes information about each surface and any associated legends.

Streamlines

The Streamlines Restart File saves all rakes. It includes information about each rake and any associated legends. A Streamline Restart can also be used as a template for placing seeds at specific `IJK` or `XYZ` locations, described as follows:

Streamline Template

This procedure uses the Streamlines Panel (see [Chapter 6 of Working with FieldView](#)) to create 'dummy' seeds, which are saved in a Streamline Restart file and edited to the desired locations.

1. With a dataset in memory, create a Streamline rake with a couple of seeds. Use either `IJK Int` or `XYZ` for the `SEED COORDINATES` (this technique is less effective in `IJK Real` mode).
2. Save a Streamline Restart.
3. Open the Streamline Restart (`*.stl`) file in your favorite editor. This step and the next can be done in a program outside of **FieldView** in any number of languages.
4. Replace the current seeds with the desired seed positions, using the same format.
5. Read the new Streamline Restart into **FieldView**. This will replace the current rake (with the few 'dummy' seeds) with the restart file seed positions.

Assuming the same dataset is used to create the 'dummy' template and read the edited restart back in, the streamline seeds should appear at the desired locations.

There are several portions of the Streamline Restart file that need to change: the `total_seeds` value near the top of the file (this is the total number of all seeds in all rakes, summed up), information about the number of seeds (`num_seeds`), and the seed positions. The only seed position information that is required for `IJK Int` seeding is the grid number and the `IJK` values. The `XYZ` position information is defaulted (with an `"*"`). For `XYZ` seeding, the opposite is true.

Note: Using an invalid Streamline Restart file may result in *no* seeds being shown. For example, using a Streamline Restart file with only `IJK Int` seeding on an *unstructured* dataset will result in the seeds being *silently* thrown away and no seeds or streamlines will be produced.

Example: `IJK Int` Seeding

The following shows the portion of a Streamline Restart file made for the F18 dataset (used in the `/demo` script and `/aerospace` tutorial). Three randomly placed seeds were created on Grid #3 on an `I=14` Computational Surface. This created a Streamline Restart file from which only the relevant portion to edit is shown:

```
num_seeds: 3
seed_ijk: 3 14.000000 9.000000 28.000000
seed_xyz: *
seed_ijk: 3 14.000000 9.000000 25.000000
```

```
seed_xyz: *
seed_ijk: 3 14.000000 10.000000 23.000000
seed_xyz: *
```

Merely edit the `IJK` values to those desired, increasing the `num_seeds` value if additional seeds are needed. For example:

```
num_seeds: 5
seed_ijk: 3 14 3 2
seed_xyz: *
seed_ijk: 3 14 3 4
seed_xyz: *
seed_ijk: 3 14 3 6
seed_xyz: *
seed_ijk: 3 14 3 8
seed_xyz: *
seed_ijk: 3 14 3 10
seed_xyz: *
```

The format of the numbers is not important, nor is their spacing. The only requirement is that the `IJK` or `XYZ` values have at least one (1) space between each value.

Example: `XYZ Seeding`

The following shows the portion of a Streamline Restart file made for the F18 dataset (used in the `/demo` script and `/aerospace` tutorial). Three randomly placed seeds were created on Grid #3 on an `X=1.5` Coordinate Surface. This created a Streamline Restart file from which only the relevant portion to edit is shown:

```
num_seeds: 3
seed_ijk: *
seed_xyz: 1.50000000e+000 9.92904082e-002 3.32836539e-001
seed_ijk: *
seed_xyz: 1.50000000e+000 7.64748678e-002 4.49470043e-001
seed_ijk: *
seed_xyz: 1.50000000e+000 1.11091778e-001 6.22056007e-001
```

Merely edit the `IJK` values to those desired, increasing the `num_seeds` value if additional seeds are needed. For example:

```
num_seeds: 4
seed_ijk: *
seed_xyz: 1.5 0.01 0.33
seed_ijk: *
seed_xyz: 1.5 0.02 0.43
seed_ijk: *
```

```
seed_xyz: 1.5 0.03 0.53
seed_ijk: *
seed_xyz: 1.5 0.04 0.63
```

The format of the numbers is not important, nor is their spacing. The only requirement is that the `IJK` or `XYZ` values have at least one (1) space between each value.

Particle Paths

The Particle Paths Restart File saves all of the paths information. It includes information about each group of paths and any associated legends.

Annotation

The Annotation Restart File saves all titles and arrows. It includes information about each title and arrow such as position, font, color, size, etc.

View (World)

The View (World) Restart File saves the current World transform specification. This file includes the current settings for Rotate, Translation, and Zoom for the Object World. It also includes information about Outline, Perspective and the axis marker. Note: All Dataset transforms are stored in the Data File Input Restart File.

Colormap Specification

The Colormap Specification Restart File saves the current settings for Geometric Color, Background Color, Colormap, Filled Contour, Invert, and the RGB values of the 8 editable geometric colors. If the Colormap setting is User Defined, the pathname of your colormap file will be present.

FieldView stores information about the 10 geometric colors in the Scalar Colormap file. In addition to the geometric color (`geom_color`) and background color (`back_color`) entries in the Scalar Colormap file, each surface (`*.bnd`, `*.crd`, `*.cmp`, `*.iso`), rake (`*.stl`, `*.par`), feature extraction object (`*.vtx`) and annotation (`*.ttl`) restart file contains a color entry followed by the color index. The default color indices and their associated colors are given in the following table. Note that color indices 1 and 2 (for black and white, respectively) *cannot* be changed by the user. The other color indices can be assigned to different colors using the Color Mixer feature (see [Chapter 14](#) of **Working with FieldView**). The actual Red, Green, Blue (RGB) values associated with each of the 8 editable geometric colors is given in the Scalar Colormap file.

1	2	3	4	5
black	white	red	green	blue
6	7	8	9	10

cyan

magenta

yellow

purple

gray

Surface Plot

This restart file saves the setting from the Surface Plot sub-panel on the Computational Surface panel.

Boundary Surface

This restart file saves all boundary surfaces. It includes information about each surface and any associated legends.

Vortex Cores / Surface Flows

This restart file saves all vortex core and surface flow information. It does not include shock surface information, which is a type of Iso-Surface and saved in the Iso-Surface restart file. Vortex Cores / Surface Flows restart behaviors include the following:

1. Target boundary surface(s) #1, #2, etc. for a given Surface Flow restart must exist, otherwise the Surface Flow will fail to be re-created, and **FieldView** will print:

```
Dataset 1, Object 1 (Surface Restricted Flow: No Slip) created,  
but some or all of the dependent boundary surfaces do not exist.
```

2. If the target surface(s) exists, but has changed extent (thresholding, or number of boundary types), the Surface Flow will re-calculate based on the new settings. This is not true for Dynamic Clipping; if performed before the restart is loaded, the clip has no effect on the Surface Flow lines.
3. If the target boundary surface(s) has Visibility=off, it will still be used to re-calculate the Surface Flow.

Coordinate Surface

This restart file saves all Coordinate Surfaces. It includes information about each surface and any associated legends.

2D Plots Restart

This restart file saves all information from the 2D Plot panel. When this restart is read back in, the 2D Plot panel becomes the current visualization panel. **FieldView** behavior has changed when reading Complete Restarts from earlier versions. If the Complete Restart includes any 2D Plot specifications the 2D Plot panel will become the *current* visualization panel while reading the Complete Restart. Since curve plots are dependent on surfaces this assures that the surface exists prior to recalculating the plot. If the Complete Restart does not specify any 2D Plots, **FieldView** behavior is unchanged.

Point Probe Input

This option allows the user to read in a file (.csv, .txt, or .mat format) containing a series of XYZ values and have **FieldView** output a file (using the same format) repeating those coordinates, as well as the value of the current scalar/vector functions at each coordinate. This feature is also available via **Tools.. Point Query**. This feature is further described in section [Point Query...](#) in [Chapter 14](#) of **Working with FieldView**.

If the Point Probe panel is up when saving a Complete Restart, a "hidden" Point Probe file will be created and read with the Complete or Complete, No Data Read Restart. The hidden file cannot be restarted individually; it exists so **FieldView** can properly load registers on the Point Probe panel during transient sequences.

The Point Probe panel is documented in [Chapter 13](#) of **Working with FieldView**.

Presentation Render

This restart file saves all current information about Presentation Rendering. It includes information about surfaces and rake types that have Presentation Rendering turned ON, plus values for the Highlight Size and Intensity parameters.

Clip Groups

This restart file saves the properties of all clip groups created in the current **FieldView** session. The clip lines and/or boxes stored within each clip group are independent of the dataset on which they were created. Thus, the user can read-in and apply a Clip Groups Restart to any dataset in **FieldView**. For more information on Dynamic Clipping, see [Chapter 14](#) of **Working with FieldView**.

FieldView Script Language Commands

The Script Language feature allows you to save a text file to replay common **FieldView** operations. A useful set of commands is provided for better control of replays, such as PAUSE and SLEEP. The Script File can later be executed using the Script Restart option in the Restart Files menu or restarted automatically during **FieldView** startup with the -s command line switch (see [Chapter 1](#) of the **User's Guide**).

Running a Script

The Script filename must be saved with the extension .scr.

All command words may be in upper or lower case, or any mixture of these.

Leading and trailing blanks are ignored.

Lines beginning with an exclamation point (!) are considered to be comment lines and are ignored during execution.

Filenames may need to be quoted in some script commands. See descriptions of individual script commands for specific instances.

Conditions

FieldView will “beep” your display when script execution is completed or canceled.

Since a SCRIPT containing the command, PANELS OFF, makes it impossible to stop a long task in **FieldView**, a script abort has been provided. The `ESCAPE` key will be the abort “hot” key and will be active during the execution of any script. When the `ESCAPE` key is pressed, the user will be presented with a confirmation pop-up. If the abort is confirmed, the **FieldView** script will exit after the current script command finishes. If the `ESCAPE` key is pressed during an `INTERPOLATE` or `SWEEP` script command, these will completely finish before the script is exited. Note: If you have explicit pointer focus policy on your computer, you may have to click in the graphics window before the `ESC` key will have an effect.

Syntax

The **FieldView** Script Language includes the following commands (listed in alphabetical order):

```
3DPDF_WRITE filename
```

The script command `3DPDF_WRITE` exports the current window as a 3D PDF format file. For more details, see [3D PDF Write](#) in **Working with FieldView**.

ALIGN +X/+Y/+Z/-X/-Y/-Z

The first form of the `ALIGN` command takes a single viewing direction argument from one

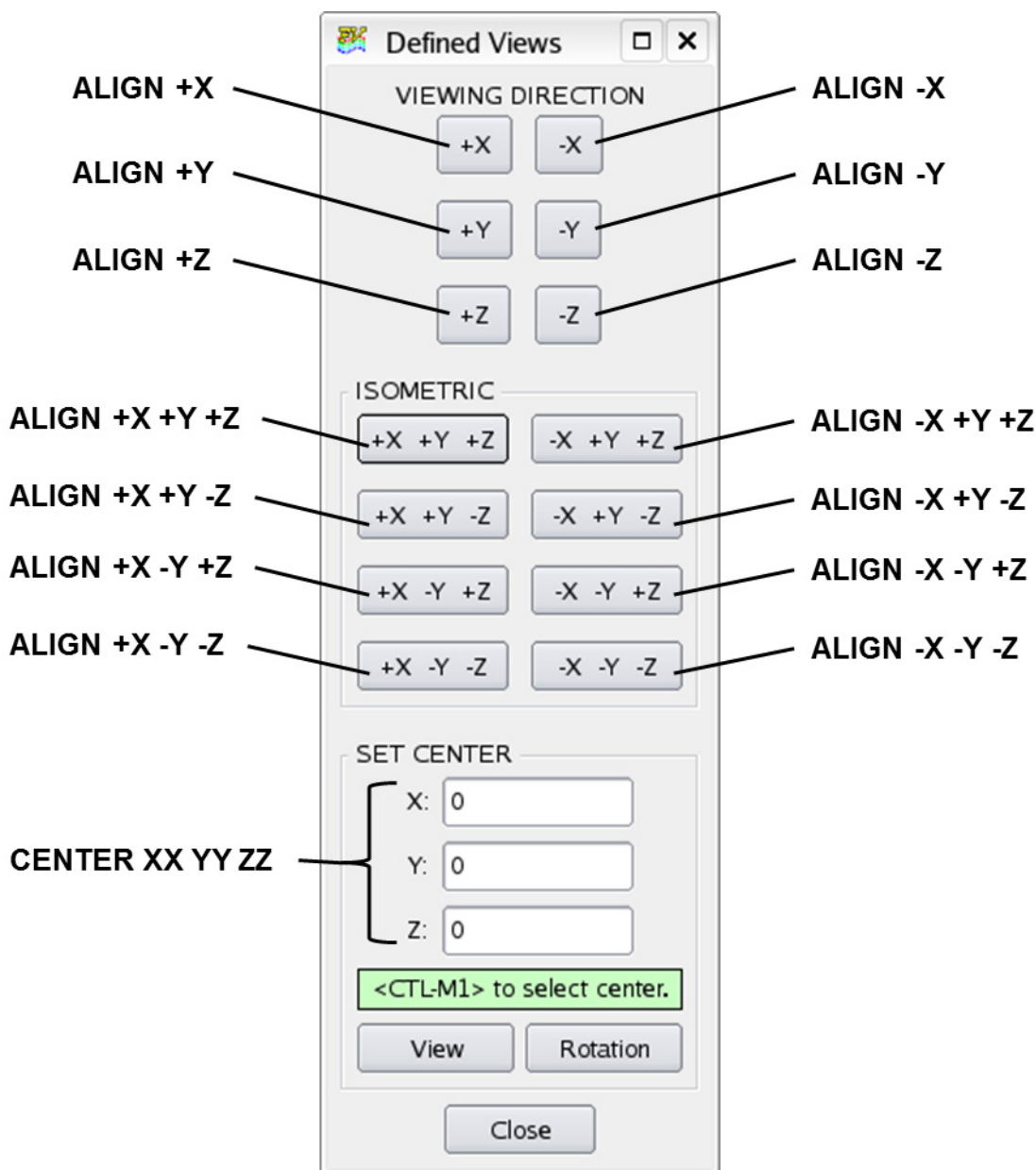


Figure 110 Script View Controls and matching GUI actions

of the six possible options. It will have the same effect as navigating to the Align... panel and choosing from one of the six available viewing directions.

The second form of the `ALIGN` command supports the 8 isometric views. There must be 3 arguments, specified in `XYZ` order. The 8 valid combinations are:

+X +Y +Z
+X +Y -Z
+X -Y +Z
+X -Y -Z
-X +Y +Z
-X +Y -Z
-X -Y +Z
-X -Y -Z

For example, the following script commands would generate errors:

```
ALIGN +X -X +Z  
ALIGN +Z +Y +X
```

Applying the script commands in the order of:

```
ALIGN +X  
ALIGN +Y  
ALIGN +Z
```

is *not* the same as the applying the isometric view `ALIGN +X, +Y, +Z`. This is consistent with what would happen if the user interacts with the GUI.

```
ANIMATE number-of-cycles
```

The script command `ANIMATE` allows scripted animation of curved vectors on a coordinate surface that has them. It can also animate particle paths and streamlines, just as `SWEEP number-of-cycles` does.

```
ANTIALIAS ON/OFF
```

Turns anti-aliasing `ON` or `OFF` when rendering in the graphics window.

```
AXIS_MARKER ON/OFF
```

When encountered in a script, this will set the state of the `AXIS_MARKER` toggle. If the `AXIS_MARKER` is `ON` already and the script command is issued to redundantly turn the `AXIS_MARKER ON`, then nothing will change. The script command is explicit and requires either `ON` or `OFF`.

To simply control the background color, the command is:

```
BACKGROUND color n
```

where *n* is either an index from 1–8 representing one of the user-definable colors in the color palette, or one of the strings, "white" or "black", corresponding to the fixed colors in the color palette. An illustration of the color number assignments is shown in **Figure 111 Summary of Image Background SCRIPT command**. The `BACKGROUND` script commands can be used to create a dynamic script or **FVX** program by applying a series of backgrounds to an animation or transient sequence.

The script command to permit the automated operation of selecting and applying a background image is:

```
BACKGROUND position-type filename
```

position-type is one of `STRETCH`, `CENTER`, `FIT` or `OFF`, and
filename is the full name of the background image file

Note: If *position-type* is `OFF`, *filename* should be omitted to avoid a script error.

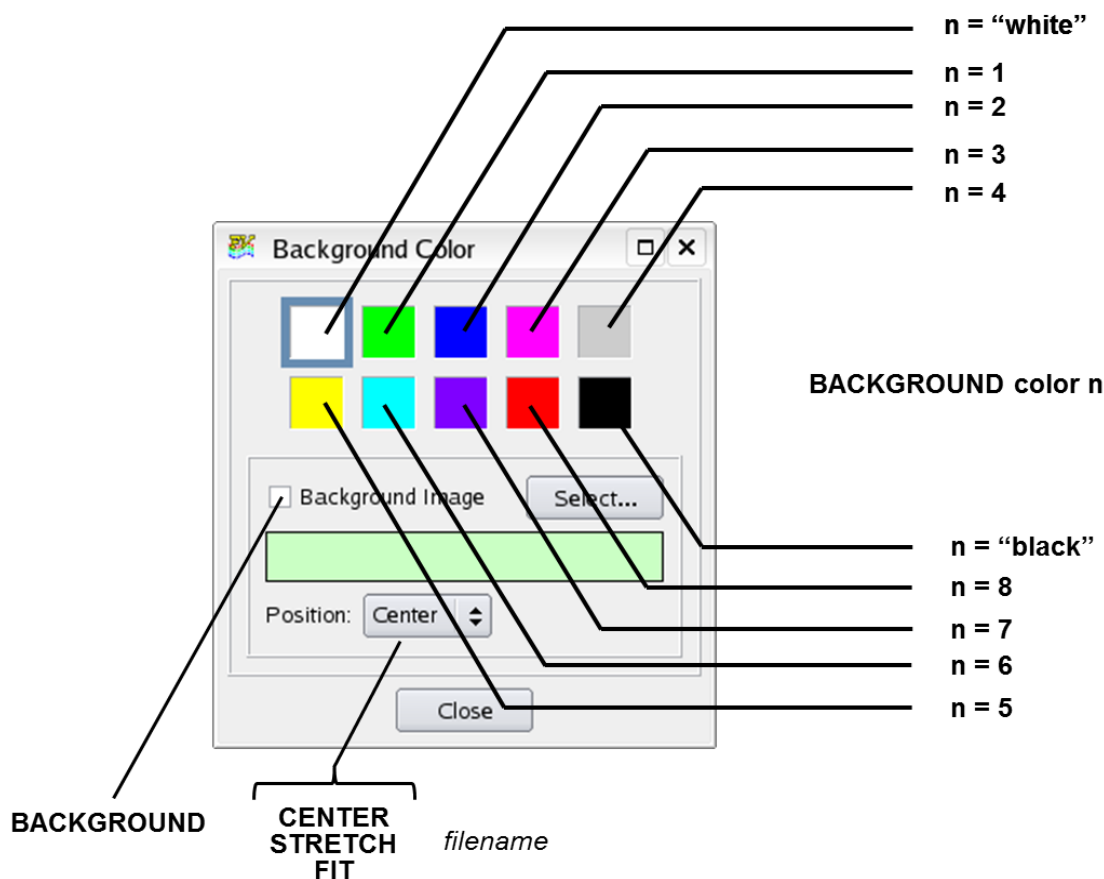


Figure 111 Summary of Image Background SCRIPT command

`CENTER [XX YY ZZ]`

When encountered in a script, this will apply the `CENTER` action at the Object: World Transform level. Executing this script command before any data has been read into **FieldView** will do nothing, and no warning error or message will be issued.

The arguments `XX`, `YY` and `ZZ` are optional and correspond to values that the user can set in the Align... panel (see **Figure 110 Script View Controls and matching GUI actions**) to specify a center of view which is different from the default of 0, 0, 0. If the script command is called with these arguments, it will be the same as if a user has navigated to the Align... panel, entered values in the `SET CENTER` GUI for X, Y and Z, and then pressed the View button.

The `CENTER` script command works at the World level of the transform hierarchy. At present, there is no support for centering at lower levels of the transform hierarchy such as Dataset, Region or Surface. **Note:** The `CENTER` script command (without arguments) will typically produce a result which would be different from the command *with* XYZ arguments, because it will center all visible objects on screen, not center around 0,0,0 World Coordinates.

The `CENTER` script command will produce a result which can be different from `CENTER 0 0 0`.

`DATASET_SAMPLING results_target_dataset_number sampled_dataset_name`

`results_target_dataset_number` = number (order) of Results Target
`sampled_dataset_name` = name of Sampled Dataset to be created

Use the command above to automate the process of creating Sampled Datasets.

`DEMOTION n [n]`

This command allows you to change the demotion mode. This may be useful if rotating very large datasets, etc. The syntax is:

<code>DEMOTION 0</code>	None
<code>DEMOTION 1</code>	Bbox
<code>DEMOTION 2 3</code>	Points (and Subset Number)
<code>DEMOTION 3</code>	Wireframe
<code>DEMOTION 4</code>	Simple

Due to the single-color-per-face display of face data, demotion of boundary surfaces with face data is handled differently. If `DEMOTION` is set for `Points` or `Wireframe` (mesh), then the demotion color will be white (if the background is black) or black (if the background is white).

```
DUPLICATION dataset-number NONE
```

Setting `DUPLICATION` to none, will turn off any current duplication settings which may be active for your Dataset.

```
DUPLICATION dataset-number MIRROR axis1 [axis2 [axis3]]
```

This command allows you to automate the mirroring of a dataset around any or all model axes. The axis can be specified as any one of X, Y or Z. There can be up to three axis fields specified. Any entries after three valid axes will be silently ignored.



Figure 112 Interactive Dataset Mirror Copy

If an axis is repeated, for example: `DUPLICATION 1 MIRROR X X Y`

It counts as a valid axis value, but is redundant. The above command will result in the dataset being mirrored around the X and Y axes.

```

DUPLICATION dataset-number TRANSLATE axis1 total_copies1 delta1
[axis2 total_copies2 delta2 [axis3 total_copies3 delta3]]

```

This command is used to perform automated translational copies of a dataset.

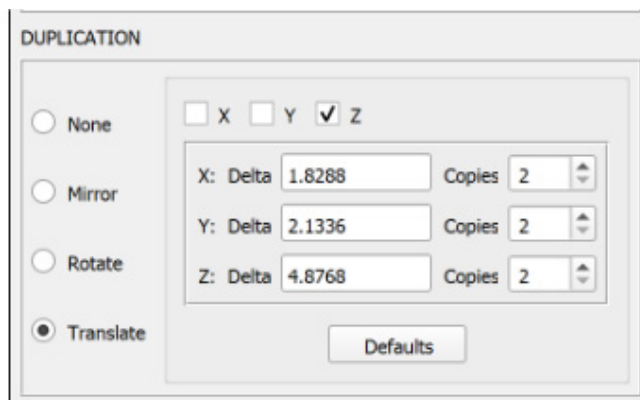


Figure 113 Interactive Dataset Translate Copy

One, two, or all three axis can be specified. If an axis is specified, total_copies and delta *must* also be set for that axis. The delta field will accept an asterisk '*' as a value in order to default to your CFD data's range for that axis.

```

DUPLICATION dataset-number ROTATE axis total_copies total_sweep

```

This command is used to perform an automated rotational copy of a dataset.

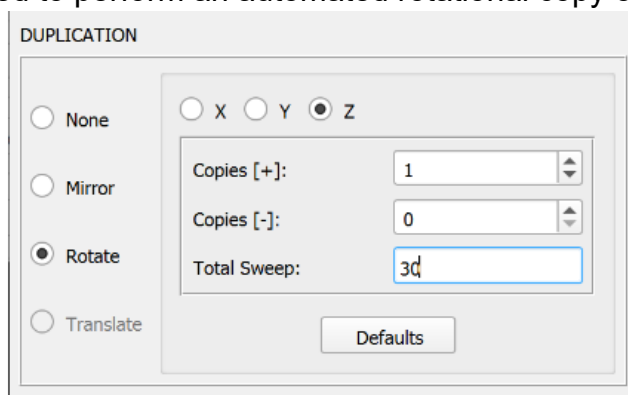


Figure 114 Interactive Dataset Rotational Copy

The axis argument is either X, Y or Z. The total number of copies is an integer number between 1 and 360, and the total sweep is a real number specifying the the total sweep range, in degrees, for a copy. Setting a total number of copies to 4, and a total sweep to 90 creates 4 copies spanning a total range of 360 degrees.

```
ENABLE_SWEEP_EXPORT [ TEXT | MAT-File | CSV ] COORD "filename"  
ENABLE_SWEEP_EXPORT [ TEXT | MAT-File | CSV ] BOUNDARY "filename"
```

This command (see [Transient Export...](#) in **Working with FieldView**) will affect the next `SWEEP TIME` command encountered and the surface will be exported at each step of the time sweep. Note: Putting the filename in double quotes is optional.

```
EXIT
```

This command will automatically exit **FieldView** without a confirmation.

```
EXPORT COMP filename  
EXPORT ISO filename  
EXPORT [MAT-File | CSV] COORD filename  
EXPORT [MAT-File | CSV] BOUNDARY filename  
EXPORT STREAM filename  
EXPORT PLOT filename  
EXPORT VCORE filename
```

`EXPORT` only works on the current object of the current dataset. In order to `EXPORT` the desired object, it may be necessary to first use the `SELECT DATASET` and the `SELECT` commands for the corresponding object, to be sure the object of interest is the one which is exported. These `SELECT` commands are described below.



Note: Putting the `filename` that appears in the `EXPORT` command in double quotes is optional.

Note: There is no script support for exporting a Surface Plot with this `EXPORT` command.

These commands will export information from the current object into the given output file. If the specified `filename` exists, it will be overwritten .

If the optional format argument `MAT-File` is specified for a supported object and `"filename"` does not include the extension `.mat`, `.mat` will be appended. If the optional format argument `CSV` is specified for a supported object and `"filename"` does not include the extension `.csv`, `.csv` will be appended. If a format option is not specified, the current object will be exported as a file without an extension, as in previous versions of **FieldView**. For more information on the `EXPORT` feature, see the `EXPORT` section of [Chapter 14](#) of **Working with FieldView**. For the format of the exported file, see [Appendix I](#) of this **Reference Manual**.

If there is no current object, no error message will be displayed, and no file will be created. Errors will be reported opening/writing the specified file and if a format argument is given for an unsupported object.



Cylindrical Note: When cylindrical coordinates are specified using an `FVREG` file (Region definition), the `XYZ` labels (and values) become `RTZ` (Radius, Theta, Z) labels and values for Iso-Surfaces, Coordinate Surfaces and Boundary Surfaces. Computational Surface exports are unaffected. See [Chapter 3](#) for more information.

FIT

The script command `FIT` takes no arguments and operates on all 3D data objects with visibility on. `FIT` adjusts the World transform by applying a `CENTER` command, then zooming and translating to best fit the current window. This should result in no more than a small percentage of blank space on each side of the visible objects along the tightest direction, vertical or horizontal, of the current window.



Note: The `FIT` command uses an iterative process to adjust the view, which takes more time than the simpler `CENTER` command. When writing scripts, it is recommended to only call the `FIT` command when an update to the scene requires it.

In the event that the `FIT` operation fails, a warning is issued and a default view is set.

```
INTEGRATE [current | all | sweep] filename
```

If the specified `filename` already exists, the information will be appended to that file. When an append is done, two blank lines will be inserted before the new output.

The file format for the script command `INTEGRATE sweep` is the same as if it had been done interactively.

The file format for the `current` and `all` options is:

```
Integral of S = [function name] on [current <type> surface |
all surfaces]
<blank line>
Integral      Surface Area    Average Value
<value>      <value>         <value>
```

Face Data

If the argument is `current` and face data exists for the current boundary surface being integrated, then the file format becomes:

```

Integral of S = [function name], V = [function name] on [current boundary surface]:
<blank line>
Integral   Surface Area   Average Value   Int(S*Nx)   Int(S*Ny)
<value>   <value>        <value>         <value>     <value>
Int(S*Nz) Int(V dot N)
<value>   <value>

```

Notes:



The first line will omit “, V = [function name]” if surface normal direction information is not available, or if the vector register is empty.

Columns 4-7 will be omitted if surface normal direction information is not available.

Column 7 will be omitted if the vector register is empty.

Conditions that require confirmation interactively will assume confirmation when the script command is executed (for example, the `subset increment is > 1` confirmation).

Error conditions will be reported. For example, if an `INTEGRATE` is attempted and the current surface has no Scalar Function defined for it, an error pop-up will be produced.

The Integration Controls panel will *not* be displayed when an `INTEGRATE` script command is executed.



Note: This script command does not support transient sweep integration.

```
INTERPOLATE number-of-steps view-restart-file
```

This command interpolates between the current view and the view defined in the specified Transform Controls (World) restart file, `view-restart-file`, where `number-of-steps` is a positive integer specifying the number of interpolation steps. For example, if `number-of-steps` is 2, for the first step the view will change to one that is halfway between the current view and the view in `view-restart-file`, and for the second

step the view will change to the view in `view-restart-file`. The name of `view-restart-file` may include full pathnames, but wildcards are not expanded.

Note: The script `INTERPOLATE` command will transition between views without obeying the specified center of rotation if this was changed by using the Set Center of Rotation controls (and subsequently saved in restart files used with the `INTERPOLATE` command).

```
KEYFRAME [start-frame end-frame [frame-increment]] filename
```

The `KEYFRAME` script command accepts optional `start-frame`, `end-frame` and `frame-increment` arguments. Batch mode also supports the keyframe restart, so a keyframe animation can be made in this fashion. Using multiple `KEYFRAME` commands in a single script will allow you to assemble separate animation scenes into a complex whole.

Note that the `ESC` key functions as an abort command which takes place *after* the current script command is finished. Pressing the `ESC` key during script execution of a keyframe restart will stop the script *after* the keyframe is finished, meaning that the animation will play through completely before aborting.

```
LIGHTINGVALUES ambient-light diffuse-light
```

The `LIGHTINGVALUES` script command provides the option to change the lighting parameters. Both input parameters are required and have a valid range of 0 to 1. Values of 0.3 for `ambient-light` and 0.7 for `diffuse-light` correspond to the **FieldView** defaults.

```
LINKED_SURFACE_SWEEP ON/OFF
```

The script command `LINKED_SURFACE_SWEEP ON` changes the mode of **FieldView** to permit linked surface sweeping. A script command following this command to sweep a surface will sweep all like surfaces. To select a particular surface to be used as the control surface for linked sweeping, use the `SELECT` script command.

```
MAXIMIZE ON/OFF
```

The script command `MAXIMIZE ON` makes the graphics window full screen size as if the user had pressed the maximize button in the window frame.

The script command `MAXIMIZE OFF` returns the graphics window to its original size (prior to the previous maximize command). If there was no previous `MAXIMIZE ON`

`command`, the `MAXIMIZE OFF` command will be ignored. See also the script command `PANELS ON/OFF`.

`OUTLINE ON/OFF`

When encountered in a script, this sets the state of the `OUTLINE` toggle. If `OUTLINE` is already `ON` and the script command is issued to redundantly turn `OUTLINE ON`, nothing will change. The command is explicit and requires either `ON` or `OFF`.

`PANELS ON/OFF`

The script command `PANELS OFF` will shut off any subsequent display or updating of panels in **FieldView** for the remainder of the script or until a `PANELS ON` command. It will do this by removing them. This will help ensure that a full screen graphics window will not be covered up by a **FieldView** panel at any time during the script.

In a `PANELS OFF` state, the `PLOT` and `RESTART PLOT` commands display the Plot Display, overlaying the graphics window if it is maximized. The Plot Controls panel, however, will not appear. Otherwise, script commands which normally cause a panel to appear will do everything they normally do except make the panel appear. Panels that are open when the `PANELS OFF` command is encountered will remain open but will not be updated until a `PANELS ON` command is issued.

The following pop-ups are not affected by the `PANELS OFF` command:

- The script `PAUSE` popup
- Error pop-ups

These pop-ups will always appear if necessary.

The script command `PANELS ON` will allow **FieldView** to display/update panels again.

`PAUSE`

This command displays a small `PAUSE` dialog box in the upper right corner of the screen. **FieldView** will wait for you to click the `OK` button before continuing.

`PERSPECTIVE ON NN/OFF`

When encountered in a script, the state of `PERSPECTIVE` can be toggled either `ON` or `OFF`. If `PERSPECTIVE` is already `ON` and the script command is issued redundantly, it will stay `ON`. When turning `PERSPECTIVE ON`, the optional value of `NN` is applied. The number `NN` will be silently clamped to the allowable range of values 1–179, as it is when set

in the **FieldView** GUI, where lower values approach no PERSPECTIVE and higher values approach a fish-eye lens effect.

PLOT

The `PLOT` command will bring up a line plot of the current computational surface. This command is similar in action to `Select`, but is used for Surface Plots.

`PLOT_SIZE width height`

This command resizes the 2D Plot Window to the specified `width` and `height`, which are required arguments with values in pixel dimensions. When used with the `PRINT PLOT` command, the `PLOT_SIZE` command generates an output file based on the specified dimensions. Note that the plot window cannot be set below 500x400 pixels.

`PRESENTATION ON/OFF`

Turns Presentation Rendering `ON` or `OFF`.



Note: In general, the following **PRINT commands** are used with an output filename argument. If no filename is given, FieldView will provide one and place it in the directory specified by your `TMPDIR` environment variable. For example, a PNG format file created by user 'myname' on the 17th of December, at 1:27:09 PM will result in a file *myname.17Dec19.13.27.090000604801.png* (where the 10 extra randomized characters are provided to avoid file overwrite in the same second.)

```
PRINT <GRAPHICS|WINDOW> <BMP|JPEG|PNG|TIFF|EMF> [filename]
PRINT <GRAPHICS|WINDOW> <PS|EPS> [[BACK|NOBACK] [SEND|NOSEND]
    [GRAY|NOGRAY]] [filename]
PRINT PLOT <BMP|PNG|EMF> [filename]
```

Note that `<>` arguments to the command are required, and `[]` are optional arguments, and that the EMF output option applies only to Windows.

Example usage:

```
PRINT GRAPHICS PNG myimage
    ..produces an output image of the FieldView Graphics window. This may consist of
    multiple windows.
PRINT WINDOW PNG myimage
    ..produces an output image of only a single window.
```

The `PLOT` argument requires that either the Surface Plot or 2DPlot Window is open. If neither panel is open, an error message will be issued.

Note about additional Postscript `<PS|EPS>` options:

For Postscript `<PS|EPS>` output, there are four optional settings to determine if the background color should be preserved or not, whether the file should be sent to the printer, whether it should be printed in gray scale, and the name of the file. The default settings are: `BACK` (use white background), `NOSEND` (don't send to the printer), `NOGRAY` (don't use gray scale), and use the default filename. Note: The options `BACK/NOBACK` and `GRAY/NOGRAY` only apply to `GRAPHICS` window printed files. When the `SEND` option is used, FieldView creates a temporary PostScript file and sends it to the printer. The file is removed unless an output file is also specified. See [Chapter 7](#), for reference to the script `fv_to_printer.sh.sample`, required to enable the `SEND` argument.

```
PROBE dataset-number x y z
PROBE [MAT-File | CSV] dataset-number "file1" "file2"
```

This feature provides the functionality of Point Probing, combined with the scripting language. The functions which will be returned and written to the output file, are those which currently reside in the **Function Registers**, as set by the last visited or created visualization object. There are two forms of this command. The first will make the Point Probe panel the current visualization panel and behave as if the user had probed at the given location. It does not produce an output file.

The second form, without the MAT-File or CSV argument, will read a file in a format identical to Point Probe Input (a simple column of three coordinate values) and write a new file ("`file2`") in a format as described in [Appendix G](#) 2D Plot Format. If one of the optional format argument `[MAT-File | CSV]` is specified, a MAT-File or CSV file will be read and written. For information on MAT-File and CSV, see [Point Query...](#) in [Chapter 14](#) of **Working with FieldView**. The MAT-File format is the most computationally efficient format and is detailed in [Appendix J](#); CSV is detailed in [Appendix K](#)



Note: Putting the filenames that appear in the `PROBE` command in double quotes is optional.

```
RAISE OFF
RAISE ON
```

The **RAISE** commands are used to change the window popping behavior of **FieldView**. **RAISE ON** raises the graphics window to the foreground for every graphics window update. The 2D Plot Display panel (if present) will be raised over the graphics window. The purpose of this command is to allow recording of full-screen graphics without interference from other panels. The momentary panel popping will not appear on videotape during recording with the **STEP** or **SYSTEM** commands. **RAISE OFF** is the default.

Windows:

```
RECORD ON [GRAPHICS|WINDOW] [MP4[frame rate]|AVI|PNG|JPEG|BMP|TIFF] filename
```

Linux/Mac:

```
RECORD ON [GRAPHICS|WINDOW] [MP4[frame rate]|PNG|JPEG|BMP|TIFF] filename
```

The **RECORD** command is used to create flipbooks from within a script. When **RECORD** is turned **ON** and a **filename** is given, changes to the graphics window will be recorded in a flipbook until the **OFF** command is given. This can allow you to include sweeping of surfaces, changing of views and sweeping through time, all in one flipbook.

An optional argument allows you to create a flipbook animation from the entire contents of the graphics window including multi-window layouts (**GRAPHICS**) or the current window only (**WINDOW**). If no argument is specified, the default is **WINDOW** and a flipbook will be saved of the current window only, providing backward compatibility for single window scripts.

To save a flipbook of the current window, use these **SCRIPT** commands:

```
SELECT WINDOW window-number
RECORD ON WINDOW filename
(or RECORD ON filename)
    SWEEP ...
RECORD OFF
```

Optional arguments allow you to specify output format type and frame rate (fps). The valid output format types are platform specific. Frame rate is only valid for MP4 output. A script error will be issued if frame rate is specified with a non-MP4 output type. Default values are consistent with the previous version of **FieldView**.

For PNG, JPEG, BMP, and TIFF formats, each frame of the flipbook will be saved as an individual file. A base file name is specified. A one-based frame number will be appended to the base name for the individual frame file names. These numbers are displayed as 4 places with leading zero padding.

```
RESET
```

When this command is called in a script it will return the view to what **FieldView** has calculated to be the starting view after all dataset(s) have been read in, and *before* any transforms have been performed. Intent of this control is to match what will happen if the user presses the RESET button on the GUI when the Object Transform is set to WORLD. This setting will be particularly useful after a change to perspective has been initiated.

```
RESTART ALL complete-restart-name
RESTART ALL_CURRENT_WINDOW current-window-restart-name
RESTART ALL_NO_DATA_READ complete-restart-name
RESTART CURRENT_DATASET current-restart-name
RESTART MULTI_WINDOW_LAYOUT layout-restart-file
RESTART COLOR colormap-restart-file
RESTART COMP computational-surface-restart-file
RESTART DATA data-input-restart-file
RESTART FORMULA formula-restart-file
RESTART ISO iso-surface-restart-file
RESTART PREF preference-restart-name
RESTART STREAM streamlines-restart-file
RESTART PATHS particle-path-restart-file
RESTART TITLES titles-restart-file
RESTART VIEW viewing-controls-restart-file
RESTART PLOT surface-plot-restart-file
RESTART BOUNDARY boundary-restart-file
RESTART LINE 2D-plot-restart-file
RESTART COORD coordinate-surface-restart-file
RESTART PRESENTATION presentation-restart-file
RESTART VCORE vortex-core-restart-file
RESTART DYNAMIC_CLIP clip-groups-restart-file
```

The RESTART commands perform the specified Restart Files Read operation. The same RESTART operations are available as on the Restart Files menu, with the exception of Script Restart itself. The name of the restart file may include full pathnames, but wildcards are not expanded. File extensions may be omitted; they are automatically appended as detailed in [“File Naming Convention” on page 262](#).

When a Complete Restart is read interactively, the contents of all windows and the layout information is restored. To read a Complete Restart, including the multi-window layout component restart (if present), use the script command `RESTART ALL`.

To restore a Complete Restart to the Current Window ONLY in a multi-window session, use the script command `RESTART ALL_CURRENT_WINDOW`, noting that a multi-window layout restart will be ignored (if present).

To apply an existing Complete Restart to one or more datasets without reading the datasets originally specified within the restart, use the script command `RESTART ALL_NO_DATA_READ`. If multiple windows are present, this Complete, No Data Read restart will be applied to the dataset(s) in the current window only. The multi-window layout will not be affected.

To apply a previously saved restart from one dataset onto another dataset, use the script command `RESTART CURRENT_DATASET`. Note that a Current Dataset restart can be saved for a dataset in one window and applied to a dataset in a different window.

To restore a multi-window layout, use the script command `RESTART MULTI_WINDOW_LAYOUT`. Note that applying a multi-window layout will likely result in the redistribution of datasets, and that this action cannot be undone.

```
SAVE ALL complete-restart-name
SAVE ALL_CURRENT_WINDOW current-window-restart-name
SAVE CURRENT_DATASET current-restart-name
SAVE MULTI_WINDOW_LAYOUT layout-restart-file
SAVE COLOR colormap-restart-file
SAVE COMP computational-surface-restart-file
SAVE DATA data-input-restart-file
SAVE FORMULA formula-restart-file
SAVE ISO iso-surface-restart-file
SAVE PREF preference-restart-name
SAVE STREAM streamlines-restart-file
SAVE PATHS particle-path-restart-file
SAVE TITLES titles-restart-file
SAVE VIEW viewing-controls-restart-file
SAVE PLOT surface-plot-restart-file
SAVE BOUNDARY boundary-surface-restart-file
SAVE LINE 2D-plot-restart-file
SAVE COORD coordinate-surface-restart-file
SAVE PRESENTATION presentation-restart-file
SAVE VCORE vortex-core-restart-file
SAVE DYNAMIC_CLIP clip-groups-restart-file
```

The `SAVE` command allows the user to automatically save restarts.

When a Complete Restart is saved interactively, the contents of all the windows and the layout information is saved. To save a Complete Restart, including the multi-window layout component restart, use the script command `SAVE ALL`.

To save the contents of the Current Window ONLY in a multi-window session, use the script command `SAVE ALL_CURRENT_WINDOW`, noting that a multi-window layout restart will not be written.

To save the multi-window layout **ONLY** at any time during a session, use the script command `SAVE MULTI_WINDOW_LAYOUT`.

To save a restart that allows you to re-create the visualization objects on the current dataset to another dataset, use the script command `SAVE CURRENT_DATASET`.



Note: If you wish to turn off the display of all surfaces/objects of a particular type during a script, e.g. computational surfaces, merely create one computational surface, turn its visibility off and save it out as a Computational Surface Restart. In the script, restart this file to turn all your computational surfaces off. The types of surfaces/objects that this technique can be used for is: computational, coordinate and iso-surfaces, streamlines, annotation and 2-D plots.

```
SELECT COMP grid-number surface-number
SELECT ISO surface-number
SELECT STREAM rake-number
SELECT PATHS path-number
SELECT TITLES title-number
SELECT BOUNDARY boundary-number
SELECT LINE 2D-plot-number
SELECT PLOT plotnum [pathnum]
SELECT COORD surface-number
SELECT VCORE vortex-core-number
SELECT DATASET dataset-number
SELECT WINDOW window-number
```

The `SELECT` commands display the specified visualization panel (if not already displayed), and select the specified object as the *current* one on that panel. All actions in **FieldView** occur on the current object. If the specified object does not exist, the specified panel will still be displayed, but the current selection will not change, nor will an error be generated. All numbers specified should be positive integers; a value of 0 will display the panel without changing the current selection.

The `SELECT PLOT` command permits selection of a particular 2D Plot from one of many 2D Plots based on the argument supplied for `plotnum`. The optional argument `pathnum` permits selection of a specific plot path within a specific 2D Plot. If a non-existent value is specified for either `plotnum` or `pathnum`, the script will silently pass over the error. The script command `SELECT LINE [pathnum]` has been deprecated in favor of the `SELECT PLOT` command. However, it will still work and will now be interpreted to select `pathnum` on the current 2D plot for the current dataset.

The `SELECT DATASET` command allows you to switch between datasets in the script. This is required to export an object that does not belong to the current dataset. This is also necessary for sweeping through datasets (to pick a different dataset starting point). This command will also force the visibility of the dataset *on*.



Note: The `SELECT PLOT` command from a script may not work correctly if there are multiple datasets unless the `SELECT DATASET` command is present. A warning message will be printed to the xterm window in this case.

Note: `SELECT COMP grid-number surface-number` may give unexpected results when multiple datasets are present. This is due to internal Computational Surface numbering in **FieldView**. This command should be preceded by a `SELECT DATASET` command. For example, if you have two 2-grid datasets in memory, even if the 2nd dataset is current, the command `SELECT COMP 1 1` will not select the first grid of the second dataset, but the first grid of the first dataset. To correctly select the first grid of the 2nd dataset, precede this with a `SELECT DATASET 2` command.

Note: Selecting a computational surface requires specifying two numbers: a Grid number and a Surface number on that grid.

Note: Some Streamline attributes apply for all rakes. Those that do are `DISPLAY TYPE` (Complete, Filament, etc.) and whether the streamline is animated or not. Therefore, the `SELECT STREAM rake-number` will select a specific rake, but changes in these attributes will affect all rakes in the visualization. It is not possible to animate only certain rakes, for instance.

For restarts of the type Complete, Current Window..., Complete, No Data Read... or Current Dataset..., it may be necessary to specify the current window to ensure that visualization objects are displayed in the correct window. To make a window in a multi-window layout current, use the script command `SELECT WINDOW`. For the Current Dataset... restart type, it may be further necessary to specify the dataset. Dataset numbering is relative within each window. To select a specific dataset within a window, use the script command `SELECT DATASET`.

```
SHINE ON/OFF surface_type
```

This controls whether the Presentation Rendering property `SHINE` is added as an attribute to a given surface. Through this command, the user may allow `SHINE` to exist or not on several surface types independently. The possible values of `surface_type` are: `boundary`, `comp`, `coord`, `iso`, `stream`, `paths` and `vcore`.

Note: This command will not take effect until a refresh is performed. Therefore, if you wish this command to take effect *before* the script is exited, you must use the `SELECT` command on the appropriate surface/rake.

```
SHINEVALUES shine_intensity shine_highlight_size
```

This script commands allows the user to control the parameters of the `SHINE` command for surfaces. These settings hold *per surface type*. That is, all Computational surfaces have to have the same `SHINE` settings.

Note: This command will not take effect until a refresh is performed. Therefore, if you wish this command to take effect *before* the script is exited, you must use the `SELECT` command on the appropriate surface/rake.

`SIZE width height`

The `SIZE` command resizes the graphics window to the specified `width` and `height`, which are required arguments with values in pixel dimensions. This is similar to the `-size=WIDTHxHEIGHT` command-line argument (See [Chapter 1](#) of the **User's Guide**) which can be used when starting **FieldView**. This command will allow you to explicitly control the window size. This script command is meant to compliment the Tools pull-down menu giving you control over the graphics window size (see [Chapter 14 Working with FieldView](#)).

Warning: Use of the `SIZE` command during the recording of an animation (i.e., resizing the graphics window between the `RECORD ON` and `RECORD OFF` commands) will *not* generate an error, but will produce animation files that will cause problems for players and converters and is not recommended.

`SLEEP number-of-seconds`

The `SLEEP` command causes script execution to be suspended for the specified number of seconds, which must be a positive integer. Execution resumes automatically as soon as the specified time elapses.

`SPIN number-of-steps`

The `SPIN` command will spin the dataset the given number of steps. Each step increment is equal to 0.1 radians (approximately 5.73 degrees). A nearly complete revolution is accomplished by the command `SPIN 63`.

`STEP operating-system-command`
`UNSTEP`

The `STEP` command is identical to the `SYSTEM` command below, except that the operating system command is executed after every graphics window update. A second `STEP` will override the previous `STEP` command. The `UNSTEP` command cancels the previous `STEP` command (if any).

```
SWEEP number-of-cycles  
SWEEP BOUNCE number-of-cycles  
SWEEP DOWN number-of-cycles  
SWEEP UP number-of-cycles
```

The `SWEEP` commands perform a sweep operation if the current panel is Computational Surface, Iso-Surface, Coordinate Surface or the current Boundary Surface is sweepable (included in an XDB that has been generated in Build mode), or an “animate” operation to be performed if the current panel is Streamlines or Particle Paths. A `SWEEP` operation continues until the original surface value has been reached the specified number of times. For example, if you are performing a `SWEEP` on a computational surface defined as $J = 5$, and the number of cycles is given as 1, then the sweep operation will stop as soon as J equals 5 again (or the next sweep step would cause $J = 5$ to be passed over).

For Streamline restarts, an Animate operation continues until the original particle positions have been reached the specified number of times. The number of cycles specified for a sweep or animate must be a positive integer.

If the `SWEEP` direction is specified with `BOUNCE` or `DOWN` or `UP`, the current sweeping or animation direction is changed as requested prior to starting the sweep. If the `SWEEP` direction is not specified, the current direction is used. A `SWEEP` direction of `BOUNCE` is ignored if the current panel is Streamlines or Particle Paths. The entire `SWEEP` command is ignored if the current panel is Point Probe, Titles, 2-D Plots, Surface Plots, Vortex Cores / Surface Flows or the current Boundary Surface is non-sweepable (not included in an XDB that has been generated in Build mode).

```
SWEEP DATASET number-of-cycles
```

This script command allows you to sweep through all of the datasets that are currently loaded into **FieldView** (as when the Sweep button is pressed interactively).

```
SWEEP TIME number-of-cycles [skip] [ streakline_export_filename ]
```

The `SWEEP TIME` command is used to animate the results through time, as if the Sweep button were pressed on the Transient Data Controls. The `SWEEP` will occur from the current position, and will be based on the current beginning and end points on the Transient panel. Also, the current setting of Loop is used (see additional script command containing the `LOOP` format below). If you wish to change these values, you may use the Time Step or Time Solution commands. The Skip argument will cause that many time steps to be skipped before the next one is displayed. The `streakline_export_filename` argument will create a Particle Path file containing the streakline data. If this argument is

not present but streaklines are exported, **FieldView** will create a Particle Path file with a default filename. The file name will have the following naming convention:

```
streak<loginname>_12Jun99_12_14_29.fvp
```

where the numbers after the date are Hours_Minutes_Seconds

```
SWEEP TIME LOOP number-of-cycles number-of-loops [skip] [ streak-
line_export_filename ]
```

Either optional argument can be used, but if both are specified, then `[skip]` must be first. The relationship between cycles and loops is: SWEEP number-of-cycles cycles of number-of-loops loops.

Example:

Suppose you have 4 time steps, and the current setting of `LOOP` on the panel is 2.

`SWEEP TIME 2` would produce the following time steps:

```
1  2  3  4  1' 2' 3' 4' 1  2  3  4  1' 2' 3' 4'
<----- (1st cycle) -----> <----- (2nd cycle) ----->
```

The same result would be obtained by the command: `SWEEP TIME LOOP 2 2`.

`SWEEP TIME LOOP 1 3` would produce the following time steps:

```
1  2  3  4  1' 2' 3' 4' 1" 2" 3" 4"
<----- (one cycle) ----->
```

where `N'` and `N"` indicate extended step/times.



Note: With either form of the script command, if the transient sweep is going to calculate streaklines, `number-of-cycles` is ignored. The value of `LOOP`, whether using the current setting or explicitly specified, takes precedence. The `SWEEP` will do only one cycle of loops to ensure the validity of the exported streakline data.

See also `SELECT DATASET` for controlling the dataset number that the `SWEEP` starts on.

`SYSTEM operating-system-command`

The `SYSTEM` command is used to submit commands to the operating system. Everything on the command line, starting with the first non-blank character after the word `SYSTEM`, is sent to the operating system as a command string. This command may be an

executable module or a shell script or any other legally executable operation. Because this command uses the Unix system call, the command will be executed by a Bourne shell (/bin/sh). Note: If an error is encountered with a `SYSTEM` command and **FieldView** is running in the background, the process may hang.

```
TIME STEP dataset-number current [beginning end]
TIME SOLUTION dataset-number current [beginning end]
TIME SET DELTATIME float-delta-time
TIME SET MERGEDTIMES ON/OFF
TIME SET SEQUENTIAL ON/OFF
```

The `TIME STEP` and `TIME SOLUTION` commands are used to specify a time value for the specified dataset, and to set a beginning and end time value for use with the `SWEEP TIME` command. Setting a `TIME STEP` or `TIME SOLUTION` is identical to changing the time value on the Transient Data Control panel and pressing the Apply button. The `TIME SET DELTATIME` command allows you to specify the delta time step. The delta time can be used to sync transient datasets which have no solution time, or to create streaklines on data which has no solution time. See [Use Delta Time in Working with FieldView](#) for more information about this feature. The `TIME SET MERGEDTIMES` command lets you create a merged time line for multiple transient datasets. When this mode is enabled, a `SWEEP TIME` command will sweep through all the transient datasets loaded in the current **FieldView** session. If `TIME SET SEQUENTIAL ON` is also set, data not corresponding to the current time step will be blanked (made invisible). See [Use Merged Times in Working with FieldView](#) for more information about this feature.

```
TIMING ON
TIMING OFF [string]
```

The `TIMING ON/OFF` commands can be used to return timings for a given set of sequential lines of script. For example, the amount of time used to execute the script lines which fall between the "TIMING ON" command and the "TIMING OFF Timing for example commands is" will be reported by the following block of text in the terminal running **FieldView**.

```
[Client]: Timing for example commands is: CPU time = 146.61 + 0.93
= 147.54 Elapsed time = 166.91
Page faults = 0 per elapsed sec = 0.00 Swapouts = 0
Context switches: voluntary = 2502 involuntary = 258
```

Note that the CPU time will be small in comparison to the Elapsed time if a **FieldView** server is performing more of the work than the **FieldView** Client.

```
WRITE [text]
```

The `WRITE` command allows the user to print blank lines or quoted strings of text to the `xterm` prompt with lower overhead than using the `SYSTEM` command. With no argument, the `WRITE` command produces a blank line. Note: The `WRITE` command may not be able to process `text` strings that contain a dash (-). Also, `WRITE` commands will *not* work if you have put the **FieldView** process in the background at any time during the current session.

```
XDB_WRITE xdb-filename [THRESHOLD/NOTHRESHOLD] [title-string  
[notes-filename]]
```

This command allows you to create a `.xdb` file for subsequent viewing within **FieldView** or **XDBview**. Here, `xdb-filename` refers to the output filename to be used. The title, as interactively entered in the Viewer XDB Notes, is specified as a string value. The accompanying notes section can be included by indicating an additional file, with appropriate comments.

The optional argument `[THRESHOLD/NOTHRESHOLD]` toggles "Maintain Thresholded Surfaces." The keyword `THRESHOLD` enables thresholded surface export and the keyword `NOTHRESHOLD` bypasses thresholded surface export. Omitting this argument is equivalent to `NOTHRESHOLD`.

```
XDB_ENABLE xdb-filename [THRESHOLD/NOTHRESHOLD] [title-string  
[notes-filename]]
```

This command allows you to create a `.xdb` file, based on a `SWEEP` of a surface or transient dataset, for subsequent viewing within **FieldView** or **XDBview**. A subsequent `SWEEP` command will write the `.xdb` file and close it when the `SWEEP` finishes.

The optional argument `[THRESHOLD/NOTHRESHOLD]` toggle has the same function as described for `XDB_WRITE`, above.

IMPORTANT: The **Working with FieldView** section [Function Selection for XDB Export](#) describes the location and use of files `xdb_vars` and `root.xfn` which are **required** to specify the functions to be stored in your XDB extract when using the above two FVX commands. (Note that boundary variables [FVBND] are unsupported for Structured data.) For additional information and examples, see [Script Support for Viewer XDB Files](#) in [Chapter 14](#) of **Working with FieldView**.

Sample Scripts

For all of the sample scripts shown below, the lines:

```
RECORD ON filename  
RECORD OFF
```

..can be inserted at the beginning and the end, respectively, to create an animation file. See additional options for output file type, etc. in the description of the `RECORD ON` command, above.

Changing the View in an Animation

To change from the current view to a new view, the `INTERPOLATE` script command is used. The number of frames created will be equal to the `number-of-steps` in the `INTERPOLATE` command. The following example is taken from the start of the `F18_script_f18_show.scr` found in the `demo` directory of the **FieldView** installation:

```
!..restart initial position/view
RESTART ALL start
SLEEP 5
RESTART TITLES t0
INTERPOLATE 8 view2
```

Holding View (pausing)

While the `SLEEP` command will momentarily halt script execution, it is only useful for live script viewing. If a hold is required for an animation/movie (file), **FieldView** can create a number of held frames, each showing the same image. Thus, when played, the movie will appear to halt. To do this, use the `INTERPOLATE` command, and interpolate to the same view as the current view (in the following example, this is defined by `begin_movie.vct`):

```
RESTART ALL begin_movie
INTERPOLATE 20 begin_movie
```

Animating Streamlines During View Interpolation

Using the `SWEEP` command is independent of the `INTERPOLATE` command; they execute sequentially during a script. In order to have streamlines *SWEEP during* a view interpolation, a Streamline Restart with the `Animate` button `ON` must be saved:

```
!..restart initial position/view
RESTART ALL begin_movie
!..restart animated streamlines, then interpolate view
RESTART STREAM stream_on
INTERPOLATE 20 view2
!..finished view change - turn off streamline manually
SLEEP 5
```



Note: Restarting streamlines will recalculate the streamlines, giving the starting position of the rake, which may be different (i.e. different filament positions) than the position before the `stream_off` restart is used. This is why the `SLEEP` command is used.

Animating Streaklines For Transient Data

Starting with a Complete Restart of the desired visualization with the streamline seeds in their desired locations, here is how a streakline animation with an image of the last time step can be created:

```
RESTART ALL begin-streaklines
SWEEP TIME LOOP 1 4 1 streaklines.fvp
!..save out file to be able to jump to desired time step
!..
RESTART ALL begin-streaklines
RECORD ON streakmovie.avi
SWEEP TIME LOOP 1 4 1
RECORD OFF
!..
TIME STEP 1 48
PRINT GRAPHICS BMP last_time_step.bmp
```

Integrating Multiple Surfaces

Because the `INTEGRATE` script command operates on the current surface, the desired surface must first be made current using the `SELECT` script command. Note also that only the scalar function associated with the surface (in the surface restart file) can be integrated. For example, to integrate `pressure` over the Boundary Surface `outlet`, a Boundary Surface restart must first be created that has `outlet` as one surface with the function `pressure` loaded in the Scalar Register. This sample script integrates the scalar register function over several different boundary surfaces:

```
!..Use restarts to set up functions on boundaries of interest
RESTART ALL integrated_boundaries
!..Integrate Inlets, Outlets
SELECT BOUNDARY 1
INTEGRATE current integral.out
!..change surface, integrate (result is appended to existing file)
SELECT BOUNDARY 2
INTEGRATE current integral.out
!..repeat for each boundary surface
SELECT BOUNDARY n
INTEGRATE current integral.out
```



Note: If `INTEGRATE` is attempted and the current surface has no Scalar Function defined for it, an error pop-up will be produced.

Integrating Multiple Functions

Only the scalar function associated with the surface in the surface restart file can be integrated. In order to integrate more than one function over the same Boundary Surface (outlet, for example), one Boundary Surface restart must be created that has the surface `outlet` with one function (`pressure`, for example) loaded in the Scalar Register, and a second restart with a different function (`density`, for example) loaded in the Scalar Register. This sample script integrates different functions over the same boundary surface:

```
!..Use restarts to set up functions on boundary of interest
RESTART ALL integrate_start
!..restart boundary restart with "pressure" loaded
RESTART BOUNDARY outlet_pressure
INTEGRATE current integral_outlet_pressure.output
!..restart boundary restart with "density" loaded
RESTART BOUNDARY outlet_density
INTEGRATE current integral_outlet_density.output
```



Note: If `INTEGRATE` is attempted and the current surface has no Scalar Function defined for it, an error pop-up will be produced.

Automating the creation of a Sampled Dataset

A simple script example to read a restart and create a sampled dataset is shown below:

```
! Read a tet mesh dataset followed by a hex mesh
RESTART ALL tet_then_hex

! Choose the second [hex] dataset as the Grid Target
SELECT DATASET 2

! Choose the [tet] dataset as the Results Target, and
! create a sampled dataset called tet_2_hex
DATASET_SAMPLING 1 tet_2_hex
```

Chapter 6

Animation

6

Introduction

A simple animations can be created in **FieldView** using the Flipbook Controls panel and manually sweeping a surface or rake. The **FieldView** script language (along with restarts) can be used to sweep transient data, surfaces, rakes or interpolate from one view to another. These animations have the advantage of being extremely simple to create but lack the advantage of flexibility.

Keyframe animation is a more advanced tool that is used to create animations in **FieldView**. These animations can be simple (the same as or similar to those created with scripts) or they can be very advanced, allowing for exploding views, multiple surface and dataset sweeping, surface fade-in's and fade-out's. Keyframe animations have the advantages of flexibility and extreme control over all aspects of the animation and the disadvantages of needing more up-front planning.



Important Note: The abilities that keyframe animation adds to the animation capability of **FieldView** are such that animations often require planning. The animation steps should be thought through. The steps within the creation of a keyframe animation will often need to be done in a certain order, or the result will not be what was expected. A keyframe restart should be saved often during animation creation so that you can recover the animation at various stages.

There are several features in **FieldView** that complement the creation of simple and keyframe animations. These include specific mouse controls that allow you to create needed spatial transforms, a tool that lets you control the image size and format of your animation, and features that allow you to detach surfaces and regions from the view hierarchy.

[Flipbook Animation](#) and [Keyframe Animation](#) are covered in detail in subsections of this chapter.

Animation examples using the FieldView script language are provided in the Sample Scripts section of **this section** including a special way to animate (interpolate) the view while streamlines are animating (see [“Animating Streamlines During View Interpolation” on page 305](#)).

Flipbook Animation

Select the Flipbook Build Mode option from the Tools pull-down menu (see [Figure 115](#)) to turn the mode on and create a simple flipbook animation from the entire contents of the graphics window including multi-window layouts (Graphics) or the current window only (Window).

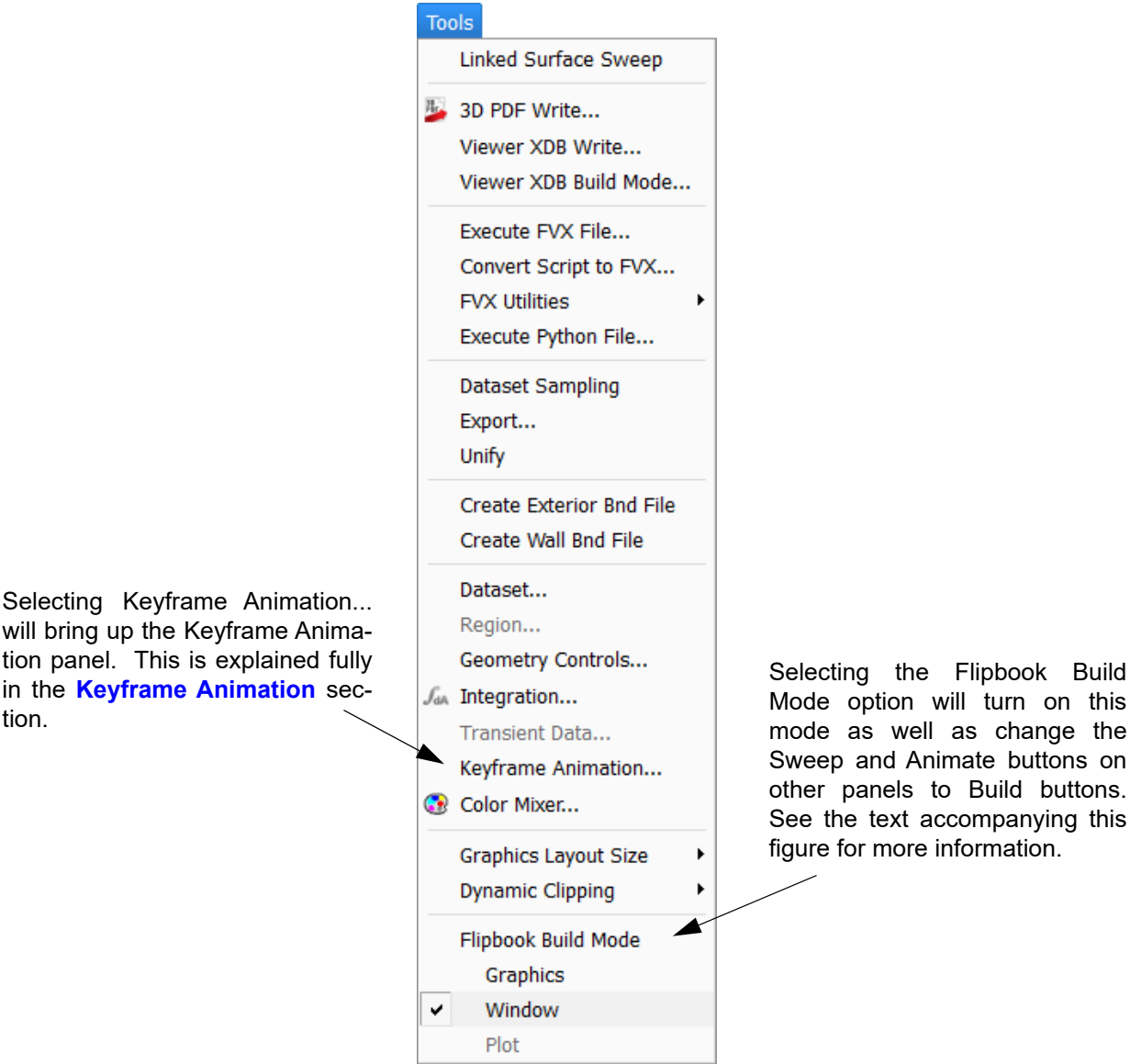
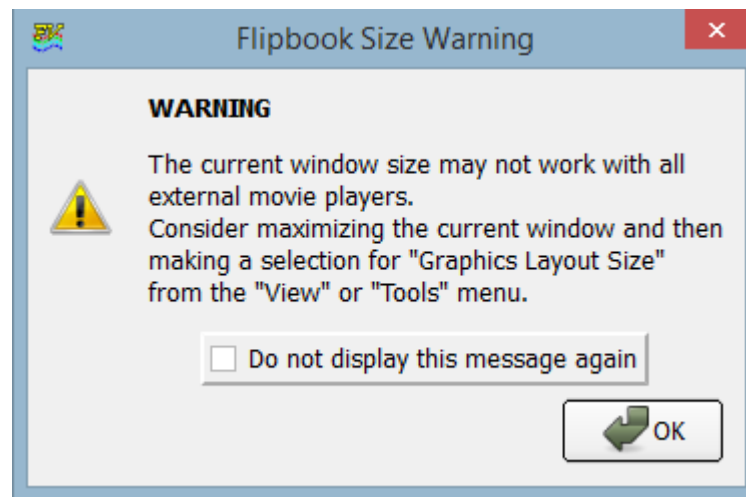


Figure 115 Tools Pull-Down Menu

The Flipbook Size Warning panel (see [Figure 116](#)) that comes up can be dismissed by clicking OK; refer to [Graphics Layout Size](#) in **Working with FieldView** for more information.

Refer to [Graphics Layout Size](#) in **Working with FieldView** for window size information.



Click OK to dismiss this panel.

Figure 116 Flipbook Size Warning Panel

In Flipbook Build Mode, the Sweep or Animate buttons on all **FieldView** surface and rake panels become Build buttons. Specifically, the Sweep button on the Computational, Coordinate and Iso-Surface panels becomes a Build button, allowing you to easily create an animated surface sweep. The Animate button on the Streamlines and Particle Paths panels becomes a Build button allowing you to create an animated rake. And, the Sweep button on the Transient Data Controls panel becomes a Build button allowing you to easily create animated transient data. If multiple datasets are loaded, the Sweep button on the Dataset Controls panel will change to Build. This function is used to alternately display the datasets (Visibility will be cycled for each dataset in memory).

Note that this form of animation only allows you to animate one object at a time.

Click the Build button on the desired panel.

Controlling the Sweep Extent and Step Control

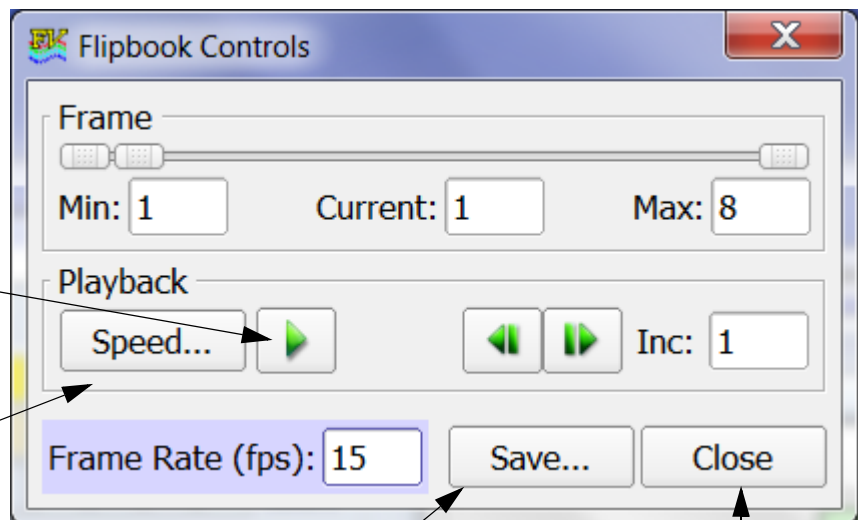
For surfaces and transient data, the number of frames that will be created for the animation will be equal to the value of Steps. For surfaces, this is the number of increments used to move the surface from the current Min value to the current Max value. Either or both of the Min and Max values can be adjusted by moving the sliders or by editing the Min and Max fields. This can be particularly useful when the region of interest in the dataset is of limited extent. Specific examples will be shown in the [Examples](#) section describing how to control the extent of the sweep process.

Building a Flipbook Animation

After the Build button is clicked (or, from the Keyframe Animation panel, the Build Flipbook button clicked and the Flipbook Size Warning panel (see [Figure 116](#)) dismissed), the Building Flipbook popup reports progress as the animation is created in memory. When done, the Flipbook Controls panel (see [Figure 117](#)) comes up and can be used to play the animation or save it to a file.

Once a flipbook has been created, turning on the Play / Pause button will continue to play the flipbook until the button is turned to Pause. Note: All other panels will be disabled while a flipbook is playing.

Once the flipbook is built, the playback speed may be too fast; to slow it down, click the Speed button and increase the Minimum Time Between Frames. Frame Rate (fps) applies only when saving a flipbook in MP4 format; the default, 15 (frames per second), may be adjusted to any integer from 1 to 60.



Pressing the Save button will bring up a browser giving you format options and allowing you to choose a directory in which to save the file. See [Figure 118](#) below. The frames from Min to Max will be saved.

Pressing the Close button will erase the current flipbook from memory.

Figure 117 Flipbook Controls Panel

Control and Playback of Animations

Once a flipbook (simple or keyframe animation) has been built, the **FieldView** graphics window becomes a 'playback' window. Until the flipbook has been saved or deleted, no interactive use of the graphics window is possible. Only after the Flipbook Controls panel is Closed will the graphics window return to its normal interactive mode.

While the graphics window acts as a playback window, you can control the range of frames that are played, the Playback Speed, and the Inc. Click the Speed button to bring up the Minimum Time Between Frames panel and increase the setting from 0 (zero) to slow down the flipbook playback in **FieldView**. Note that there is no dependency between this setting and the frame rate used for video export. An Inc value of 1 will cause all frames to play, an Inc of 2 causes every other frame to play, etc.

Output Formats

By default, a flipbook is saved as an MP4 file (see [Figure 118](#) below), which is the recommended video format. MP4 is a state of the art video format supporting up to Ultra High Definition resolution. MP4 videos will be encoded to H.264, complying with the video compatibility recommendations from many applications, including YouTube and Microsoft for PowerPoint 2013 and later (note that for older versions of Power Point, Microsoft recommends installing QuickTime). The H.264 codec has a quality

adjustment referred to as the Constant Rate Factor (CRF). This value may be specified with the environment variable

`FV_MP4_COMPRESSION_FACTOR`

where this is assigned an integer value from 0 to 51. A value of 0 will produce a lossless compression of the original image content at the expense of a large file size. A value of 51 will produce a highly compressed and probably low quality rendition of the original image content but with a drastically smaller file size. A default value of 23 has been chosen to be a good balance of fidelity vs. file size. A "sane" range of 18 for the highest quality and 28 for the smallest file size is recommended.

The "Frame Rate (fps)" entry on the Flipbook Controls panel (see [Figure 117](#)) lets you adjust the default MP4 frame rate of 15 frames per second to any integer between 1 and 60, inclusive. Note that this entry is not supported for other video formats. Selecting a non-MP4 file type will bring up a warning that Frame Rate will be ignored.

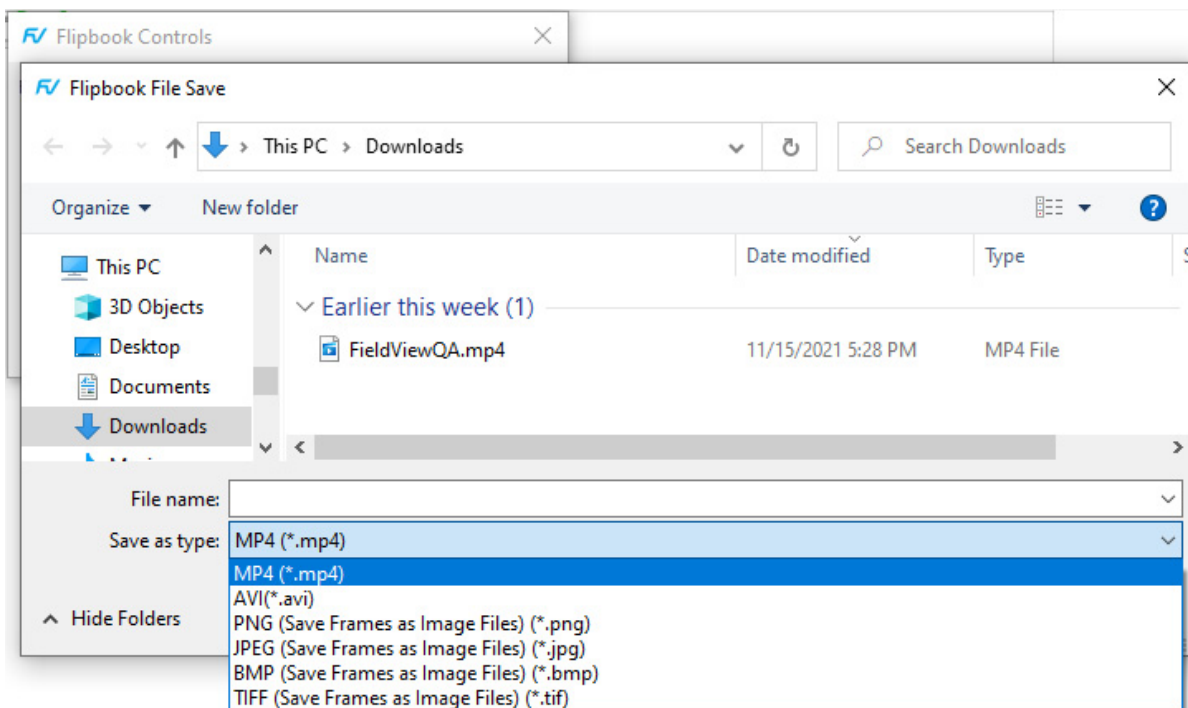


Figure 118 Flipbook File Save Panel

Note that Tools.. Graphics Layout Size (or interactively resizing your main FieldView window) can be used to select the image size prior to saving images or animations. See [Graphics Layout Size](#) in **Working with FieldView** for more information.

In Windows, animations can also be saved in AVI format. To use the high quality option with AVI files, set the following environment variable to any value:

FV_HQ_AV

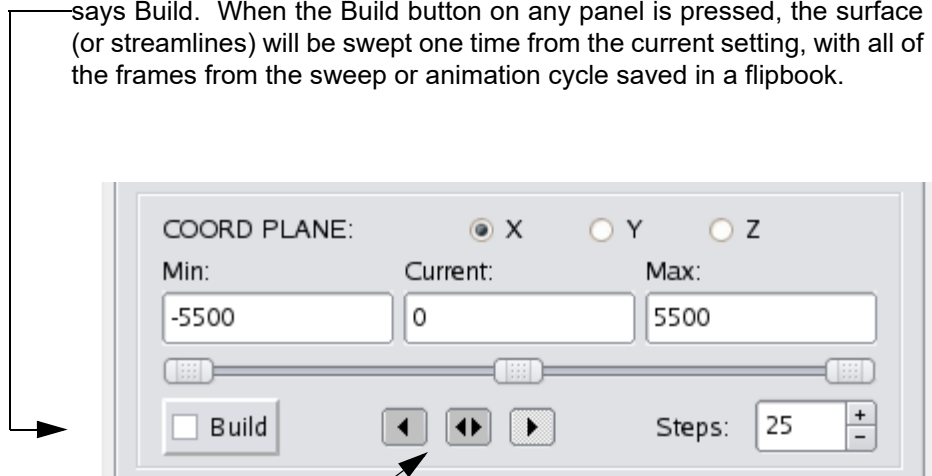
Alternatively, each frame of a flipbook may be saved as an individual file in PNG, JPEG, BMP or TIFF format. Specify a base file name in the Flipbook File Save panel. A one-based frame number will be appended to the base name for the individual frame file names. These numbers are displayed as 4 places with leading zero padding.

Examples

Example:

An \times Coordinate surface was created. Only the salient portion of the panel is shown in [Figure 119](#). The Min value is set to -5500 and the Max value is set to 5500 . The value of Steps is set to the default of 25, so the surface will sweep from -5500 to 5500 in 25 equal steps. Note also that the Sweep button has been replaced with a Build button. This is because Flipbook Build Mode has been turned on. To create this 25 frame animated surface sweep, merely press the Build button.

When Flipbook Build Mode is on, all Visualization Panels that contain a Sweep/Animate button will have that button replaced by a button that says Build. When the Build button on any panel is pressed, the surface (or streamlines) will be swept one time from the current setting, with all of the frames from the sweep or animation cycle saved in a flipbook.



The outer left and right arrows allow you to control the direction of the sweep. The right arrow is used to sweep forwards and the left arrow to sweep backwards.

Figure 119 Surface Sweep Extent and Step Control

Example:

A streamline rake was created. Only the salient portion of the Streamlines panel is shown in [Figure 120](#). The number of frames created depends on the streamline type. Some streamline types cannot be animated (Complete and Ribbons), some only create 10 frame animations because they can be looped (Filament, Spheres, Line of Spheres and Line of Dots) and others create the same number of

frames as the value of Div, which is 25 in this example (Growing, Spheres & Lines). Note also that the Animate button has been replaced with a Build button. This is because Flipbook Build Mode has been turned on. To create animated streamlines, merely click the Build button.



Figure 120 Streamline Build Control

Example:

A transient dataset has been read in and the Transient Data Controls panel is shown in [Figure 121](#). The Sweep button has been replaced with a Build button. This is because Flipbook Build Mode has been turned on. To create an animation of the current visualization for this transient dataset, merely click the Build button.

The Skip and Loop controls are covered in the [Chapter 6](#) section on [Streaklines](#) in **Working with FieldView**. Please refer to that chapter for details of these two controls. A Skip value of 1 means that every time step will be read-in, a value of 2 means only every other time step will be read-in, etc. The Loop value is the number of times the transient sequence (defined by the values of Begin and End) will be read. A Loop value of 1 means read from the Begin time step to the End time step once. A Loop value of 2 means read through the sequence twice, etc.

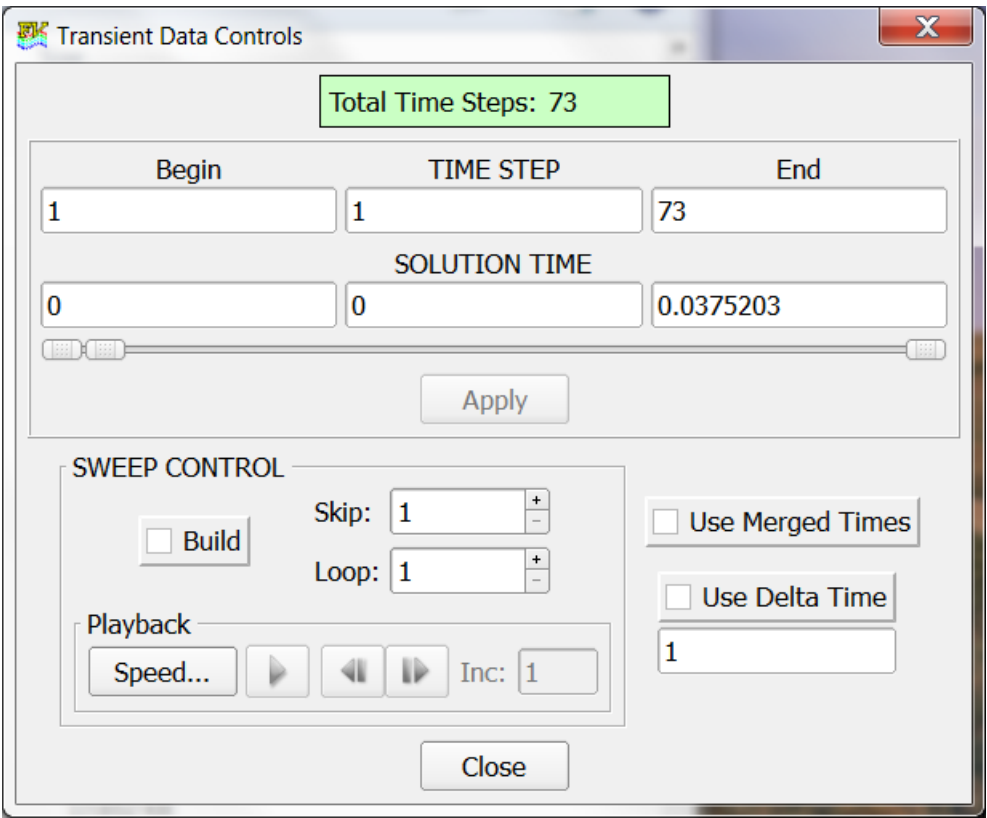


Figure 121 Transient Data Controls Panel in Flipbook Build Mode

Example:
Three datasets have been read into **FieldView**, the second and third with the Append data input button turned on. The data has been visualized and you wish to Sweep them, saving the results into an animation file. When Dataset Sweeping is performed, **FieldView** will toggle the Visibility of each of the datasets off in turn, so that only one dataset at a time is visible. The Sweep button can be found at the top of the Dataset Controls panel, only a portion of which is shown in [Figure 122](#) below. Note that the Sweep button has been replaced with a Build button. This is because Flipbook Build Mode has been turned on. To create an animation of the sweeping datasets, merely click the Build button.

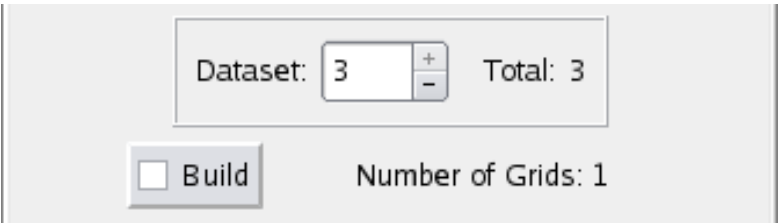


Figure 122 Dataset Control Sweep

Keyframe Animation

Keyframe Animation allows for more complete control of the animation process. With a flipbook animation, for example, you can easily sweep a surface or animate a rake, but nothing else. With a Keyframe animation, while you are sweeping the surface you can be clipping the sides, turning on the visibility of a parallel surface showing a different scalar (and then sweeping them simultaneously), then change them to geometric colored then fade them out while rotating the view. However, you can also create a simple sweep or spin Keyframe animation with just a few steps. Because of this potential to be complex, it is recommended that you plan out your keyframe animations, including what you want to accomplish, before you start creating them.

A keyframe animation consists of tracks of keyframes and actions. Tracks exist for each dataset, region, surface, etc. Keyframes are user-created and attached to a given object's track. They summarize what happens and when to datasets, regions, surfaces and rakes. Actions are used to specify the 'what' and the keyframe location on the track is used to specify the 'when'. Other things also have tracks associated with them. For instance, to rotate all of the datasets on the screen at the same time, a track can be created for the 'World' level within the **FieldView** hierarchy. Tracks can also be made for Streamline and Particle Path display and Lights.

Keyframe animation is accessed using the Keyframe Animation panel (see [Figure 123](#)) from the Tools pull-down on the Main Menu. This panel, shown and explained below, provides access to virtually all properties of **FieldView** that are normally available through the interface. Therefore, with this one panel you will be accessing all of the normal surface and rake properties (DISPLAY TYPE, SURFACE TYPE, Visibility, Thresholding, etc.) that you are familiar with.

There is Script and Restart support for launching and saving a keyframe animation. The script command, `KEYFRAME`, is described in [Chapter 5](#) of this **Reference Manual**. In addition, a Keyframe Restart file will save your keyframe animation for future use in a similar way that the other **FieldView** restarts do. If you Close the Keyframe Animation panel, you will be prompted to save a Keyframe Restart. The Keyframe Restart is much like a Complete Restart in that it requires **FieldView** to re-read the data as well. Therefore, if you wish to work on the animation at a later time (whether during the same **FieldView** session or not), you will need to Open the Keyframe Restart. This will load the animation and perform a Complete Restart, including reading in the data, even if the same data is currently in memory. If you do not wish to replace the data in memory, you can use the No Data Read option, which will restore all *but* the Dataset Restart. Important differences in operation compared to other **FieldView** restarts are described in the Keyframe Restarts section below.

Examples with step-by-step instructions showing you how to create keyframe animations for several different applications are intended to serve as a basis for creating keyframe animations of your own. Refer to the **User's Guide** for:

Basic Aerospace Tutorial [Step 10: Create a Keyframe Animation](#)
Feature Extraction Tutorial [Step 5: Create a Keyframe Animation](#)

Keyframe Actions

Keyframe actions fall into 3 classes. In this section we will examine each class in turn, list the actions that fall in each particular class and give simple examples describing actions and their effects on the animation.

Note that anything set by a keyframe will update the visualization *and* the appropriate panel. For example, suppose that a Complete Restart is read which has a Coordinate surface with its Visibility *on*, and you create a keyframe which turns *off* the Visibility. The Visibility of the Coordinate surface will be turned *off* in the graphics window. If that keyframe is then subsequently deleted, the Visibility of the Coordinate surface *will remain off*. It will *not* revert to its original state, but reflect its most recent setting.

Similarly, changing a setting on the panel instead of with the Keyframe Animation panel is permitted during keyframe specification mode (creating keyframes). However, please note that changes to surfaces using the panel may invalidate keyframe specifications. For example, if a keyframe is created which animates subsetting for J and K of an $I=5$ Computational surface, this keyframe would become invalid if the $I=5$ surface was changed to a J surface using the Computational Surface panel.

The available keyframe actions will be grouped according to keyframe action class. Keep in mind that not all keyframe actions are available for all track entities. For example, only Visibility and Transformation are available for Arrows (Titles), whereas surface entities allow most of the actions.

Simple Actions

The simplest action to set for a keyframe, this type allows you to create an animation with a single keyframe. These keyframe parameters affect the display on that frame number and after (sometimes for a specific duration), unless the duration is reached, or if the setting is changed by another keyframe later.

Affects Keyframe Location and After	
Animate	Spin
Fade In/Out	Current Value Sweep

Example: Spin the World

1. Select the World TRACK.
2. Set the Current Frame to 1.
3. Press the KEYFRAME Create button.
4. Change the Spin pull-down from Off to On.
5. Press the Play button.

This keyframe animation will cause the World to spin once around the default World axis (Y), taking the default number of Steps of 120 (this functions differently than the `SPIN` script command found in [Chapter 5](#) of this **Reference Manual**). Note: Turning On Spin will set a Transformation at the same

frame number. Turning Off Spin will do the same. This is done to correctly remember the initial orientation and to avoid confusing interactions between spin and transformations.

Example: Sweep a Surface

1. Select the desired surface TRACK.
2. Set the Current Frame to 1.
3. Press the KEYFRAME Create button.
4. Change the Current Value Sweep from Off to Up.

This last step will turn on the Current Value Sweep button. This keyframe animation will sweep the surface starting at Frame 1 and will continue until the Current Value Sweep is changed to Off by setting another keyframe.

Interpolated Actions

Interpolated actions use at least two keyframes to set a start and end condition for a track; more than two keyframes form an interpolation path. The smooth interpolation between the start and the end is done internally by **FieldView**. The number of steps of the interpolation will be determined by the keyframe locations.

Sets Position for Possible Later Interpolation	
Transformation	Base Transformation:
Legend Transformation	Scale X, Y or Z
I, J, or K Min/Max	Rotate Axis X/Y/Z
X, Y, or Z Min/Max	Rotate Axis X/Y/Z
Threshold Min/Max	Rotate Axis X/Y/Z
Scalar Min/Max	Translate X, Y, or Z

Example: Create a Moving View of the Model

1. Select the World TRACK.
2. Set the Current Frame to 1.
3. Press the KEYFRAME Create button.
4. Turn on the Transformation button (this step is required to set the initial view).
5. Change the Current Frame to 20.
6. Press the KEYFRAME Create button.
7. Using the mouse controls, move the model to the desired position.

This last step will turn the Transformation button on and define the final position. When the keyframe animation is played, **FieldView** will smoothly interpolate between the initial and the final views. It will *not* retrace the path you took to get to the final position. You can therefore use as many mouse actions to get just the right final view without worrying that **FieldView** is 'capturing' your every movement. If

you *want* to have your model trace a particular path from one view to another, you need to create multiple Transformation keyframes, which will act as ‘control points’ to guide the interpolation along.



Note: If only one Interpolated Action is set, then this will set the same state for the entire animation. Only by creating a second Interpolated Action of the same type do you provide sufficient information for interpolation.

Attribute Actions

Attribute Actions make a change to the attribute of a track. These keyframe parameters affect the display for the entire animation, unless the setting is changed by another keyframe later or earlier. No interpolation will occur.

Affects Entire Animation			
Surface Type	Vector Length Scale	Axis Markers	Presentation Rendering
Display Type	Vector Head Scaling	Outline	Transparency Value
Legend Type	Vector Head Scale	Perspective	Visibility
(Streamline) Coloring		(Streamline) Display Type	

Example: Change a Mesh Surface to a Constant Shaded Surface

1. Select a surface TRACK.
2. Set the Current Frame to 10.
3. Press the KEYFRAME Create button.
4. Turn on the Display Type button and set the value to Mesh (if not set already).
5. Change the Current Frame to 20.
6. Press the KEYFRAME Create button.
7. Change the Display Type value from Mesh to Constant Shading.

This last action will turn on the Display Type button. When this animation is played, the surface will be shown as a Mesh surface from frames 1 to 19 (there is no other Display Type set at frames earlier than 10, so they will display Mesh as well). At frame 20, the surface will change to Constant Shading and remain this Display Type unless changed by setting another Display Type keyframe for this track later.



Note: In this example, the Display Type Mesh keyframe could be created on any frame from 1 to 19 with exactly the same result, since this is an Attribute Action type.

Keyframe Animation Panel

This section is used to Create a new keyframe animation (which will delete any existing keyframe animation, if present), Open an existing animation (this will perform a Complete restart, or a Complete, No Data Read restart - see below), Clear the animation in memory, or Save the animation (and associated restart files).

Also shown is the length of the animation and access to the Time Line display panel. The Length is a type-in field that you can use to change the default length of 120.

This section shows the current Track. The Clear button will delete all keyframes on the current track. The Select... button allows you to change the current track through the Track Selection panel (see below). Selecting different types of entities may change the Object: setting on the Transform Controls panel. See below for more details.

This Create button allows you to add a keyframe to the animation. It is used to create keyframes once an animation has been created or opened.

The Delete button allows you to delete the current keyframe.

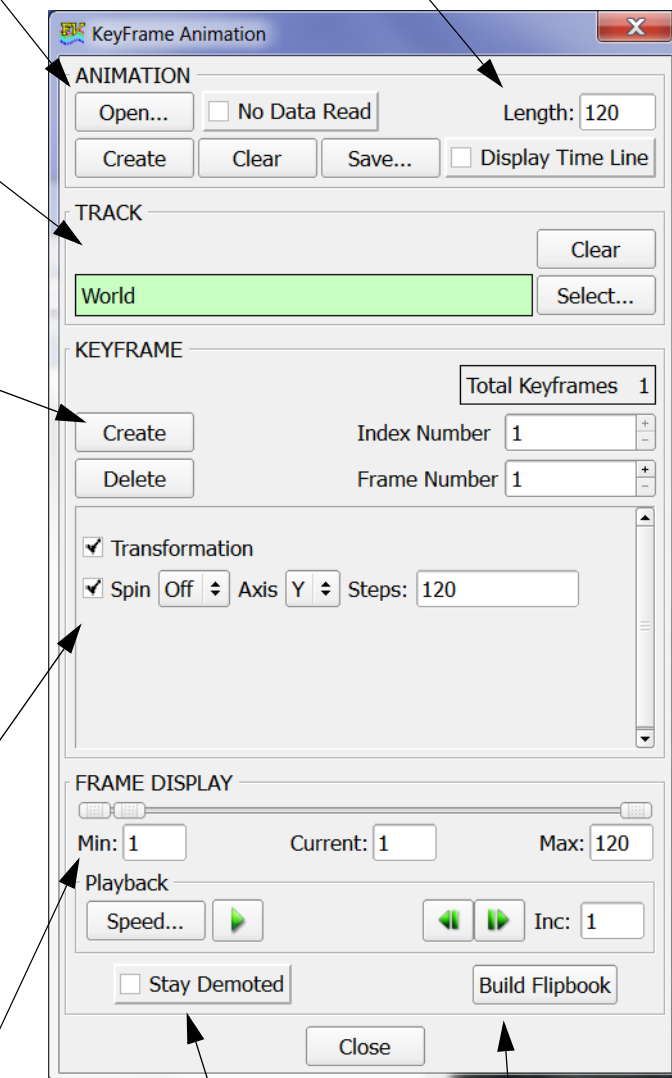
Shown on the right is the Total number of Keyframes for the current track, the Index Number and the Frame Number (location) of the keyframe in the animation.

See [Figure 124](#) for a complete description of this section.

This section provides access to virtually all panel settings. It shows all attributes that can be adjusted or changed for a given keyframe. If there is no keyframe for the current frame, then this section will be entirely grayed out. The contents of this section varies depending on the type of track currently loaded. The initial value of most fields will reflect values found on the panel in question.

The Set... button allows you to set the parameter value (which can also be done by editing the field to the left of the button), and see the parameters' current minimum and maximum (see below).

The sliders and type-in fields can be used to set the Current Frame and the Min: and Max: values (to shorten the range over which the animation is played). Speed allows you to increase the Minimum Time Between Frames for slower playback. Inc allows you to skip frames for faster playback. The Play button allows you to preview the animation before saving it.



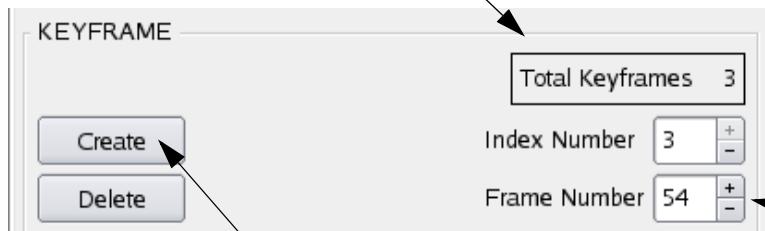
The Stay Demoted button will display the animation with the current demotion setting (for faster playback). See [Viewer Options > Viewer Demote: Pull-Down](#) for

Press the Build Flipbook button to save an animation file (see [Building a Flipbook Animation](#)).

Figure 123 Keyframe Animation Panel

Due to the importance of the KEYFRAME section of the Keyframe Animation panel, it has been singled out for explanation below. Here we emphasize the importance of differentiating *this* Create button with the Create button at the top of the Keyframe Animation panel. We also discuss the interaction of the Index Number and Frame Number plus (+) and minus (-) signs with the Current Frame field and parameter information displayed.

This display-only field shows the total number of keyframes that have been created for the current track.



This Create button is used to add a keyframe to the current track. The frame at which the keyframe will be added is initially the same as the Current Frame. If a keyframe already exists at the current frame for the current track, a Warning pop-up will be issued informing you of this.

The Delete button will delete the current keyframe shown by the Index Number.

Index Number + and - allow you to change between keyframes for the current track. You can also highlight and type-in the desired index. The Frame Number and panel will update showing the frame location of the index number.

Frame Number + and - allow you to change the location of the keyframes for the current track. In this example, clicking + (the plus sign) will cause Keyframe 3 to move from Frame 54 to Frame 55. If this is not desired, merely click - (the minus sign) to move it back to its original position.

Figure 124 Keyframe Animation Panel Create Action

The Create button will create a new keyframe for the current track at the frame number designated by the Current Frame field. This will cause the Total Keyframe count to increase by one. If a keyframe is created *before* any existing keyframes, then the successive keyframes will be renumbered. For example, if Keyframe 1 exists at Frame 1 and Keyframe 2 exists at Frame 40, then creating a new keyframe at Frame 20 will cause Keyframe 2 to be renumbered to 3. Whenever keyframes are deleted, moved *beyond* existing keyframes, or added, the keyframes will be sorted and renumbered in order of ascending frame number (time). Creating a keyframe at a frame where a keyframe already exists for the track is permitted. If such keyframes make conflicting specifications for one or more parameters, a warning will be issued and the keyframe with the higher number (which will be the *newest* keyframe), will prevail.



Note: Surfaces, Rakes, Annotation, etc. can be created at any time with the normal **Field-View** interface during keyframing. You need not rely on every desired element being present before starting the keyframe animation process. Anything new added will be added to the list of tracks available for selection through the Keyframe Animation panel.

The Frame Number shows the frame location of the keyframe indicated by the Index Number. If you decide that an action is occurring too early or too late, you can move the keyframe location simply by

clicking the plus or minus sign (+ or -) to the right of the Frame Number setting. You do not have to delete the keyframe and recreate it at the new position.



Important Note: If the Frame Number does not match the Current Frame, then the parameter fields, buttons and pull-downs will be grayed out, even if there is a keyframe for the current track at the Current Frame value. The fields on the Keyframe Animation panel will, however, show proper parameter values. In this specific case, the values may be the result of interpolation between two set keyframes. In the above example, if Index 2 is at Frame 20, moving the Current Frame slider to 20 will not display the keyframe information. However, clicking the Index Number minus sign (-) to move to Index 2 will show the proper, active parameters for Frame 20. Be careful not to use the Frame Number + or - or you will move the keyframe location. If this occurs, merely use the opposite sign to move it back to its original location.

Note: Surface, Streamline and Particle Path tracks can be accessed by “quick-picking” (double-clicking) the surface, rake, etc. in the graphics window. The appropriate track will then be shown as if it were selected using the Keyframe Track Selection panel.

Keyframe Track Selection Panel

The Keyframe Track Selection panel ([Figure 125](#)) shows the current objects available on which keyframes may be specified. All datasets, surfaces and rakes are numbered showing their dataset value and the surface/rake number. The surface/rake numbers reflect those that appear at the top of each of the surface and rake panels (Coordinate, Streamlines, etc.). Notation for Computational surfaces will also include their grid number, as well as the dataset number.

If a track contains keyframes then the “>” symbol indicates that a track has one or more keyframes associated with it, and the number of keyframes is indicated to the right of the track name.

By highlighting one of the tracks and pressing OK (or double-clicking), the track is loaded into the Keyframe Animation panel. From there, a keyframe can be added, edited or deleted for the selected track.

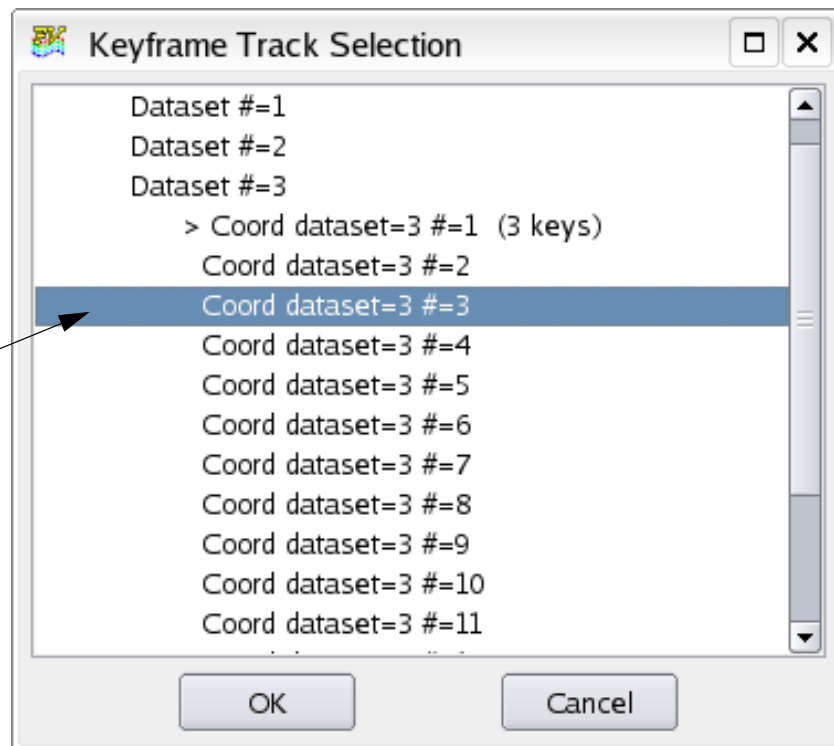


Figure 125 Keyframe Track Selection Panel



Note: Selecting a track may cause the current Object: setting on the Viewer Toolbar to change. This is done to make it easier to perform transformation actions on the track entity. If the track consists of a surface (Computational, Coordinate, Boundary or Iso-Surface), then the Object: setting will change to Surface. If a track is Light, then Object: will change to Light. If the track is a Title, then Object: will change to Title. If the track entity is a rake (Streamline or Particle Path), since they are not themselves transformable, the Object: setting will change to Legend, since rakes can have legends. If the track entity is a Legend, the Object: setting will not change because Legends can be connected to surfaces. You will have to manually change it to Legend to perform a legend transformation.

The Keyframe Value Specification panel ([Figure 126](#)) is brought up when the Set... buttons on the Keyframe Animation panel are pressed. It allows you to view and change the current value of a particular parameter. [Figure 126](#) shows that the X Min value of the current keyframe is 3. Note that the slider is not at its left-most position. There are three reasons why this may be: i) the X Min value *on the panel* was changed before this keyframe was created, and this panel is merely reflecting this change, ii) the Set... button was pressed *after* the value was set by editing the field of this parameter on the Keyframe Animation panel, or iii) this keyframe is showing an intermediate step between two other keyframes, caused by interpolation (one before the current keyframe, or one later) where the X Min

value changes. In the example shown in [Figure 126](#) below, the value of X Min is 3 because the value of X Min had been adjusted on the surface panel before a keyframe was created.

Streamlines

There are two different streamline tracks. Those settings (Visibility, Coloring, Scalar Min/Max and Legend information) are available *per rake* and different settings can be used for each rake. However, some Streamline parameters are *global* in **FieldView**. These global parameters are set by using the Streamline_Display track. These settings include whether the rakes are animated and the animation direction, the Divs value, Cycle Length and Display Type.

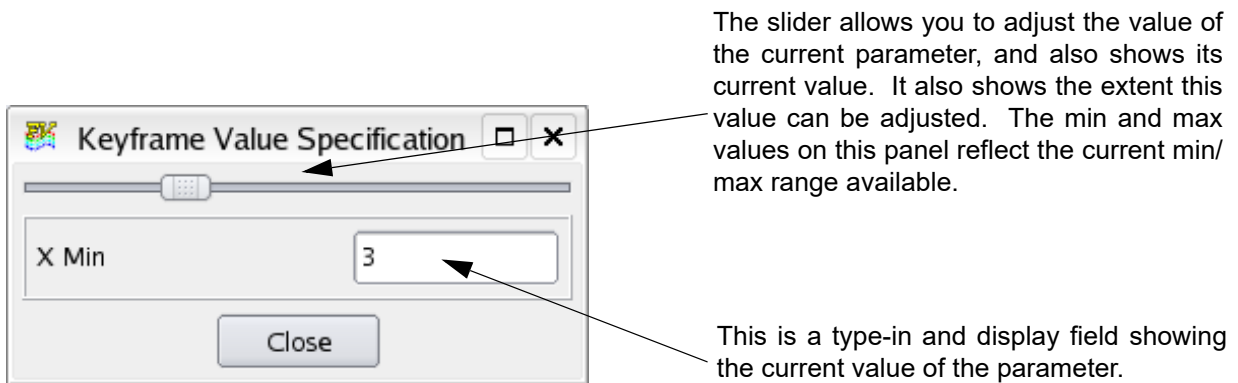


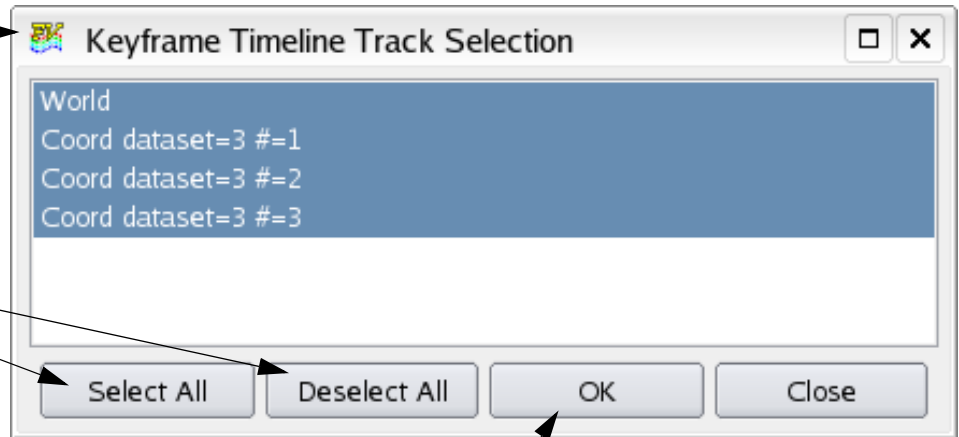
Figure 126 Keyframe Value Specification Panel

Keyframe Time Line

This is a 1-D graphical representation of the keyframe animation. A single horizontal line is shown for each track with symbols showing the location of the related keyframes. The length of the time line will be equal to the length of the highest numbered keyframe *that has been selected* using the Keyframe Time Line Track Selection panel ([Figure 127](#)). The symbol locations will be dictated by the frame number of each keyframe. That is, all keyframes at Frame 30 will line up vertically. An example of a time line as seen in the Time Line panel is shown in [Figure 128](#) below.

Individual tracks can be deselected or selected by clicking on them with the left mouse button. Press OK when finished.

Pressing Select All will highlight all of the tracks for plotting. This is the default when the Display Time Line button is pressed. Pressing Deselect All will turn off all tracks.



Pressing OK will bring up the Time Line panel showing all of the selected tracks. Cancel will close the panel.

Figure 127 Keyframe Time Line Track Selection Panel

The Keyframe Time Line Display ([Figure 128](#)) will graphically show the selected tracks that have keyframes associated with them as well as the location of the keyframes plotted as horizontal axes. Each track that has a keyframe will produce a time line in the window. An “x” symbol shows the location of a keyframe with the keyframe position noted numerically below it. An “⊗” symbol indicates two or more keyframes at the same location. The lack of an axis for a given track indicates a single keyframe. Two or more keyframes will be joined by the horizontal axis.

The spacing of the keyframe annotation is determined by the current selection set. The panel can be manually resized, but is not refreshed when resized.

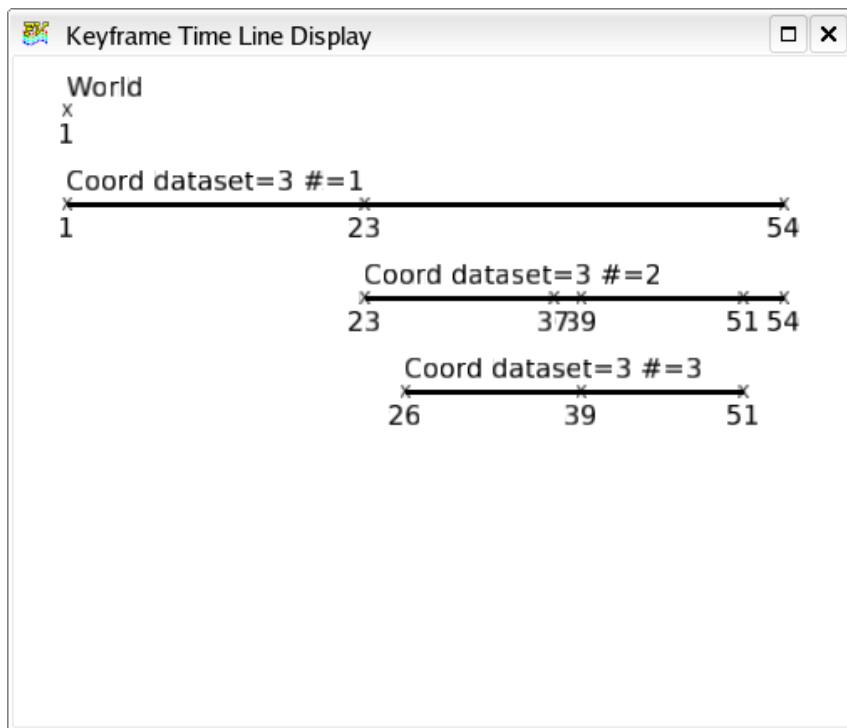


Figure 128 Keyframe Time Line Display

Keyframe Restarts

The Open and Save buttons on the Keyframe Animation panel allow you to load and store keyframe animations. However, the Keyframe Restart operates differently than other **FieldView** restarts. These differences are explained in this section.

A Keyframe Restart functions more like a Complete Restart than an individual surface/rake/view/etc. restart. When a Keyframe Restart is saved, **FieldView** will store the information needed to recreate the keyframe animation in a *.key file. However, it will *also* store the information needed to recreate the visualization in a Complete Restart, meaning that any necessary files (*.dat, *.vct, *.map, etc.) will be created as well. This set of files will follow a unique file naming convention. Suppose the Complete Restart used to create the visualization from which to work is called *start* (i.e., the restart files are *start.dat*, *start.vct*, etc.) and the keyframe restart is given the same filename prefix of *start*. Then the Keyframe Restart file will be called *start.key*, and the associated Complete Restart files will be given the prefix *start_key* by **FieldView**, and be named *start_key.dat*, *start_key.vct*, etc. This is to avoid the possibility of overwriting the initial Complete Restart used to start the process.

If you read the Complete Restart *start_key* with the Restart Files pull-down hypothetically created in the paragraph above, you will get the starting visualization but *without* the keyframe information. Instead, you should Open the Keyframe Restart, *start*, which will then read the Complete Restart *start_key* and load the keyframe information contained in the *.key file. That is, a Keyframe

Restart should only be read on the Keyframe Animation panel. This will replace the data, all visualization and the keyframe currently in memory. To restart *just* the keyframe animation, press the No Data Read button. This will act the same as the Complete, No Data Read restart in that the data files will not be re-read.

Keyframe Restarts can be saved at any time during the process of creating a keyframe animation. Choosing the same Keyframe Restart filename as already exists will overwrite the files, similar to other types of restarts. We recommend that you save your work often.

There is script support for a Keyframe Restart. This is described in the previous chapter. This will allow you to run a keyframe animation from a script. Batch mode also supports the keyframe restart, so a keyframe animation can be made in this fashion. The script command accepts optional “start frame”, “end frame” and “frame increment” arguments. Using multiple “keyframe” script commands in a single script will allow you to assemble separate animation scenes into a complex whole.

Like other **FieldView** restart files, the Keyframe Restart is a simple ASCII file. While this file *can* be edited, it is not meant to be edited. Like a script (see the previous chapter), comments *can* be added to a Keyframe Restart file. Any line beginning with a “!” character will be ignored. However, if the *.key file is overwritten, all comments will be lost.

Warning: If you have a keyframe animation created, but then read in a Complete Restart, a warning pop-up will be issued, as this will *delete* the current keyframe animation.



Note: If a Complete Restart is saved at any time during keyframing (as opposed to a Keyframe Restart), you will save the current visualization state of the graphics window, with the exception that any “fade” value (due to a Fade-In/Out setting) will be ignored, and all faded objects will be completely visible. To save a transparency value, it must be set via the Transparency field in the keyframer or on the surface panel.

Why do I get a perspective viewing warning every time I create a new keyframe animation?

The following warning pop-up is issued when a keyframe animation is created with Perspective off:

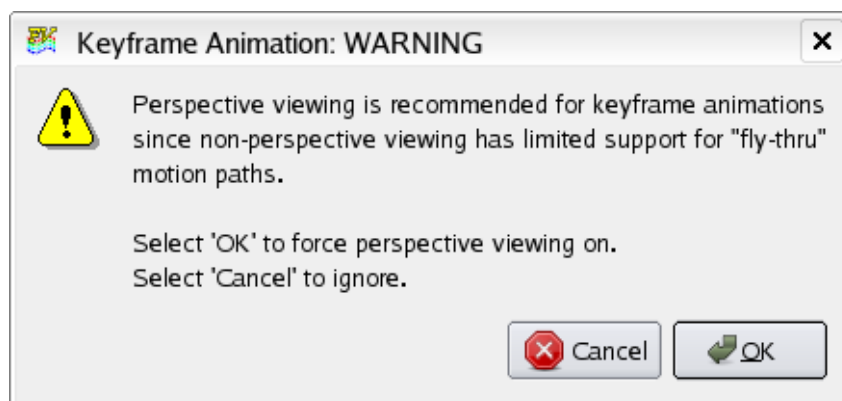


Figure 129 Keyframe Animation Perspective Warning

While creating an animation with perspective off is allowed, normally, perspective will need to be on to perform fly-by's, fly-through's, and other camera motions that can be performed with the keyframe animator.

What happens if I delete the surface, rake, etc. that has keyframes associated with it?

Deleting a surface (or other entity) for which you have specified keyframes will create 'orphaned' keyframes. In this case, the following warning pop-up will be issued.

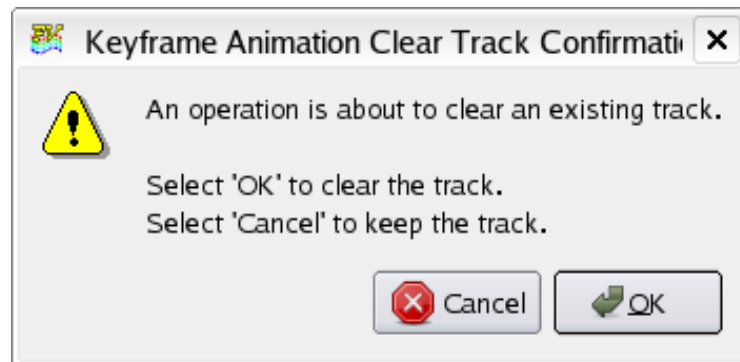


Figure 130 Keyframe Delete Track Confirmation

Pressing OK will delete the surface, rake, etc. and clear the track of any keyframes associated with it. Pressing Cancel will keep the track and keyframes intact and also cancel the surface deletion. Deleting a track may cause existing tracks to be renumbered, due to surface renumbering.

Why does my fade-in/out or transparency seem to use integer steps?

Fade-In/Out for surfaces, rakes, and titles (Helvetica only) uses the 9 step preset **FieldView** transparency settings [0 (in), 0.125, 0.250, 0.375, 0.500, 0.625, 0.750, 0.875, 1 (out)]. Since Fade-In/Out starts immediately, the default Count is 8. Values of Count used should be multiples of 8. If 16 is used, then each fade (transparency) value will be held for an additional frame, so that the fade will happen more slowly. However, no interpolation will be used on transparency values. Using values of Count that are not multiples of 8 will cause sampling, meaning that some of the frames will be held for an additional frame, some will not.

Transparency is a Simple action, meaning it sets the transparency for the entire animation unless it is set again, which will cause interpolation. The interpolation will actually consist of the preset transparency settings, with additional hold frames inserted if necessary.

Fade-In/Out and Transparency have no effect on mesh surfaces, contour lines, outlines, point or plot markers, streamlines, particle paths or line-drawn fonts (non-Helvetica).

Why does Set Center of Rotation set from the Defined Views panel have no effect on keyframe translations?

The Set Center of Rotation only has an effect on the center of rotation during interactive viewing operations done in the graphics window. It has no effect on keyframe interpolation of successive transfor-

mations. Consider using a Region file to re-locate the origin of the dataset. Subsequent spins will occur about this newly defined origin and coordinate system. See [Chapter 3](#) of this **Reference Manual** for more details on regions.

Why are streaklines not produced from streamline seeds when performing a transient sweep with the keyframe animator?

Streaklines force specific display settings for streamlines and disable any changes to the transient panel during the first sweep while a Particle Path file is being built (for use during subsequent sweeps). Allowing this would unnecessarily complicate the keyframe animation process. In order to display streaklines with the keyframer, you should pre-calculate them and read them in using the Particle Path Import feature. Then you can set keyframe parameters for them as particle paths. See [Chapter 6](#) and [Chapter 7](#) of **Working with FieldView** for more information about Streaklines and Particle Paths.

Why do my streamlines disappear when I perform a transient sweep with the keyframer?

If a transient sweep is performed with the keyframer then no streamlines will be shown for the current time step and all successive time steps. A streamline will only be made visible if the time step on which it was created is the current time step and that time step is not part of a transient sweep performed by the keyframer.

Why is there no Thresholding button available in the Keyframe Animation panel?

A Thresholding button would be redundant in this context. Thresholding can be accomplished by setting the Min: or Max: values to any value other than their defaults. If the Min: is, by default, 0, then setting it to 0.2 will threshold the surface by the function loaded into the Threshold register so that values between 0 and 0.2 will be excluded.

Error Conditions

Error - line N - char n: Track target not present in currently loaded dataset

This error message can be produced when reading a Keyframe Restart. It means that there are keyframe settings for some entity (surface, rake, etc.) which does not exist. This may occur if, for example, a Complete Restart is saved (after having changed the entities present) with the same filename as an already existing Keyframe Restart. For example, a Keyframe Restart might be called `start.key` with the accompanying files: `start_key.dat`, `start_key.vct`, etc. If you then later save a Complete Restart and call it `start_key`, you will overwrite the base Keyframe Restart files.

Perspective and Mouse Controls

The default start-up mode for **FieldView** is for Perspective to be on. Perspective *on* is required for certain types of 'fly-through' animations, where the (camera) view actually moves through the data space in a direction perpendicular to the screen. This type of movement is only possible if the view uses Perspective, thus allowing **FieldView** to draw elements "behind" you as well as in front of you.

In order to provide easier access to the Perspective-Z transform, use the spacebar to toggle between two different states of mouse interaction. This toggle will change the icons of the Transform Control panel.

If the current Object is World, Dataset, Region, Surface or Light (but *not* Title or Legend) *and* the current Action is Multi-Transform, then an additional set of transform controls can be toggled merely by pressing the Space Bar on your keyboard. This will change the icons and their behavior to those shown in [Figure 131](#) below (see also [Chapter 14](#) of **Working with FieldView**). Pressing the Space Bar again will return the icons to their previous state.



Note: The toggle mode only functions if the graphics window is the window which has focus. Depending on your computer settings, the graphics window may need to be clicked to gain focus. “Quick-picking” (double-clicking) on an object will normally cause the panel that comes up to have focus. You may need to click on the graphics window to gain focus there for this feature to work.

Depress [M1] for combined XY translation (X translation is horizontal movement, Y translation is vertical movement).



Depress [M3] for Magnification when Object is World, Scaling when Object is Dataset.

Depress [M2] for XYZ rotation.

Figure 131 World/Dataset/Region/Surface Multi-Transform

Click and drag to create Zoom Box. Action may be repeated for a total of 10 zooms. To Unzoom, you must change the Action to Zoom Box.



Depress [M3] for Z translation. This only works when Perspective is turned on.

Depress [M2] for Screen Z rotation.

Figure 132 Toggle Multi-Transform Icons



Note: The [M3] magnification transform shown in [Figure 131](#) has the effect of scaling the object. This makes the object look larger in the graphics window. Using the [M3] z-translation transform shown in [Figure 132](#) moved you closer to or further from the object, and only functions when Perspective is on. With the z-translation transform, you can actually pass

through an object (surface, etc.). This feature is necessary for fly-through animations. By default, Perspective is on when starting **FieldView**.



↑
Depress [M2] for the XYZ rotation.

Figure 133 Light Multi-Transform



↑
Depress [M2] for screen Z rotation.

Figure 134 Light Toggle Multi-Transform

Surface and Region Detach

The Detach button on the Viewer Toolbar (see [Figure 26](#)) can be used to isolate the current Region or Surface from the transform hierarchy. This means that once a region/surface is detached it is no longer affected by subsequent World, Region or Dataset transformations. Regions will not be affected by World, Dataset or Surface transforms and Surfaces will not be affected by World, Dataset or Region transforms.



Important Note: Detaching a Region or Surface will cause all interactive and panel transforms for that surface/region to reset. This means that any Dataset transform such as Scaling, Region transforms such as Rotate or Translate (on the respective panels), or any interactive Dataset or Region transforms done prior to the detach will be undone. This may cause the surface to move out of view, for instance. Since a detached Surface or Region is not affected by further World transforms (such as Center), it may be difficult to reset the view to re-acquire the surface in some instances. Reset will reattach the Region or Surface to the hierarchy.

Note: A detached Region or Surface will be affected by World zooms (using [M3] when the transform Object is World and Action is Multi-Transform).

Chapter 7

7

Printing and Saving Images

Introduction

The **File** → **Save Image** pulldown menu has flyouts for saving the entire contents of the graphics window including multi-window layouts (Graphics), the current graphics window only (Window) or the 2D plot window (Plot) to a file. To print directly to a PostScript-compatible printer, use the **File** → **Print** pulldown. Several file formats are supported and are illustrated in **Figure 135**.



Note: The **Tools** → **Graphics Layout Size** sub-menu provides a way of easily setting the graphics window sizes for image output. The list offers several standard image sizes. Choosing one of these predefined sizes will cause the graphics window to be resized. The ability to resize the graphics window is redundantly available from **View** → **Graphics Layout Size** as well.



Note: It is possible to create printouts of the graphics window even if it has been partially covered by another application or minimized.

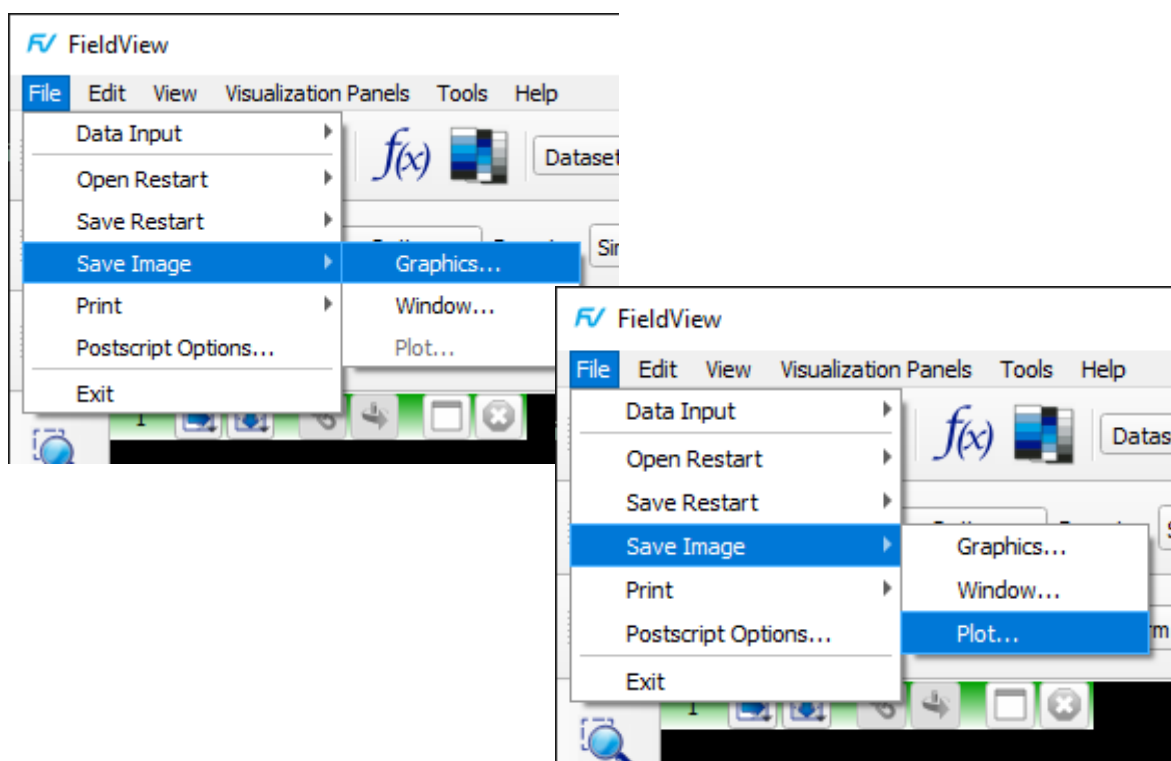


Figure 135 Save images to file

Note: Be aware that white objects output on white paper will not be visible.

Printing and Saving Images

How do I save my graphics or 2D Plot window to a file?

The **File** → **Save Image** pulldown menu has flyouts for saving the entire contents of the graphics window including multi-window layouts (Graphics), the current graphics window only (Window) or the 2D plot window (Plot) to a file. Several file formats are supported and are illustrated in **Figure 135**.

The display in the current window or the entire multi-window layout can be saved to one of **FieldView**'s supported image file formats. To save the current window, navigate to **File** → **Save Image** → **Window** and choose the desired format. To save the entire contents of the graphics window, including all the windows in the layout, navigate to **File** → **Save Image** → **Graphics** and choose the desired format. Saving a multi-window image to a file is illustrated in **Figure 136**. The graphics window has been split into six windows. **View** → **Graphics Layout Options** provides control over the multi-window hard-copy background color and the separator width, both of which affect the rendering of the spaces between windows in a multi-window rendering. The three windows at left contain coordinate planes showing three different scalars and have their background color set to

black. The three windows at right contain embedded plots and have their background color set to white. In the resulting BMP file, black separators are used to provide space between the windows.

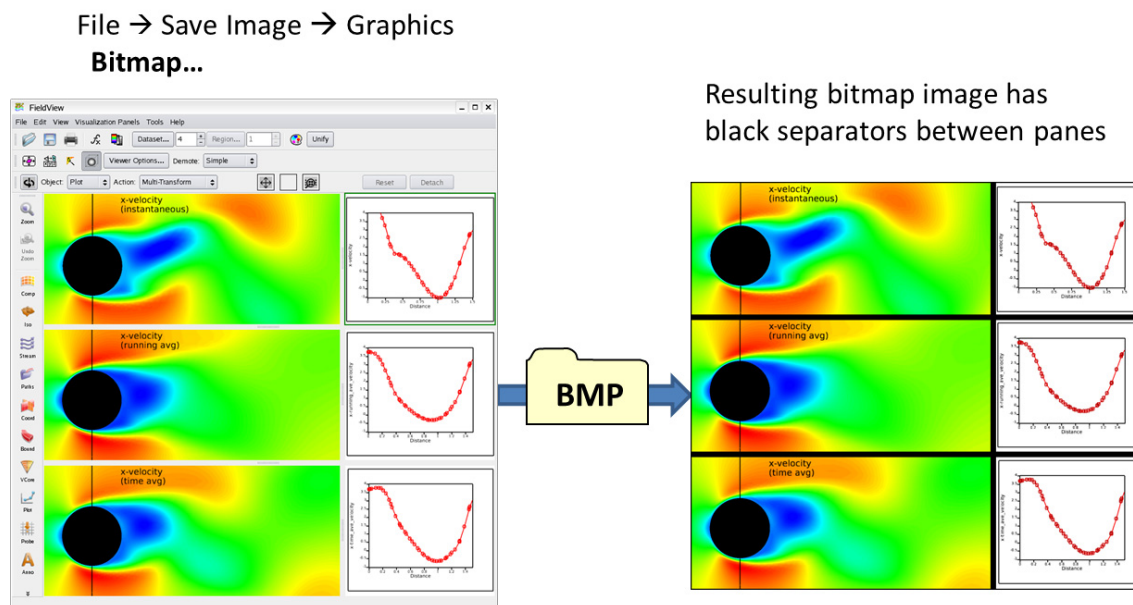


Figure 136 Saving a Multi-Window Image to File

How can display images or play animations created with FieldView?

FieldView's output images and animations are suitable for all widely used graphics programs and players, including ImageMagick's *display* and *animate* utilities, typically installed with Linux.

What is the difference between Encapsulated PostScript and Auto-Scaling PostScript?

Encapsulated PostScript files are intended for use inside another document. Your document preparation program will scale and rotate the image as you instruct it. However, you can also send an Encapsulated PostScript file directly to the printer. Please note that these Encapsulated PostScript files do not contain a preview image. As a result, the image itself will not show up in the document program, but will appear in the final print.

Auto-Scaling PostScript files are intended for stand alone (full-page) output. An Auto-Scaling PostScript file fills the page as much as possible, while still preserving the height:width ratio of the image. In addition, it will turn the image 90 degrees (landscape) if this will produce a larger image.

How can I get my background color in my PostScript prints?

Go to **File** → **Postscript Options...** On the Postscript Options panel check the **Use White Background with Black Border** option. When this button is on (the default) your background will always be white, and a black border will be drawn around your print. If you would like the background color in the graphics window to be used in your

print, turn this button off before pressing one of the print options. Note: This option will work only for graphics window PostScript prints.

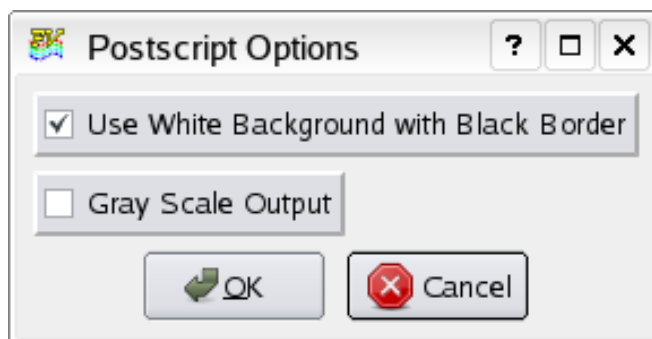


Figure 137 Postscript Options panel

What does the Gray Scale output option do?

If you press the Gray Scale option button on the Postscript Options panel, your print will come out as gray scale, even on a color printer. This is useful if you will be printing to a black and white printer, or will be using your print in a report that will be copied on a black and white copy machine. Prints that are made using the Gray Scale option will also be substantially faster than color PostScript prints. Note: The Gray Scale option is only available for the Graphics Window, not the Plot Display.

Can I control the resolution on my PostScript prints?

FieldView is set up to print at high resolution on most standard paper sizes. If you are printing on very large paper, or if you would like even higher resolution on your printer, you may set the environment variable: `FV_DPI`. This variable controls the number of image pixels per inch that will be calculated (based upon a standard paper size). By default this value is set equal to 120. Please note that changing this value will not affect the color banding (amount of colors) on your printout, and will use significantly more memory. Only change this value if the edges of objects appear jagged on your printout.



Note: The default `FV_DPI` value of 120 was chosen to give the best overall performance. If you want to see if increasing this value noticeably improves image quality, try creating and printing an image at `FV_DPI=120` and `FV_DPI=240`. If you do not see a significant improvement, then increasing the DPI setting will not help the output. Side effects of higher `FV_DPI` values are: i) more memory required - sometimes significantly higher, ii) move-draw lines will get thinner and thinner. Setting `FV_DPI` to 600 because your printer is a 600 dpi color printer is usually incorrect because the colors are 'dithered', as described in the next paragraph. Also, keep in mind that some printers have printing quality settings.

Unless you have a TrueColor printer (e.g. a dye sublimation printer), any colored object will be dithered when printing. Dithering is the simulation of color gradations by drawing

a pattern of different-colored dots. PostScript does not allow scalar-colored (gradually changing color) move-draw commands, which handle lines, vectors, most text, etc. So if you have scalar-colored vectors, for example, they are created in the PostScript file by **FieldView** as a bitmap. When printed on a non-TrueColor printer, they will be dithered. If you have a 75 dpi printer, these dithered objects will look worse than on a 300 or 600 dpi printer. However, increasing the DPI of the PostScript file **FieldView** creates will not significantly improve the results (on the same printer) unless (as stated above) you need to print on very large paper.

Do I need to save a file to capture my graphics or 2D plot window?

No. Using **Edit** → **Copy**, you can paste the contents of either the graphics window or the 2D plot window to the clipboard. The contents of your clipboard can then be pasted directly into most applications including Microsoft Office and Open Office.

*Can I have **FieldView** print directly to my printer?*

In the `fv/bin` subdirectory, there is a shell script called `fv_to_printer.sh.sample`. In order to instruct **FieldView** to send your file to a postscript compatible printer, you must edit this shell script and rename it to: `fv_to_printer.sh`. The best thing to do is to copy this script to your home directory and edit the file there. You will need the appropriate permissions to edit the files in `fv/bin` directory. **FieldView** will search the normal path hierarchy, searching your home directory first. The supplied script can be used to send the contents of your graphics or 2D plot window to any PostScript compatible output device. To enable this, edit the “print file” section of the script according to your device configuration. An example print command is shown in the script.

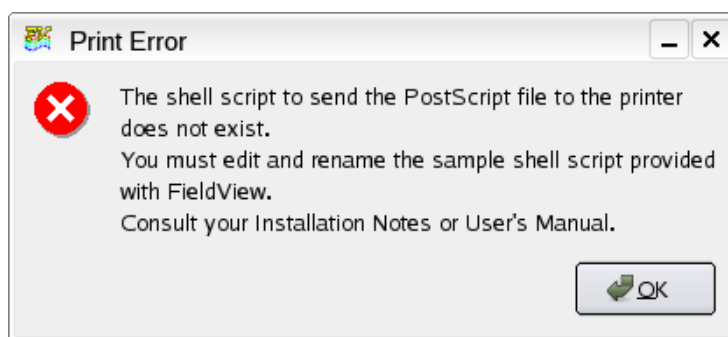


Figure 138 Error message when direct printing is not configured

FieldView Pixel Resolution

The actual number of pixels used in a PostScript image created by **FieldView** is calculated as follows:

1. Calculate the ratio of larger dimension of the graphics window to the smaller dimension. The default graphics window on 1280x1024 PC monitors is nearly square. The graphics window is 746x734 (width x height). So the default aspect ratio is:

$$\text{larger/smaller} = 746/734 = 1.016$$

2. The aspect ratio of an 8.5x11 inch sheet of paper is:

$$\text{larger/smaller} = 11/8.5 = 1.294$$

3a. If the aspect ratio of the graphics window is *less* than 1.294, then the smaller dimension of the graphics window is the limiting factor, and it fills the smaller (8.5 inch) dimension of the page. The number of pixels in the smaller dimension of the image is then:

$$\text{FV_DPI} * 8.5$$

where `FV_DPI` defaults to 120 in **FieldView**. In this case, the larger dimension of the image will have the pixel count:

$$\text{FV_DPI} * 8.5 * (\text{graphics window aspect ratio})$$

3b. If the aspect ratio of the graphics window is *greater* than 1.294, then the larger dimension of the graphics window is the limiting factor, and it fills the larger (11 inch) dimension of the page. The number of pixels in the larger dimension of the image is then:

$$\text{FV_DPI} * 11$$

where `FV_DPI` again defaults to 120 in **FieldView**. In this case, the smaller dimension of the image will have the pixel count:

$$\text{FV_DPI} * 11 / (\text{graphics window aspect ratio})$$

`FV_DPI` can be converted into actual dots per inch and vice versa. If the image is rendered by a PostScript renderer to a size other than 8.5x11 inch paper, the actual “dots per inch” will be different from `FV_DPI`. Note that EPS images will not render to different sizes unless embedded in a parent document - they never resize when printed “stand-alone”.

Roughly, if the desired image size is 1/3 the size of a piece of paper, then the actual dots per inch will be 3 times `FV_DPI`. More exactly, assuming uniform (non-distorted) scaling of the image, then

1. If the aspect ratio of the graphics window is *less* than 1.294, the smaller dimension of the graphics window is the limiting factor. Then the actual “dots per inch” is:

$$FV_DPI * 8.5 / (\text{smaller dimension of final image in inches})$$

2. If the aspect ratio of the graphics window is *greater* than 1.294, then the larger dimension of the graphics window is the limiting factor. Then the actual “dots per inch” is:

$$FV_DPI * 11 / (\text{larger dimension of final image in inches})$$

If you know the desired “dots per inch”, you can calculate the necessary FV_DPI by doing this in reverse:

1. If the aspect ratio of the graphics window is *less* than 1.294, the necessary value of FV_DPI is:

$$FV_DPI = (\text{desired dpi}) * (\text{smaller dimension of final image in inches}) / 8.5$$

2. If the aspect ratio of the graphics window is *greater* than 1.294, then the larger dimension of the graphics window is the limiting factor. Then the actual “dots per inch” is:

$$FV_DPI = (\text{desired dpi}) * (\text{larger dimension of final image in inches}) / 11$$

Possible Problems

My text appears blocky and gray.

The default color for text is gray (the default Geometry color). Text that is not black can be rendered in a dithered form and this usually results in a lower quality output. For best results, set the background color of the screen to white (so it looks like the paper) and then set your text color to black.

My text disappears when I create a PostScript print.

The default PostScript option does not print the background color. As a result, white text will be printed on white paper, and will not show up. To fix this problem, you should change the background color of the screen in **FieldView** to white (or something other than black), and change your text color to something other than white (black text tends to print the best), before creating your PostScript file.

Why do my contour lines and meshes appears jagged?

Black lines are drawn with the best appearance in PostScript. If possible, change line drawn objects to black.

Error Conditions

Window is not visible.

Selecting an icon'ed or absent window will cause the error message popup Window is not visible, and will not create the PostScript file.

Timeout error

After the PostScript file is printed, if you receive a second page from your printer indicating a timeout error, this indicates a problem with your print spooler. The spooler is supposed to place an end of job marker (usually a control-D) at the end of each job. If the printer does not receive this marker, a timeout error will occur. If you are receiving this error, you will need to have your spooler modified to append the end of job marker when necessary.

Chapter 8

Advanced Numerical Functions

8

This chapter will detail many useful techniques and formulas which can be used with the Function Formula Specification panel ("Function Calculator") to allow users to define quantities and visualize their data better. Note that the case of the function is irrelevant (i.e. "GRAD" will work as well as "grad").

Vector Quantities

In this section, we will describe how to create unit vectors, surface normals, vectors from scalars, and how to extract a component of a vector.

Unit Vectors

Unit vectors are: $\text{Unitx} = [1, 0, 0]$, $\text{Unity} = [0, 1, 0]$, $\text{Unitz} = [0, 0, 1]$

Unitx , Unity and Unitz can be used in the function calculator to compose vector quantities. The names are not case sensitive. An alternative way to define the unit vectors is to compute gradients:

X direction: $[1, 0, 0]$ is $\text{grad}(\text{"X"})$ or UnitX (with no quotes)
Y direction: $[0, 1, 0]$ is $\text{grad}(\text{"Y"})$ or UnitY (with no quotes)
Z direction: $[0, 0, 1]$ is $\text{grad}(\text{"Z"})$ or UnitZ (with no quotes)

Surface Unit Normals

Computational Surfaces:

For a J surface, the gradient of J is normal to the surface (pointing in the direction of increasing J). Therefore, in the **FieldView** formula panel, to get a unit normal for a J surface, use:

```
nrmlz(grad("J"))
```

where "nrmlz" normalizes the gradient (converts it into unit vectors). Remember this always points toward increasing J, not necessarily in the direction of flow, so you may need to use:

```
-nrmlz(grad("J"))
```

if you are dealing with flow quantities, and the flow is going toward decreasing J.

Iso-Surfaces:

For an Iso-Surface of pressure,

```
nrmlz(grad("pressure"))
```

are unit vectors normal to the Iso-Surface. They point in the direction of increasing pressure.

Coordinate Surfaces:

See [Unit Vectors](#) above.

Boundary Surface:

The surface normal is not reliably known to **FieldView**, unless the direction is marked in the PLOT3D or Unstructured data passed to **FieldView**. For viscous cases this will usually work:

```
grad(mag("velocity"))
```

(The vector function "velocity" may appear different for various reader formats). To see if this is likely to work create a vector function:

```
grad(mag("velocity"))
```

Make this the current vector function, change your surface type to vector and see if these vectors look like they are correct vector normals.

Create a Vector from a Scalar

Given the scalars: a, b, and/or c, we can convert them to a vector by creating the function:

```
UnitX*a + UnitY*b + UnitZ*c
```

Example:

Create a vector function which is the Velocity vector (in m/s) minus a 50 m/s freestream in the X-direction:

```
"Velocity" - UnitX*50.0
```

Extract a Component of a Vector

To extract the X component of a vector, in this case "Velocity",

```
VecX("Velocity")
```

To result in a vector rather than a scalar,

```
VecX("Velocity") * UnitX
```

since `UnitX = [1,0,0]`

Equal Length Vectors for Two Datasets

FieldView makes the default vector length (1) a certain length in pixels based on the size of the graphics window and the magnitude of the vector used for the vector field. Visualizations of multiple datasets in **FieldView** are independent of each other with respect to available functions, function min/max's, etc. This gives you a much greater ability to visualize multiple datasets. However, since the lengths of the vectors of each dataset will be determined independently of any other datasets in memory, the vector lengths of the first dataset may appear similar to the vector lengths of the second dataset, but actually stand for very different velocities.

The lengths of the vectors are determined by the local vector value and overall scaling is set by the normalization factor for that dataset (discussed below). The method described in this section will show you how to set the displayed vector lengths so that vectors from different datasets are approximately the same. That is, for example, a vector with a magnitude of 50 m/s in one dataset is the same length as a vector of 50 m/s in a second dataset, even though the normalization factors of the two datasets are not the same.



Important Note: The view of a given dataset affects the size of objects displayed. This includes scaling factors which, in turn, affect vectors. In order to create equal vector lengths, the two datasets must have the same view and dataset scaling. You can translate and rotate, but not zoom. Zooming will change the scale factor.

The way **FieldView** determines the size of vectors is by normalizing the vector field by the following:

$$N = |u|_{\max} + |v|_{\max} + |w|_{\max}$$

where

$|$ indicates absolute value
 $_{\max}$ indicates maximum value

Therefore, in order for two datasets to have representatively the same length vectors, N would have to be the same for both datasets. If this is not the case, you need to scale the vector by adjusting the

Length Scale found in the Vector Options panel. Creating equal length vectors cannot be properly done by scaling the vector function with the Function Formula Specification panel (CFD Calculator).

Using the first dataset's vector length as the baseline, the length scale factor for the second dataset needs to be:

$$\text{Length Scale} = N_2 / N_1$$

where

$$N_1 = |u_1|_{\max} + |v_1|_{\max} + |w_1|_{\max}$$

$$N_2 = |u_2|_{\max} + |v_2|_{\max} + |w_2|_{\max}$$

Example:

This normalization was performed on two of the datasets provided with **FieldView**, `bluntfin` (found in the `fv/examples` directory), and the F18 dataset used in the Basic Aerospace Tutorial ([Chapter 3](#) of the **User's Guide**, and also in the `fv/demo` directory):

	$ u_i _{\max}$	$ v_i _{\max}$	$ w_i _{\max}$	N_i
bluntfin	3.12293	1.88989	2.00529	7.01811
F18	.486598	0.49028	0.522133	1.499011

This gives a value of $N_2/N_1 = 0.21357$. Therefore, whatever the Vector Length used for the first dataset (`bluntfin`, in this case), the Vector Length used for the second dataset (F18) needs to be 0.21357 times this. The resulting Vector Length Scales used, velocity magnitude, length of displayed vectors and results were:

	Vector Length Scale	$ V $	Displayed Length (mm)	$ V $ for 1 (mm)
bluntfin	40	2.64544	160	60.48
F18	$40 \cdot (N_2/N_1) = 8.5428$	0.314374	19	60.44

Thus we see that for properly scaled vectors, a unit of velocity magnitude will display as (essentially) the same length vector (within the accuracy of hand length measurement).

Non-Rotating Velocities using Rotating Quantities

If the Velocity Vector found in the results file is defined for a rotating frame of reference, but a non-rotating frame of reference vector is desired for visualization, this can be done in **FieldView** using the Function Formula Specification panel ("CFD Calculator"; see [Chapter 2](#) and [Appendix A](#) of this **Reference Manual**).



Note: These formulas assume the vector field is velocity. If the vector is not velocity, different formulas are needed.

Note: The inverse conversion, from non-rotating to rotating velocities, is supported in **FieldView** by automatically creating the relative velocities. This is described in [Chapter 3](#) of this **Reference Manual**.

1. Export rotational frame of reference velocity components from the flow solver.
2. Note rotational frame of reference speed, ω , [rad/s]
3. Read the grid/results data into **FieldView**
4. Use the "CFD Calculator" to set up the following functions:

ω	= Ω , [rad/s] (define as a constant)
U_{x_rel}	= "u-velocity" - "Y" * " ω "
U_{y_rel}	= "v-velocity" - "X" * " ω "
V_{noro}	= UnitX * " U_{x_rel} " + UnitY * " U_{y_rel} " + UnitZ * " w -velocity"

5. Load the new vector " V_{noro} " into the Vector register. Plotting a surface as vectors will now display the new non-rotating vector field.

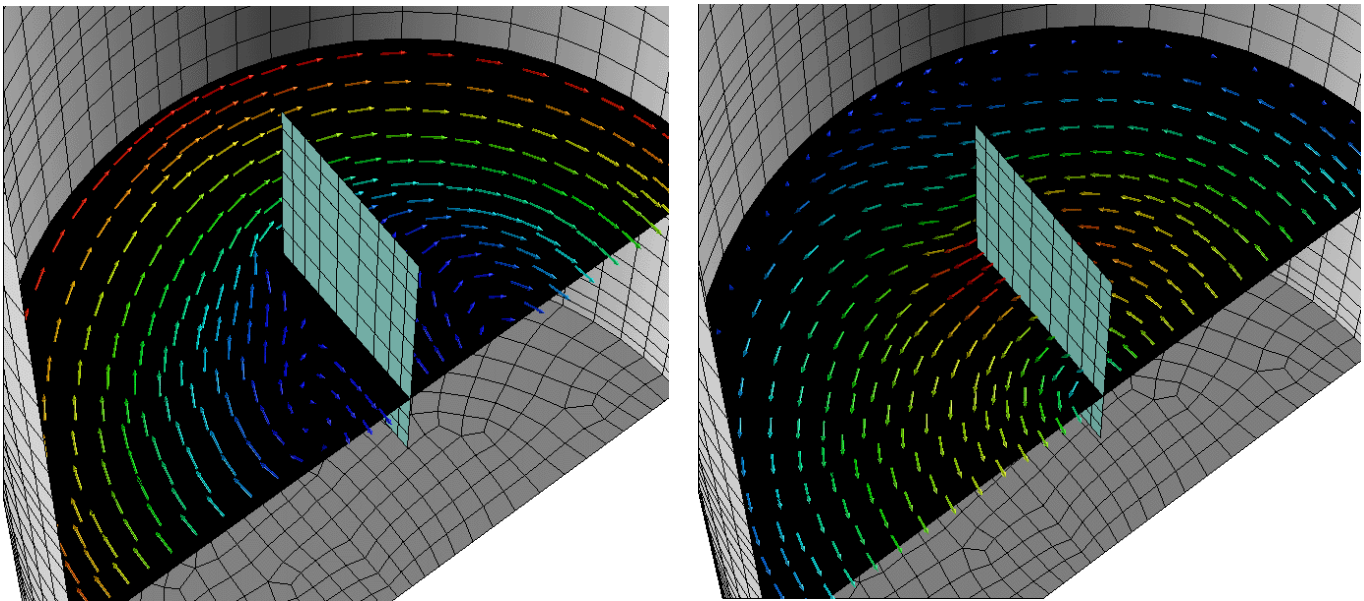


Figure 139 Rotating (left) and Non-Rotating (right) Vector Fields

Integral Quantities

In this section, we will describe how to compute line integrals, volume integrals and integrated forces.

Line Integrals

A line integral can be accurately approximated in **FieldView** by using thresholding. The following example explains how to integrate along an $I=\text{constant}$ line in a 2D dataset. The same kind of approach works for line integrals in 3D datasets.

Example:

Assuming that your 2D data is in the I & J directions (so $K=1$), make a $K=1$ Computational Surface and scalar color it with the function you want to integrate. Bring up the Function Specification panel and load up the Threshold register with "I". Turn on the THRESHOLDING button for the Computational Surface. Assume you want to integrate $I=5$. What you need to do is the following:

1. Set Min: on the Threshold slider to 4.99 by typing in this value.
2. Set Max: on the Threshold slider to 5.01 by typing in this value.
3. Bring up the Integration Controls panel. Press the Integrate Current button. Write down "Average value (integral/area)= NNN.NNNN" number.
4. Set Min: on the Threshold slider to 4.999 by typing in this value.
5. Set Max: on the Threshold slider to 5.001 by typing in this value.
6. Bring up the Integration Controls panel. Press the Integrate Current button. Write down "Average value (integral/area)= NNN.NNNN" number.
7. Set Min: on the Threshold slider to 4.9999 by typing in this value.
8. Set Max: on the Threshold slider to 5.0001 by typing in this value.
9. Bring up the Integration Controls panel. Press the Integrate Current button. Write down "Average value (integral/area)= NNN.NNNN" number etc.

The values for "Average value" should be converging. Two things will happen:

1. You will reach the precision of **FieldView** in the type-in fields.
2. The numbers for "Average value" will stop converging and may start to diverge. When this happens, go to the last "good" number written down - this is the closest **FieldView** can come to the correct value.

Volume Integrals

This section contains information regarding the calculation of the Volume of a region, not necessarily the volume integral of a Function, which can only be done under special circumstances.



Note: This method's results will depend heavily on the coarseness or fineness of the grid. Coarse grids will give poorer results and vice versa. Also, this can be used to calculate volumes but not volume integrals. That is, F cannot be varying, but must be a constant function.

FieldView can calculate some volumes if you use Gauss' divergence theorem. This theorem says that volume integral of " $\text{div}(F)$ " is equal to surface integral of " $F \cdot \text{surface-normal}$ " over the bounding surface of the volume, where " F " is your function.

$$\iint \mathbf{F} \cdot \mathbf{N} = \iiint \text{div} \mathbf{F}$$

$\text{div}(\mathbf{F})$ is the divergence of \mathbf{F} . \mathbf{F} must be a vector function. (If F_x is the x-component of \mathbf{F} , and F_y is the y-component of \mathbf{F} , and F_z is the z-component of \mathbf{F} , then the divergence of \mathbf{F} is:

$$\partial F_x / \partial x + \partial F_y / \partial y + \partial F_z / \partial z$$

which is a scalar function).

To get a volume integral, you must integrate over the bounding surface (or surfaces) of the volume, and you must know the surface normals. See the sections above on [Unit Vectors](#) and [Surface Unit Normals](#) for descriptions on how to do this.



Note: The surface of the volume must be closed (no gaps, no holes).

You can also use boundary surfaces or Iso-Surfaces to make volumes, and calculate the volume integral from the boundary surfaces or Iso-Surfaces.

The type of surface does not matter (computational or coordinate or boundary or Iso-Surface). If the surfaces form a closed volume, then Gauss' divergence theorem is valid.

Example:

We will compute the volume of a sphere created by an Iso-Surface in the dataset `bluntfin` found in the `examples` directory of your **FieldView** installation.

We need to create the vector function `["X", 0, 0]`. We use this function because the divergence of `["X", 0, 0]` is the scalar constant 1. Recall from the Unit Vectors section above that `UnitX` is the same as the vector function `[1, 0, 0]`. Therefore, the vector function we need is:

$$\mathbf{F} = ["X", 0, 0] = "X" * \text{UnitX}$$

For the dataset `bluntfin` used in this example, the sphere is located at `[1.0, 3.5, 2.5]`.

```
formula_restart_version: 1
sphere
sqrt(("X"-1)^2+("Y"-3.5)^2+("Z"-2.5)^2)
F
"X"*UnitX
```

First, create your isosurface, using the variable sphere as the iso function. Set the Vector Function to F. For this case, with a radius of 2.5 units, the volume should be $(4/3) * (\pi) * (2.5^3) = 65.44985$. **FieldView** gives approximately 64.2379, (in the Integration RESULTS section, look for the value of $\text{Int} (V \cdot N)$), yielding a relative error of about 1.8%. If you threshold the Iso-Surface with Z and reduce the maximum value to 2.5 (i.e. only allow the left hemisphere to show), the relative error reduces to about 1%. This is due to the fact that the `blunifin` grid is finer near the plate and coarser the farther away you get. Thus, the right hemisphere (the side with $Z > 2.5$) is less spherical than the left hemisphere. The finer the grid, the better the spatial resolution.

Integrated Force

The (pressure) force on a surface or surfaces can be calculated by integrating pressure over the surface(s).

To get the component of this force in a given direction, such as for the purpose of calculating drag or lift, it is necessary to know the surface normals.

If the user or the solver supplies surface normals N at grid points, then **FieldView** can integrate $(P * N_x)$, $(P * N_y)$, and $(P * N_z)$ as separate scalars, where $N_x = N \cdot (1, 0, 0)$, $N_y = N \cdot (0, 1, 0)$ and $N_z = N \cdot (0, 0, 1)$, N = surface normal, to get the components you want. Internally, **FieldView** can calculate surface normals, but it does not always know which direction is away from the surface (an airfoil, for example). This is only a problem for boundary surfaces. **FieldView** supports the passing of normal information for boundary surfaces for PLOT3D and **FieldView-Unstructured** files to avoid this problem.

For an Iso-Surface of a scalar quantity, the gradient of the scalar is normal to the surface, and should point in a consistent direction. You can make this into a unit vector with the `nrmlz` function in "Function Calculator". See the above sections on [Unit Vectors](#) and [Surface Unit Normals](#) for instructions on how to create the needed normals.

Once you have the surface normal, you can extract the X component of this with the `"vecX"` function in the formula calculator.

For example:

1. To get the components of force on a K Computational Surface, integrate:

```
VecX("Pressure"*(nrmlz(grad("K"))))
VecY("Pressure"*(nrmlz(grad("K"))))
VecZ("Pressure"*(nrmlz(grad("K"))))
```

2. To get the components of force on an Iso-Surface of Temperature, integrate:

```
VecX("Pressure"*(nrmlz(grad("Temperature"))))
VecY("Pressure"*(nrmlz(grad("Temperature"))))
VecZ("Pressure"*(nrmlz(grad("Temperature"))))
```

3. For a Boundary Surface whose normals are supplied, simply integrate Pressure on the surface (using Integrate Current Surface). The three normal-based components of force are automatically calculated.

Built-In CFD Functions

FieldView provides two built-in CFD functions, available on the Function Formula Specification Panel shown in [Figure 60](#). Both require a vector as input. It is expected that this input vector will be velocity, but the user is free to provide any vector as input. Below we show an example of the Q-criterion function in use.

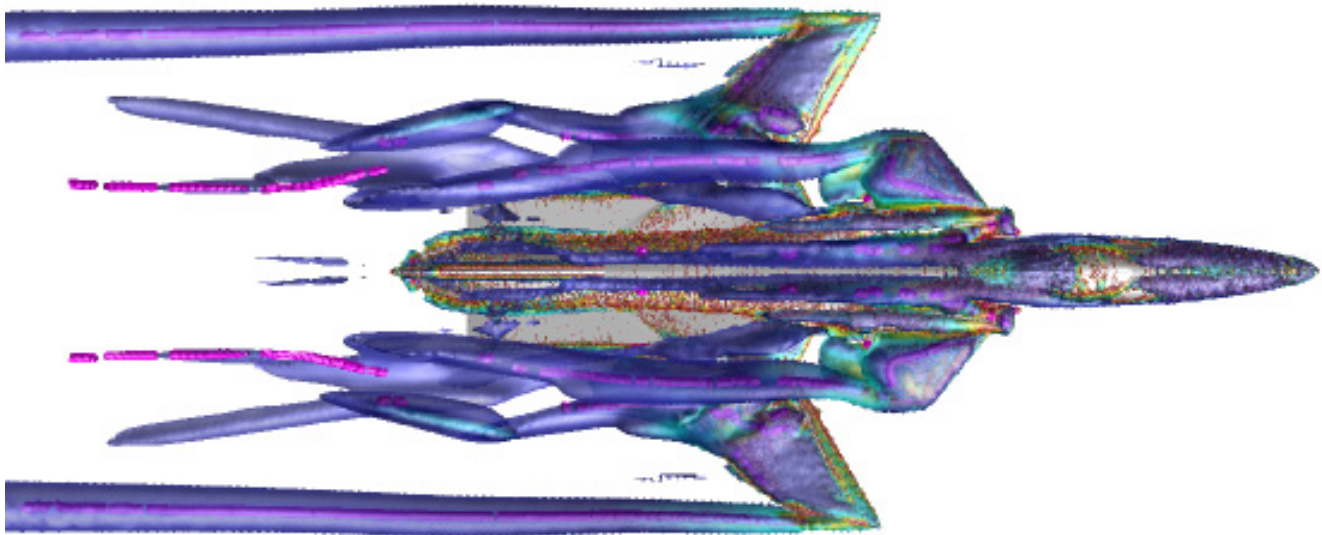


Figure 140 Q criterion feature detection

The Q-Criterion function is based on the general form of the criterion proposed by Hunt in 1988. The function is generalized such that no assumption is made on compressibility. This function computes Q , the second invariant of the gradient tensor for the vector provided as input. As a reminder, Hunt, Wray & Moin (1988) have proposed to define a vortex as a spatial region where Q is positive (and pressure is lower than the ambient one - a criteria generally dropped in literature). For this reason, an Iso Surface of a well chosen positive value of Q is often used to highlight vortices

Lambda2 (λ_2) criterion is based on the observation that, in regions where Lambda2 is less than zero, rotation exceeds strain, and in conjunction with a pressure minimum, indicates the presence of a vortex. Iso-surfaces should be created with the current value set to a negative value, corresponding to $\lambda_2 < 0$, with the specific value selected based upon characteristics of the flow (turbulence level, etc.). This offers significant improvements for capturing the vortex topology and geometry over other methods.

Miscellaneous Quantities

In this section, we will describe how to compute curve lengths and second derivatives.

Curve Lengths in Structured Geometries

To get the length of a scalar contour in a 2D structured grid, do the following:

1. Make the grid into a 3D grid by adding a second layer at any constant depth (in Z) you like. The second layer should have the X and Y coordinates as the original layer, as well as the same scalar/vector results.
2. Make an Iso-Surface of the desired scalar. Use Subset Inc = 1 for greatest accuracy.
3. Set the Iso-Surface value to halfway between the “low” and “high” scalar values, so it is in the middle of the transition region.
4. Integrate the Iso-Surface to get its surface area. Divide the surface area by the constant depth you used in step (1).

To get the length of a scalar contour in a 3D grid which is really a 2D axisymmetric problem, do the following:

1. Make an Iso-Surface of the desired scalar. Use Subset Inc = 1 for greatest accuracy.
2. Set the Iso-Surface value to halfway between the “low” and “high” scalar values, so it is in the middle of the transition region.
3. Integrate the Iso-Surface to get its surface area. Divide the surface area by the angle in radians covered by the axisymmetric problem's grid.

Second Derivatives

Second derivatives of a given function can be viewed with **FieldView**. The following is an example for the second derivative of Temperature, T . Vectors are designated by showing their components in square brackets $[\]$.

On the Functions panel, select Create to create a new function. Use the `grad` (gradient) and `VecY` (Y projection) functions:

<code>grad("T")</code>	gradient of $T = [\partial T / \partial x, \partial T / \partial y, \partial T / \partial z]$
<code>VecY(grad("T"))</code>	Y component of the gradient = $\partial T / \partial y$
<code>grad(VecY(grad("T")))</code>	gradient of $\partial T / \partial y = [\partial^2 T / \partial x \partial y, \partial^2 T / \partial y^2, \partial^2 T / \partial y \partial z]$
<code>VecY(grad(VecY(grad("T"))))</code>	Y component of gradient of $\partial T / \partial y = \partial^2 T / \partial y^2$

Warning: Because the derivatives are calculated from discrete values of temperature, and not a continuous function, they are only approximate. For this reason, the second derivative is less accurate than the first derivative. To get better accuracy, use a finer grid (more grid points).

Rotating Quantities

By using the Function Formula Specification panel (“CFD Calculator”), you can properly define the pressure coefficient and related quantities (using the local dynamic pressure or tip dynamic pressure for normalization) for rotating problems such as propellers and helicopter rotors.

The formula used for the pressure coefficient (C_p [PLOT3D]) used by **FieldView** for PLOT3D Q data is:

$$C_p \equiv (p - p_{inf}) / q_{inf}$$

where the dynamic pressure, q_{inf}

$$q_{inf} \equiv 0.5 \rho_{inf} V_{inf}^2$$

In order to compute the proper pressure coefficient, the *local* dynamic pressure needs to be used:

$$q_{local} \equiv 0.5 \rho_{inf} V_{ref}^2$$

where V_{ref} is the local velocity of the radial section of the rotating blade:

$$V_{ref} \equiv \Omega r$$

where Ω is the angular velocity (radians/sec) and r is the section radius. In **FieldView**, formulas can reference other formulas, so you can define a formula for q_{local} , and then define a formula for C_p based on q_{local} .

If the blade is along one of the coordinate axes, then you can use one of the existing **FieldView** functions (see [Appendix A](#) of this **Reference Manual**):

$$\begin{aligned} R_{cyl}: & (X^2+Y^2)^{.5} \\ & (X^2+Z^2)^{.5} \\ & (Y^2+Z^2)^{.5} \end{aligned}$$

for the section radius, r . Otherwise, you will have to define your own formula for the section radius. Then, to calculate q_{local} , you will need to plug in values for Ω and ρ_{inf} . To calculate C_p , you will need also need p and p_{inf} .

The PLOT3D functions (such as C_p) built into PLOT3D and into **FieldView** assume standard normalization of the quantities in the Q file, namely:

$$\begin{aligned}\rho_{\text{inf}} &= 1 \text{ (freestream density)} \\ c_{\text{inf}} &= 1 \text{ (free-stream speed of sound)}\end{aligned}$$

This is explained in [Appendix B](#) of this **Reference Manual**.

The PLOT3D and **FieldView** calculation of pressure is:

$$\begin{aligned}p &= (\gamma - 1) (E - 0.5 \rho V^2) \\ &= (\gamma - 1) (Q5 - 0.5 * \text{sqrt}[(Q2)^2 + (Q3)^2 + (Q4)^2] / Q1)\end{aligned}$$

where Q5 is stagnation energy, Q1 is density (ρ), and Q2, Q3, and Q4 are the components of momentum vector (all of which, as well as pressure, are available in **FieldView**). Gamma (γ), by default, is 1.4 in this and in other PLOT3D functions. You can set your own value of gamma by using a command-line switch when starting **FieldView**. This set value is then used in calculating all PLOT3D functions. See [Command Line Switches](#) in [Chapter 1](#) of the **User's Guide** for more information.

Finally,

$$\begin{aligned}P_{\text{inf}} &= \rho_{\text{inf}} c_{\text{inf}}^2 / \gamma \\ &= 1 / \gamma\end{aligned}$$

Assuming *standard* normalizations, then the modified pressure coefficient becomes:

$$\begin{aligned}C_p &= (p - P_{\text{inf}}) / q_{\text{local}} \\ &= (\text{Pressure [PLOT3D]} - (1 / \gamma)) / (0.5 \Omega^2 r^2)\end{aligned}$$

Chapter 9

Building Field-View Plugins

9

Using **FieldView**'s toolkit option, users can add their own functions or data readers to the program. The toolkit option builds "plugin" modules that add your functions and data readers to **FieldView** without the need to modify the **FieldView** executable. The routines may be written in either FORTRAN or C and, once made into a plugin, will behave just like any of the predefined data readers or functions.

Much of the low level documentation on these features has been included in the following files:

```
ftn_register_data_readers.f  
register_data_readers.c
```

These files can be found as part of the standard **FieldView** installation in the `/user` subdirectory. Anyone interested in building a plugin reader should consult these files.

Adding User-Defined Functions



Note: **FieldView** contains a built-in formula creation panel which allows you to create formulas interactively, rather than using the toolkit interface. However, the toolkit interface is still available, and is described below. For more information on the Function Formula Specification Panel, please see [Chapter 2](#) of this **Reference Manual** and [Chapter 3](#) of **Working with FieldView**.

FieldView provides a number of built-in functions that may be performed on input data. For example, in looking at grid data, you may look at various cylindrical or spherical radius functions, or angle (arctangent) functions. In looking at PLOT3D Q-file results, you may use any of the PLOT3D V3.6 functions. [Appendix A](#) of this **Reference Manual** has a complete list of the functions available in **FieldView**.

If the function you want is not available, and particularly if your results are not in PLOT3D Q-file format, you may wish to write your own functions and build them into **FieldView**. Doing this is a four step process:

- Step 1: Decide on the function's input requirements
- Step 2: Write the function
- Step 3: Register the function (so that **FieldView** knows about it)
- Step 4: Build the shared-lib toolkit.

Step 1: Decide on Your Function's Input Requirements

The arguments passed to your function depend on its input requirements. **FieldView** is told about these requirements when you register the function in Step 3, and then is able to make decisions about whether it is safe to call your function in the absence of results data (for example). There are two questions you need to answer about your function's input requirements:

(a) does it need Q-file data? (b) does it need results data other than Q-file data, such as data from a function file or data from a non-PLOT3D data reader?

All user-defined functions are provided with grid geometry, and with the previous contents of the current function register. If you tell **FieldView** during the registration step (Step 3) that your function also needs Q-file data, then the function will also be provided with Q data; however, it will not appear in the list of available functions until a Q file has been read. If you tell **FieldView** that your function needs non-Q results data, then the function will be provided with this data, but will not appear in the list of available functions until non-Q results have been read.

Step 2: Write Your Function

Example functions showing the proper input/output arguments and their meaning are given in `fv/user/user_defined_functions.f` for structured grids and `fv/user/user_unstruct_functions.f` for unstructured grids. This FORTRAN source file shows examples of functions with all possible combinations of input requirements. Functions can output either scalar or vector quantities. Functions with scalar output can be loaded into any of the 3 scalar-valued registers in **FieldView**: Iso-Surface, Scalar (for scalar coloring), and Threshold. Functions with vector output can only be loaded into the **FieldView** Vector Register.

Functions may be written in either C or FORTRAN. **FieldView** is told which language you used during the registration (Step 3). The arguments to a C function are the same as those in the FORTRAN examples, except that all arguments should be declared as pointers. Handling of multi-dimensional arrays with adjustable dimensions in C is up to the writer of the routine.

Step 3: Register Your Function

FieldView only knows about functions that have been registered by editing one of two source files: `fv/user/ftn_register_functions.f` (for FORTRAN language functions) or `fv/user/register_functions.c` (for C language functions). All functions added to **FieldView** must be registered in one of these two central locations. If you want to have several versions of **FieldView**, each with different sets of user-defined functions, then you need to have several versions of these two files. In any case, the edited versions of these two files, together with the source that you wrote in Step 2, should all

be in a single directory. This directory can be `fv/user`, but it can also be any other directory. Each file contains instructions on how to register your functions, which involves specifying its input requirements, classifying its output (scalar-valued or vector-valued), giving it a name that will appear on the list of available functions, specifying the name of the function (subroutine) you wrote in Step 2.

Step 4: Build the Shared-Lib Toolkit

At the end of Step 3, you should have a single directory containing edited versions of the function registration files, together with source for the functions you wrote in Step 2. If you or your site also wish to include user-defined data readers, the registration files and source for these should be in the same directory (see the section on [Adding User-Defined Data Readers](#) for more details).

You should change (`cd`) to this directory, and then type in the command:

```
make_fv
```

to build a **FieldView** plugin. If you have failed to add `fv/bin` to your command search path and properly set the environmental variable `FV_HOME` (suggested in the **FieldView** installation procedure), then you will need to type in the full path name of `make_fv` (located in the `fv/bin` subdirectory).

This is a script that uses the Unix `make` utility or the Windows `nmake` utility to compile all edited source files in the current directory, and then bind the resulting object files into a **FieldView** plugin. In Windows, this script is named `make_fv.bat`. You may need to edit `make_fv` (or `make_fv.bat`) to use the correct compiler names and flags for your compilers. For Unix, you may also need to edit the file `ld_fv` so the Fortran and C compiler libraries match your compilers. Further information can be found inside the script.

Install the plugin as directed by `make_fv`.

To build or use plugins for **FieldView** servers (client-server mode), see the section on Using User Defined Plugins with a **FieldView** Server.

It is possible to mistakenly create and register a function with a name that conflicts with 'reserved' names in **FieldView**. If you attempt to use a reserved name, you will be presented with a pop-up message indicating that there is a naming conflict. Appendix B provides a list of the geometric functions which are always present. The reserved names in **FieldView** also depend on the type of data that has been read. Appendix B also provides a list of the PLOT3D scalar function names which are present when you read a PLOT3D dataset, which intentionally have the suffix `[PLOT3D]` to make the incidence of variable name conflicts less likely. Similarly, OVERFLOW data has scalar functions designated with the `[OVERFLOW]` suffix. **FieldView** also automatically creates shock-related formulas with the suffix `[shock]` and face-based scalars and vectors with the suffix `[BNDRY]`. Note that if any formulas are defined, they may also conflict with solver variables. Since formulas span all datasets in a **FieldView** session and are not erased by replacing datasets, formulas created for one dataset may conflict with some appended or replacement datasets.

Adding User-Defined Data Readers

Using **FieldView**'s toolkit option, users can add their own functions or data readers to the program. The toolkit option builds "plugin" modules that add your functions and data readers to **FieldView** without the need to modify the **FieldView** executable. The routines may be written in either FORTRAN or C and, once made into a plugin, will behave just like any of the predefined data readers or functions.

FieldView supports a number of input data formats but if the input format you want is not available, you may wish to write your own input data reader and build it into **FieldView**.

Creating a reader is a four step process:

- Step 1: Decide on the data reader's input requirements
- Step 2: Write the data reader
- Step 3: Register the data reader (so that **FieldView** knows about it)
- Step 4: Build the shared-lib toolkit.

Step 1: Decide on Your Data Reader's Input Requirements

FieldView supports two types of data readers:

1. "Split file" data readers, where the grid geometry and the results are in separate files (similar to PLOT3D XYZ and Q files), and
2. "Combination file" data readers, where the grid and results are in a single file.

The data reader interface (number of functions/subroutines you provide, and their arguments) depends on which type of data reader input you require. **FieldView** is told about the data reader input type when you register the data reader in Step 3. If your solver supports both "two file" and "combination file" data, you should write two data readers. They may, of course, share a great deal of the same programming.

Step 2: Write Your Data Reader

All **FieldView** data readers have two phases: a query phase, which returns some basic information such as grid sizes and variable names, and a read phase, which transfers the grid and/or results data to **FieldView**. Therefore, for each file type, you need to register two FORTRAN subroutines or C functions: one for the query phase and one for the read phase.

Combined File

If your data reader is a "combination file" data reader, then you need to write two FORTRAN subroutines or C language functions:

1. A subroutine or function used by **FieldView** to query information about the input file (number of grids, sizes of the grids, number of results variables per grid point, etc.).
2. A subroutine or function to read the grids and results from the file, one grid at a time.

Example source showing the proper input/output arguments and their meaning is given in `fv/user/user_combined_file_read.f` for structured grids and `fv/user/user_unstruct_combined.f` for unstructured grids. This FORTRAN source file shows examples of both the query subroutine and the “read one grid” subroutine for the combination file case.

Split File

If your data reader is a “split file” data reader, then you need to write four FORTRAN subroutines or C language functions:

1. A subroutine or function to query the grid geometry file, which returns information such as number of grids and their sizes.
2. A subroutine or function to read the grids from the grid file, one grid at a time. Example source for these first two subroutines is given in `fv/user/user_grid_file_read.f` for structured grids and `fv/user/user_unstruct_grid.f` for unstructured grids.
3. A subroutine or function to query the results file, which returns information such as the number of results variables per grid point.
4. A subroutine or function to read the results from the results file, one grid at a time. Example source for these last two subroutines is given in `fv/user/user_results_file_read.f` for structured grids and `fv/user/user_unstruct_results.f` for unstructured grids.



Note to C Programmers: The arguments to a C data reader are the same as those in the FORTRAN examples, except that all arguments should be declared as pointers. The file-name argument passed to the query functions should be declared `char *`. The `var_names` argument should be declared `char *` and the start of each variable name should be separated by exactly 80 characters in this array. The `var_names` array is initialized with blanks by **FieldView**; it is not necessary to pad your variable names with blanks or terminate the names with nulls (although it does no harm). Handling of multi-dimensional arrays with adjustable dimensions in C is up to the writer of the routine. The file passed in to the query functions should be opened in the following way:

```
FILE *fp, *fv_open();

fp = fv_open(iunit, fname);
```

You should use `fv_open` instead of the standard UNIX `fopen` (or `open`) so that **FieldView** knows about the open file and can close it in the event of an error. A file opened with `fv_open` can be accessed with the same standard UNIX calls as after using `fopen`, such as `fread` or `fscanf`.



Unstructured Grids Note: When writing the data reader, the ordering of the nodes and faces becomes very important. To ensure that your elements have the proper ordering, please refer to **Appendix D** of this **Reference Manual**.

Transient Note: If you have transient data that you wish **FieldView** to recognize, see the description at the end of this chapter for information about how this may be accomplished.

Enabling the Passing of Constants to GUI Buttons

FORTTRAN users can pass constants to the GUI buttons located in the function panel. One of the below routines should be called from the `user_results_file_read` file. It should be called for each grid read in with the current grid number as the first argument.

The corresponding GUI buttons will be enabled in the function calculator, and the constants may be used in **FieldView** formulas.

```
integer ngrid
real*4 fsmach, re, alpha, time
real*4 gamma, pinf, tinf, rgas
```

```
c This will set up all PLOT3D constants.
```

```
c
      call ftn_set_q_constants(ngrid, fsmach, alpha, re, time)
```

```
c This will set up all PLOT3D and WIND constants.
```

```
c
      call cfl_set_zone_constants(ngrid, fsmach, re, alpha, time,
+   gamma, pinf, tinf, rgas)
```

Step 3: Register Your Data Reader

FieldView only knows about data readers that have been registered by editing one of two source files: `fv/user/ftn_register_data_readers.f` (for FORTRAN language data readers) or `fv/user/register_data_readers.c` (for C language data readers). All data readers added to **FieldView** *must* be registered in one of these two central locations. If you want to have several versions of **FieldView**, each with different sets of user-defined data readers, then you need to have several versions of these two files. In any case, the edited versions of these two files, together with the source that you wrote in Step 2, should all be in a single directory. This directory can be `fv/user`, but it can also be any other directory. Each file contains instructions on how to register your data reader(s), which involves calling the appropriate registration subroutine/function (two-file or combination-file), giving the data reader a name that will appear on the Data Files & Functions pull-down menu and specifying the names of the subroutines/functions you wrote in Step 2.

Step 4: Build the Shared-Lib Toolkit

At the end of Step 3, you should have a single directory containing edited versions of the data reader registration files, together with source for the data readers you wrote in Step 2. If you or your site also wish to include user-defined functions, the registration files and source for these should be in the same directory (see the section on Adding User-Defined Functions to **FieldView** for more details).

You should change (`cd`) to this directory, and then type in the command:

```
make_fv
```

to build a **FieldView** plugin. If you have failed to add the `fv/bin` subdirectory to your command search path (suggested in the **FieldView** installation procedure), then you will need to type in the entire path instead.

This is a script that uses the Unix `make` utility or the Windows `nmake` utility to compile all edited source files in the current directory, and then bind the resulting object files into a **FieldView** plugin. In Windows, this script is named `make_fv.bat`. You may need to edit `make_fv` (or `make_fv.bat`) to use the correct compiler names and flags for your compilers. For Unix, you may also need to edit the file `ld_fv` so the Fortran and C compiler libraries match your compilers. Further information can be found inside the script.

Install the plugin as directed by `make_fv`.

To build or use plugins for **FieldView** servers (client-server mode), see the section on Using User Defined Plugins with a **FieldView** Server.

Writing a User-Defined Reader

To successfully build a User Defined Reader, Fortran subroutines or C functions must be written according to certain specifications. **FieldView** supports 4 types of data readers:

1. Unstructured data with separate files for grid (geometry) data and for results (solution) data.
2. Unstructured data with a single combined file for grid (geometry) data and results (solution) data.
3. Structured data with separate files for grid (geometry) data and for results (solution) data.
4. Structured data with a single combined file for grid (geometry) data and results (solution) data.

The `fv/user` toolkit directory contains examples of each of these types of readers. The process of creating a data reader is illustrated below with an annotated example of unstructured data with separate files for grid and for results. This example, based on the example in the toolkit directory, actually reads in some types of PLOT3D-format structured data, and converts the data into unstructured grids and results for **FieldView**.

Example of Unstructured Grid File Data Reader

Returning Information from the Grid

The first section contains the subroutine to query the unstructured grid. The names of the subroutines in a data reader are arbitrary, as long as they are communicated to **FieldView** when "registering" the data reader. However, it is strongly recommended that you begin all subroutine names with the word "user", as is done in this example. Any other names may conflict with global names within **FieldView** and cause program crashes.

```
subroutine user_query_unstruct_grid (fname, lenf, iunit,
+      max_grids, max_face_types, num_grids, num_nodes,
+      num_face_types, face_type_names, wall_flags, iret)
```

Variable Definitions

Input:

`fname` = file name
`lenf` = length of file name
`iunit` = "unit number" you should use when opening the file (**FieldView** will take care of closing the file)
`max_grids` = maximum number of grids allowed
`max_face_types` = maximum number of face types allowed

Output:

`num_grids` = number of grids that will be read from data file
`num_nodes` (array of node counts for all grids that will be read)
`num_face_types` = number of boundary face types
`face_type_names` (array of face-type names)
`wall_flags` (array of flags for the "wall" behavior of boundary faces)
`iret` = 0 (success)
 = 1 (failure)



Note on `face_type_names`: The grid query subroutine should identify the "interesting" face types that might occur in grid files from your solver. It is not necessary to identify exactly which face types actually occur in any one grid file. When you create boundary faces, you will be able to assign to a boundary face any of the face type numbers between 1 and `num_face_types`. Each face type must be assigned a name (character string) in the `face_type_names` array, which is initialized to blanks. Note that the number of face types must be less than or equal to `max_face_types`!

Note on `wall_flags`: The `wall_flags` argument is obsolete. It is present for compatibility with the older deprecated version of this query routine, which you can see in `user_unstruct_grid.f.deprecated`. You must have a `wall_flags` array argument, but you do not need to fill in the `wall_flags` array. It is ignored by **FieldView** if you use the grid element creation routines described below in subroutine `user_read_unstruct_grid`.

Arguments and Variables for Subroutine

The following line causes the Fortran compiler to complain about any variables that are not explicitly declared. This is useful for catching misspellings.

```
implicit none
```

Arguments for the subroutine:

```
character*(*) fname
integer lenf, iunit, max_grids, max_face_types
integer num_grids, num_nodes(max_grids)
integer num_face_types, wall_flags(max_face_types)
integer iret
```

Note: You *must* declare "`face_type_names`" as an array of `character*80`, as in the following line:

```
character*80 face_type_names(max_face_types)
```

Local variables in this example:

```
integer istat, j
integer BYTES_PER_INTEGER
parameter (BYTES_PER_INTEGER = 4)
integer MAX_GRIDS_IN_FILE
parameter (MAX_GRIDS_IN_FILE = 100)
integer idims(MAX_GRIDS_IN_FILE)
integer jdims(MAX_GRIDS_IN_FILE)
integer kdims(MAX_GRIDS_IN_FILE)
common /user_grid_dims/ idims, jdims, kdims
```



Note on Common Block Names: It is strongly recommended that you begin any common block names with the word "user". Any other names may conflict with global names within **FieldView** and cause program crashes. The following is an example of a safe common block name: `common /user1/ mydata, moredata integer mydata, moredata`

Open the File

FieldView provides "open_binary" for opening a binary file (usually written by a C program) in a FORTRAN subroutine.

```
call open_binary (iunit, fname(1:lenf), istat)

if (istat .ne. 0) then
    iret = 1
    go to 900
endif
```

For formatted data, use this `open` command:

```
open (unit=iunit, file=fname(1:lenf), status='OLD',
+      form='FORMATTED', iostat=istat)
```

For unformatted data, use this `open` command:

```
open (unit=iunit, file=fname(1:lenf), status='OLD',
+      form='UNFORMATTED', iostat=istat)
```

If your data file cannot be opened with an ordinary C or FORTRAN `open` or `open_binary` because it is a database, for example, then **FieldView** will not automatically close it or clean up when the data read is complete. To handle this, we provide a way for you to register your own function to do the closing and cleaning up.

For FORTRAN:

```
external my_close_sub
call ftn_reg_user_close_file (my_close_sub)
```

For C:

```
register_user_close_file (my_close_func)
```

The FORTRAN subroutine or C function that you registered will be called by **FieldView** with no input or output arguments. Your function should close any open databases and free any temporary data.

Determine the Number of Grids

The following example is for a file in single-grid PLOT3D format:

```
num_grids = 1

if (num_grids .gt. max_grids) then
    iret = 1
    go to 900
endif
```

You can handle the problem of too many grids by returning immediately (after setting "iret") or by resetting `num_grids` to `max_grids` so that fewer grids are read by **FieldView**. You must be careful to skip over any data for the unused grids if you choose the second approach.

Determine the Number of Nodes in Each Grid

The following example is for a file in PLOT3D binary data:

```
do 100 j = 1, num_grids
    call read_binary (iunit, BYTES_PER_INTEGER, idims(j), istat)
    if (istat.ne.0) go to 101
    call read_binary (iunit, BYTES_PER_INTEGER, jdims(j), istat)
    if (istat.ne.0) go to 101
    call read_binary (iunit, BYTES_PER_INTEGER, kdims(j), istat)
    if (istat.ne.0) go to 101
100 continue
101 continue

if (istat .ne. 0) then
    iret = 1
    go to 900
endif

do 150 j = 1, num_grids
    num_nodes(j) = idims(j) * jdims(j) * kdims(j)
150 continue
```

For formatted data, use this read command:

```
read (iunit, *, iostat=istat)
+      (idims(j), jdims(j), kdims(j), j = 1, num_grids)
```

For unformatted data, use this read command:

```
read (iunit, iostat=istat)
+      (idims(j), jdims(j), kdims(j), j = 1, num_grids)
```

Determine Number of Boundary Face Types

For this example, we will create two boundary types at the minimum and maximum values of I in the structured PLOT3D grid.

```
num_face_types = 2
face_type_names(1) = 'I=1'
face_type_names(2) = 'I=Imax'
```

```
iret = 0
```

```
900 continue
    return
end
```

Reading the Grid

This subroutine reads one grid from the file:

```
subroutine user_read_unstruct_grid (iunit, igrd, nodecnt,
+      xyz, iret)
```

Variable Definitions

Input:

```
iunit      = "unit number" that was used when opening the file
igrd       = which grid to read this time
nodecnt    = number of nodes in this grid (as returned by query routine)
```

Output:

```
xyz        (array of XYZ coordinates of the grid points)
iret       = 0 (success)
           = 1 (failure)
```



Note: This subroutine will be called once for each grid returned by the query subroutine. The first call to this routine will have `igrd=1`, the next `igrd=2`, and so on until `igrd=num_grids`.

Arguments and Variables

The following line causes the Fortran compiler to complain about any variables that are not explicitly declared. This is useful for catching misspellings.

```
implicit none
```

Arguments to this subroutine:

```
integer iunit, igrd, nodecnt
real xyz (nodecnt, 3)
integer iret
```



Note: The `xyz` array *must* be dimensioned as follows. The last array dimension of the "xyz" array is used to specify which coordinate is being referenced. A value of 1 means X, 2 means Y, and 3 means Z.

Functions used in this example:

```
integer ftn_create_boundary_face
```

Local Variables:

```
integer i, imax, istat, j, jmax, k, kmax, nwords, normals_flag
integer node_ids(8), wall_info(6)
integer BYTES_PER_INTEGER, BYTES_PER_REAL
parameter (BYTES_PER_INTEGER = 4)
parameter (BYTES_PER_REAL = 4)

integer A_WALL, NOT_A_WALL
parameter (A_WALL = 7)
parameter (NOT_A_WALL = 0)

integer MAX_GRIDS_IN_FILE
parameter (MAX_GRIDS_IN_FILE = 100)
integer idims(MAX_GRIDS_IN_FILE)
integer jdims(MAX_GRIDS_IN_FILE)
integer kdims(MAX_GRIDS_IN_FILE)
common /user_grid_dims/ idims, jdims, kdims
```

Fill in the XYZ Coordinates of Grid

The following example is for a file that is in binary PLOT3D grid format:

```
nwords = 3 * nodecnt
```

```
call read_binary (iunit, nwords * BYTES_PER_REAL, xyz, istat)
if (istat .ne. 0) go to 900
```

Use this read statement for formatted data:

```
read (iunit, *, iostat=istat) xyz
```

For unformatted data use:

```
read (iunit, iostat=istat) xyz
```

Creating Grid Elements

The following describes how to create grid elements (cells) and boundary faces as separate entities. There is an older deprecated method of creating grid elements and boundary faces at the same time. In the older method, boundary faces are defined as faces of grid elements. This method is described in the sample source file `user_unstruct_grid.f.deprecated`. If you use the deprecated method, you will not be able to read boundary results (quantities that are defined only on boundaries, such as wall quantities). Also, you will not be able to get boundary surface integrals that involve surface normals.

The grid elements (cells) and boundary faces are defined in terms in the node numbers that belong to the grid element or boundary face. You can create the grid elements first, or the boundary faces first, or mix them together (some grid elements, then some boundary faces, then more grid elements, and so on). There is no requirement that boundary faces are also faces of grid elements.

The node numbers used to define grid elements and boundary faces are indices into the `XYZ` data for this grid. A node number of 1 means the first node in the `XYZ` coordinates array for this grid. If you have multiple grids, the node numbers start at 1 inside each grid.

See the **FieldView User's Guide** for a description of each of the available grid element types. It is important that the order of the nodes follow the rules given, to avoid elements that are "twisted" (intersect themselves).

The available boundary face types are 3 nodes (triangles) or 4 nodes (quadrilaterals). In this example, we convert the specified PLOT3D grid to hexahedra (bricks), and then create boundary faces for selected grid boundaries.

Convert the specified PLOT3D grid to hexahedron elements:

```
imax = idims(igrid)
jmax = jdims(igrid)
kmax = kdims(igrid)
do 100 k = 1, kmax - 1
do 100 j = 1, jmax - 1
do 100 i = 1, imax - 1
```

In a PLOT3D grid, the nodes are ordered as follows:

```
(1,1,1), (2,1,1), ..., (imax,1,1)
...
(1,jmax,1), (2,jmax,1), ..., (imax,jmax,1)
(1,1,2), (2,1,2), ..., (imax,1,2)
...
(1,jmax,kmax), (2,jmax,kmax), ..., (imax,jmax,kmax)
```

Thus, the position of node (i, j, k) in the PLOT3D grid is:

$$i + (j-1)*imax + (k-1)*imax*jmax$$

The cell whose lower corner is at (i, j, k) and whose upper corner is at $(i+1, j+1, k+1)$ has nodes:

1. (i, j, k)
2. $(i, j, k+1)$
3. $(i+1, j, k)$
4. $(i+1, j, k+1)$
5. $(i, j+1, k)$
6. $(i, j+1, k+1)$
7. $(i+1, j+1, k)$
8. $(i+1, j+1, k+1)$

The call we use here creates an 8-node hexahedron (6 faces):

```
call ftn_create_hex_ele (wall_info, node_ids, istat)
```

For a 4-node tetrahedron (4 faces) use:

```
call ftn_create_tet_ele (wall_info, node_ids, istat)
```

For a 5-node pyramid (5 faces) use:

```
call ftn_create_pyra_ele (wall_info, node_ids, istat)
```

For a 6-node prism (also known as a wedge with 5 faces) use:

```
call ftn_create_prism_ele (wall_info, node_ids, istat)
```

For an arbitrary element (up to 256 faces) use:

```
call ftn_create_arb_poly_ele (wall_info, node_ids, istat)
```

Some additional comments concerning arbitrary elements are provided below:

```
c          arbitrary polyhedron (maximum of 256 faces; each face has
```

```

c                                     a maximum of 256 nodes):
c      integer istat, num_faces, wall_info(256)
c      integer num_verts_per_face(256)
c      integer face_verts(256*256)
c      call ftn_create_arb_poly_ele (num_faces, wall_info,
c                                   num_verts_per_face, face_verts, istat)
c
c      The face_verts array contains the node_ids that define
c      each face.  The faces follow each inside the face_verts
c      array.  For example, if face#1 has 4 nodes and face#2
c      has 3 nodes, then:
c          num_verts_per_face(1) = 4
c          face_verts(1) = face#1 node#1
c          face_verts(2) = face#1 node#2
c          face_verts(3) = face#1 node#3
c          face_verts(4) = face#1 node#4
c          num_verts_per_face(2) = 3
c          face_verts(5) = face#2 node#1
c          face_verts(6) = face#2 node#2
c          face_verts(7) = face#2 node#3

```

If `istat` is not zero, then the subroutine failed.

For C programmers, the corresponding calls are:

```

int istat, node_ids[8], wall_info[6];
istat = create_hex_ele (wall_info, node_ids);

int istat, node_ids[4], wall_info[4];
istat = create_tet_ele (wall_info, node_ids);

int istat, node_ids[5], wall_info[5];
istat = create_pyra_ele (wall_info, node_ids);

int istat, node_ids[6], wall_info[5];
istat = create_prism_ele (wall_info, node_ids);

int istat, node_ids[256*256], wall_info[256];
int num_faces, num_verts_per_face[256], face_verts[256*256];
istat = create_arb_poly_ele (num_faces, wall_info,
                             num_verts_per_face, face_verts);

node_ids(1) = i + (j-1)*imax + (k-1)*imax*jmax
node_ids(2) = node_ids(1) + imax*jmax
node_ids(3) = node_ids(1) + 1
node_ids(4) = node_ids(1) + 1 + imax*jmax

```

```

node_ids(5) = node_ids(1) + imax
node_ids(6) = node_ids(1) + imax + imax*jmax
node_ids(7) = node_ids(1) + 1 + imax
node_ids(8) = node_ids(1) + 1 + imax + imax*jmax

```

The `wall_info` array must be filled with a wall value for each face of this cell. The only supported values are `A_WALL` and `NOT_A_WALL`, defined as parameter constants above. Specifying `A_WALL` marks that element face as a wall for the purpose of streamline calculation. This tells **FieldView** that streamlines should not pass through this element face. It has no other effect. Marking an element face with `A_WALL` does not automatically create a wall boundary face. All boundary faces (walls, inlets, outlets, and others) must be created by calling `ftn_create_boundary_face` (Fortran) or `create_boundary_face` (for C programmers) as shown below.

To find streamline walls in a PLOT3D grid with IBlanks, we should examine the faces of this cell, looking for faces whose 4 corners all have IBlank values of 2. Each such face would be marked with a `wall_info` value of `A_WALL`. Instead of reading the IBlanks from the grid file, in this example we will simply mark all faces with `I=1` or `I=imax` as walls. With the node numbering given above and the hex face numbering used in **FieldView**, the "I" faces of a grid cell are face 5 (hex vertices 1,5,6,2) and face 6 (hex vertices 3,4,8,7).

```

wall_info(1) = NOT_A_WALL
wall_info(2) = NOT_A_WALL
wall_info(3) = NOT_A_WALL
wall_info(4) = NOT_A_WALL
if (i .eq. 1) then
wall_info(5) = A_WALL
else
    wall_info(5) = NOT_A_WALL
endif
if (i .eq. imax-1) then
    wall_info(6) = A_WALL
else
    wall_info(6) = NOT_A_WALL
endif
call ftn_create_hex_ele (wall_info, node_ids, istat)

if (istat .ne. 0) go to 900
100 continue

```

Warning: In Fortran, if `istat` is not zero, the element will not be created. In C, if `istat` is zero, the element will not be created (the opposite of Fortran). Unlike other toolkit functions, the element creation functions are not consistent between Fortran and C.

Creating Boundary Faces

Boundary faces are created using the function:

```
stat=ftn_create_boundary_face(itype,num_nodes, node_ids, normals_flag)
```

where:

`itype` = an integer between 1 and the number of boundary types, specifying which boundary type includes this face

`num_nodes` = 3 (triangle) or 4 (quadrilateral)

`node_ids` = an array of 3 or 4 node numbers defining this face.

If the face has 4 nodes, they must be specified in clockwise or counter-clockwise order. Do not follow a node with the node diagonally opposite - this will cause the face to be twisted.

`normals_flag` = 0 or not 0

A non-zero means that boundary faces of this type have consistent "clockness", so that correctly facing surface normals can be calculated using the right-hand rule for all faces of this type. A value of zero means that boundary faces of this type do not have consistent clockness. If boundary faces have consistent clockness, then boundary surface integrals involving surface normals will be available for "current surface" integration of boundary surfaces in **FieldView**.

When `ftn_create_boundary_face` returns, if `istat` is not zero, the boundary face was not created.

For C programmers, the corresponding function is:

```
int create_boundary_face(int itype,int num_nodes,node_ids, normals_flag)
```

In this example, we specified two boundary types in the query subroutine at the top of this sample source file:

```
boundary type 1 was 'I=1'
boundary type 2 was 'I=Imax'
```

Therefore, we will convert all `I=1` faces of the PLOT3D grid into boundary faces of type 1, and all `I=Imax` faces into boundary faces of type 2:

```
i = 1
do 200 k = 1, kmax - 1
do 200 j = 1, jmax - 1
```

As mentioned earlier, the position of node (i, j, k) in the PLOT3D grid is:

$$i + (j-1)*imax + (k-1)*imax*jmax$$

The face whose lower corner is at (i, j, k) and whose upper corner is at $(i, j+1, k+1)$ has nodes:

```
(i, j, k)
(i, j+1, k)
(i, j+1, k+1)
(i, j, k+1)
```

Therefore:

```
node_ids(1) = i + (j-1)*imax + (k-1)*imax*jmax
node_ids(2) = node_ids(1) + imax
node_ids(3) = node_ids(1) + imax + imax*jmax
node_ids(4) = node_ids(1) + imax*jmax
```

In this example, we don't care about making sure that all boundary faces have consistent clockness, therefore:

```
normals_flag = 0
```

I=1 nodes belong to boundary type 1, therefore:

```
istat = ftn_create_boundary_face (1, 4, node_ids, normals_flag)
```

Warning: In Fortran, if `istat` is not zero, the element will not be created. In C, if `istat` is zero, the element will not be created (the opposite of Fortran). Unlike other toolkit functions, the element creation functions are not consistent between Fortran and C.

```
if (istat .ne. 0) go to 900
```

```
200 continue
```

```
i = imax
do 300 k = 1, kmax - 1
do 300 j = 1, jmax - 1
    node_ids(1) = i + (j-1)*imax + (k-1)*imax*jmax
    node_ids(2) = node_ids(1) + imax
    node_ids(3) = node_ids(1) + imax + imax*jmax
    node_ids(4) = node_ids(1) + imax*jmax
    normals_flag = 0
```

I=1 nodes belong to boundary type 2, therefore:

```
istat = ftn_create_boundary_face (2, 4, node_ids, normals_flag)
```

Warning: In Fortran, if `istat` is not zero, the element will not be created. In C, if `istat` is zero, the element will not be created (the opposite of Fortran). Unlike other toolkit functions, the element creation functions are not consistent between Fortran and C.

```

    if (istat .ne. 0) go to 900
300 continue

```



Note: We created all boundary faces of type 1, and then all boundary faces of type 2. This is somewhat more efficient in **FieldView** than creating one boundary face of type 1, followed by a boundary face of type 2, then a face of type 1, and so on. If you have large numbers of boundary faces, then making sure boundary faces of the same type are grouped together can make a difference in the amount of memory needed, and in the speed of processing the boundary faces.

```

900 continue
    if (istat .ne. 0) then
        iret = 1
    else
        iret = 0
    endif

    return
end

```

Example of Unstructured Results File Data Reader

The subroutine for obtaining information on the results file is:

```

subroutine user_query_unstruct_results (fname, lenf, iunit,
+      max_grids, max_vars, num_grids, num_nodes, num_vars,
+      var_names, iret)

```

Variable Definitions

Input:

```

fname      = file name
lenf       = length of file name
iunit      = "unit number" you should use when opening the file
              (FieldView will take care of closing the file)
max_grids  = maximum number of grids allowed
max_vars   = maximum number of result variables allowed per grid point

```

Output:

```

num_grids  = number of grids that will be read from data file
num_nodes  (array of node counts for all grids that will be read)
num_vars   = number of result variables per grid point.
              Note: a vector variable counts as 3 results!
var_names  = array of variable names in the same format as a FieldView function name file.

```

For example, if the results file is like a PLOT3D Q file, it contains 5 variables per grid point with the following names:

```
density
u-momentum; momentum
v-momentum
w-momentum
stagnation energy

iret      = 0 (success)
          = 1 (failure)
```

This defines 5 scalar functions and 1 vector function (`momentum`) whose components are scalar variable 2 together with the next two scalar variables. See **Appendix C** for more information on the Function Name File format.

Arguments and Variables for Subroutine

The following line causes the Fortran compiler to complain about any variables that are not explicitly declared. This is useful for catching misspellings.

```
implicit none
```

Arguments for the subroutine:

```
character*(*) fname
integer lenf, iunit, max_grids, max_vars, num_grids, num_vars
integer num_nodes(max_grids), iret
```

You *must* declare "var_names" as an array of `character*80`, as in the following line:

```
character*80 var_names(max_vars)
```

Local variables in this example:

```
integer istat, j
integer nvar_grid
integer BYTES_PER_INTEGER
parameter (BYTES_PER_INTEGER = 4)

integer MAX_GRIDS_IN_FILE
parameter (MAX_GRIDS_IN_FILE = 100)
integer idims(MAX_GRIDS_IN_FILE)
integer jdims(MAX_GRIDS_IN_FILE)
integer kdims(MAX_GRIDS_IN_FILE)
common /user_grid_dims/ idims, jdims, kdims
```



Note on Common Block Names: It is strongly recommended that you begin any common block names with the word "user". Any other names may conflict with global names within **FieldView**, and cause program crashes. The following is an example of a safe common block name: `common /user1/ mydata, moredata integer mydata, moredata`

Opening the File

FieldView provides "open_binary" for opening a binary file (usually written by a C program) in a Fortran subroutine:

```
call open_binary (iunit, fname(1:lenf), istat)

if (istat .ne. 0) then
    iret = 1
    go to 900
endif
```

This is the subroutine for formatted data:

```
open (unit=iunit, file=fname(1:lenf), status='OLD',
+     form='FORMATTED', iostat=istat)
```

This is the subroutine for unformatted data:

```
open (unit=iunit, file=fname(1:lenf), status='OLD',
+     form='UNFORMATTED', iostat=istat)
```

Determine Number of Grids

The following example is for a file in single-grid PLOT3D format:

```
num_grids = 1

if (num_grids .gt. max_grids) then
    iret = 1
    go to 900
endif
```

Note: You should handle the problem of too many grids using the same approach you chose when reading the grid file.

Determine the Number of Nodes and the Number of Variables at Each Grid Point

Remember that vector quantities count as three variables. Note that **FieldView** requires all points on all grids to have the same variables defined.

The following example is for a binary PLOT3D file:

```

do 100 j = 1, num_grids
  call read_binary (iunit, BYTES_PER_INTEGER, idims(j), istat)
  if (istat.ne.0) go to 101
  call read_binary (iunit, BYTES_PER_INTEGER, jdims(j), istat)
  if (istat.ne.0) go to 101
  call read_binary (iunit, BYTES_PER_INTEGER, kdims(j), istat)
  if (istat.ne.0) go to 101

```

For a Function file use:

```

call read_binary (iunit, BYTES_PER_INTEGER, nvar_grid, istat)
if (istat.ne.0) go to 101

```

For the Q file use:

```

      nvar_grid = 5
      if (j. eq. 1) then
        num_vars = nvar_grid
      else if (num_vars .ne. nvar_grid) then
        iret = 1
        go to 900
      endif
100 continue
101 continue

do 175 j = 1, num_grids
  num_nodes(j) = idims(j) * jdims(j) * kdims(j)
175 continue

```

Note: All grids must have same number of variables.

For formatted data use this read statement (For a Q file, omit `nvars(j)` from the read and set it to 5):

```

read (iunit, *, iostat=istat)
+   dims(j), jdims(j), kdims(j), nvars(j), j = 1, num_grids)

```

Fill in the variable names in **FieldView** Function Name File format (see **Appendix C** for more information on this format). The following names are correct for a PLOT3D Q file. You should change these to match the number and names of the variables in your results file.

```

var_names(1) = 'density'
var_names(2) = 'u-momentum; momentum'
var_names(3) = 'v-momentum'
var_names(4) = 'w-momentum'
var_names(5) = 'stagnation energy'

```

```

    if (istat .ne. 0) then
        iret = 1
        go to 900
    endif

    iret = 0

900 continue
    return
end

```

Alternatively, the following will identify all variables as scalars with default names:

```

do 200 j = 1, num_vars
    var_names(j) = 'V' // char(ichar('0') + j)
200 continue

```

Reading the Results File

Use this subroutine to read the results:

```

subroutine user_read_unstruct_results (iunit, igrd, nodecnt,
+      num_vars, xyz, vars, iret)

```

Variable Definitions

Input:

iunit = "unit number" that was used when opening the file
igrd = which grid to read this time
nodecnt = number of nodes in this grid (as returned by query routine)
num_vars = number of result variables per grid point
xyz = array of XYZ coordinates of the points in this grid

Output:

vars (array of result values for this grid)
iret = 0 (success)
 = 1 (failure)



Note on *igrd*: This subroutine will be called once for each grid returned by the query subroutine. The first call to this routine will have *igrd*=1, the next *igrd*=2, and so on until *igrd*=*num_grids*.

Note on *num_vars*: A vector variable counts as 3 results!

Arguments for this Subroutine

The following line causes the Fortran compiler to complain about any variables that are not explicitly declared. This is useful for catching misspellings.

```
implicit none
```

```
integer iunit, igrd, nodecnt, num_vars
```

Note: The following arrays *must* be dimensioned as shown here:

```
real xyz (nodecnt, 3)
real vars (nodecnt, num_vars)
integer iret
```

Functions used in this example:

```
integer ftn_init_bndry_results
integer ftn_store_bndry_result
```

Local variables in this example:

```
integer istat, nwords
real qstuff(4)
integer BYTES_PER_INTEGER, BYTES_PER_REAL
parameter (BYTES_PER_INTEGER = 4)
parameter (BYTES_PER_REAL = 4)
integer num_bvars, results_flags(2), num_faces_with_results
character*80 bvar_names(2)
integer i, j, k, imax, jmax, kmax
integer MAX_GRIDS_IN_FILE
parameter (MAX_GRIDS_IN_FILE = 100)
integer idims(MAX_GRIDS_IN_FILE)
integer jdims(MAX_GRIDS_IN_FILE)
integer kdims(MAX_GRIDS_IN_FILE)
common /user_grid_dims/ idims, jdims, kdims

call read_binary (iunit, 4 * BYTES_PER_REAL, qstuff, istat)

nwords = num_vars * nodecnt
call read_binary (iunit, nwords * BYTES_PER_REAL, vars, istat)
```

Boundary Results

If you also have boundary results (quantities that are defined only on boundaries, such as wall quantities), then you need to store these in a different way. In **FieldView**, boundary results are assumed to be face-based: scalar or vector values assigned to the whole boundary face, rather than the boundary nodes (vertices). Boundary results are only supported if you created the boundary faces with `ftn_create_boundary_face` (`create_boundary_face` for C programmers). See the sample source file `user_unstruct_grid.f` for details.

In order to store boundary face results, you must first call `ftn_init_bndry_results` (`init_bndry_results` for C programmers) to tell **FieldView** that you have boundary results for this grid.

`ftn_init_bndry_results` is called as follows:

```
integer ftn_init_bndry_results
integer num_bvars, results_flags(num_face_types)
character*80 bvar_names(num_bvars)
integer num_faces_with_results
istat = ftn_init_bndry_results (num_bvars, results_flags,
+      bvar_names, num_faces_with_results)
```

Where the input arguments are:

`num_bvars` = the number of boundary variables per face

`results_flags` = an array of integers, one per boundary type

Where nonzero means that boundary results are expected for this boundary type. The number of boundary types is the value of "`num_face_types`" returned by your grid query subroutine.

`bvar_names` = array of boundary variables names.

This has the same format as the "`var_names`" array returned by your results query subroutine.

The output argument is:

`num_faces_with_results` = number of boundary faces in this grid that have boundary results

This is based on the `results_flags` array and the number of faces of each type in this grid. This may help you check your boundary face loops when storing boundary results.

When `ftn_init_bndry_results` returns its results and if `istat` is not zero, then the function failed. You will not be able to store boundary results. If you have boundary results on more than one grid in this results file, each grid must pass the same arguments to `ftn_init_bndry_results`.

For C programmers, the corresponding function is:

```
int init_bndry_results (int num_bvars, int results_flags,
char bvar_names, int *num_faces_with_results);
```

In this example, we specified two boundary types in the sample source file `user_unstruct_grid.f`, called '`I=1`' and '`I=Imax`'. We will suppose that both of these boundaries have boundary results and two boundary variables, are called '`Wall Temperature`' and '`Heat Flux`':

```
num_bvars = 2
```

```

results_flags(1) = 1
results_flags(2) = 1
bvar_names(1) = 'Wall Temperature'
bvar_names(2) = 'Heat Flux'
istat = ftn_init_bndry_results (num_bvars, results_flags,
+      bvar_names, num_faces_with_results)
if (istat .ne. 0) go to 900

```

PLOT3D Q files do not have boundary results, so for this example we will suppose that the $I=1$ faces have values of zero for wall temperature and for heat flux, while the $I=I_{\max}$ faces have values of 1 for wall temperature and 2 for heat flux. First, we store all values for wall temperature.

Loop through all $I=1$ faces in the PLOT3D grid, in the same order that we called `ftn_create_boundary_face` in `user_unstruct_grid.f`:

```

imax = idims(igrid)
jmax = jdims(igrid)
kmax = kdims(igrid)
i = 1
do 200 k = 1, kmax - 1
do 200 j = 1, jmax - 1
    istat = ftn_store_bndry_result (0.0)

```

Warning: In Fortran, if `istat` is not zero, the element will not be created. In C, if `istat` is zero, the element will not be created (the opposite of Fortran). Unlike other toolkit functions, the element creation functions are not consistent between Fortran and C.

```

    if (istat .ne. 0) go to 900
200 continue

```

Loop through all $I=I_{\max}$ faces in the PLOT3D grid, in the same order that we called `ftn_create_boundary_face` in `user_unstruct_grid.f`:

```

i = imax
do 300 k = 1, kmax - 1
do 300 j = 1, jmax - 1
    istat = ftn_store_bndry_result (1.0)

```

Warning: In Fortran, if `istat` is not zero, the element will not be created. In C, if `istat` is zero, the element will not be created (the opposite of Fortran). Unlike other toolkit functions, the element creation functions are not consistent between Fortran and C.

```

    if (istat .ne. 0) go to 900
300 continue

```

Now, loop again, this time for heat flux:

```

i = 1
do 400 k = 1, kmax - 1
do 400 j = 1, jmax - 1
    istat = ftn_store_bndry_result (0.0)

```

Warning: In Fortran, if `istat` is *not* zero, the element will not be created. In C, if `istat` *is* zero, the element will not be created (the opposite of Fortran). Unlike other toolkit functions, the element creation functions are not consistent between Fortran and C.

```

    if (istat .ne. 0) go to 900
400 continue

i = imax
do 500 k = 1, kmax - 1
do 500 j = 1, jmax - 1
    istat = ftn_store_bndry_result (2.0)
    if (istat .ne. 0) go to 900
500 continue

900 continue
    if (istat .ne. 0) then
        iret = 1
    else
        iret = 0
    endif

    return
end

```

Transient User-Defined Reader

FieldView supports two ways of reading transient data. The first way is to have the user pick a file representing a single time step. The file name must have a time step number embedded in it using a naming convention described in [Transient Data](#).

In this case, selecting the data file automatically selects the time step. However, to inform **FieldView** that each time step is in a separate file, the following must be inserted in the data reader's query function:

For FORTRAN:

```
call ftn_allow_timestep_per_file
```

For C:

```
allow_timestep_per_file();
```

Do not use the time step selection functions described below.

The second way of handling transient data is to have the user pick a file or database and have the user-defined reader construct a list of time steps found in that file or database. This list is passed to **FieldView**, which asks the user to choose a time step. To use the second form of time step selection:

Using C

```
float timestep [max-number-of-steps][2], *time
int nstep, *step, *iret
time_step_get_value (timestep, nstep, step, time, iret);
```

Using FORTRAN

```
real timestep (2,max-number-of-steps), time
integer nstep, step, iret
call ftn_timestep_get_value (timestep, nstep, step, time, iret)
```

Variable Definitions

Input:

timestep = an array of floating point step number/solution time value pairs
nstep = number of time steps

Output:

step = the step number returned
time = the solution time value returned
iret = return code

This function should be called from the read phase before reading the first grid. If there are multiple grids, this function should only be called for the first grid after the query phase. If the return code `iret` is not zero, then either the user cancelled the data read or there was an error. If there is a non-zero return code, exit from the reader immediately and pass the return code back to **FieldView**.

Examples for the two supported languages, C and FORTRAN follow. The lines in the sample code which contain an ellipsis (...) are meant to stand as additional lines that your actual code may contain. Only the pertinent coding for the transient recognition is included.

C Example:

```
--- Declarations within the scope of both the query and read functions. ---
...
/* Maximum number of time steps for this example is 5000. */
#define MAX_TIMES 5000

static float times[MAX_TIMES][2];
static int ntimes = 0;

--- In the user defined reader query function ---
```

```

...
    int istep;
    float time;
...
    ntimes = 0;

    /* Loop to read step number/solution time from the solver file
     * incrementing ntimes for each pair, and assigning values
     * in the times array.
     */
    do for as many as required
    {
        /* Read the istep and time value from the file. */
        ...

        times[ntimes][0] = (float)istep;
        times[ntimes][1] = time;
    }
...

--- In the user defined reader read function, typically the first thing ---
...
    int istep;
    float time;
...
    *iret = 0;

    /* Select the desired timestep.
     * Do this once, for the first grid after the query phase.
     * If ntimes is zero this could be steady state.
     */
    if (ntimes > 1) {
        time_step_get_value(times, ntimes, &istep, &time, iret)
        if (iret != 0) {
            goto 999
        }

        if ( (istep == 0) && (ntimes > 0) ) {
            printf("File does not specify a time step. Using first as
default.\n");
            istep = times[0][0];
            time  = times[0][1];
        }
    }

    /* Continue reading the solution file. The step number

```

```
* istep has been selected.
*/
```

```
...
```

FORTTRAN Example:

```
--- In the user defined reader query subroutine ---
```

```
...
```

```
C      Maximum number of time steps for this example is 5000
      common /usertime/times, ntimes
      real*4 times(2,5000)
      integer ntimes
      integer istep
      real*4 time
```

```
...
```

```
      ntimes=0
```

```
...
```

```
C      Loop to read step number/solution time from the solver file
C      incrementing ntimes for each pair, and assigning values
C      in the times array.
      do for as many as required
          ntimes=ntimes+1
C      Error if too many time steps
          if (ntimes.gt.5000) goto 705
```

```
C      Read the istep and time value from the file
```

```
...
```

```
          times(1,ntimes)=istep
          times(2,ntimes)=time
      enddo
```

```
--- In the user defined reader read subroutine, typically first thing ---
```

```
...
```

```
C      Maximum number of time steps for this example is 5000
      common /usertime/times, ntimes
      real*4 times(2,5000)
      integer ntimes
      integer istep
      real*4 time
```

```
      iret = 0
```

```
C      Select the desired timestep.
C      Do this once, for the first grid after the query phase.
C      If ntimes is zero this could be steady state.
      if (ntimes.gt.1) then
```

```

call ftn_tstep_get_value(times, ntimes, istep, time, iret)
if (iret .ne. 0) then
    goto 999
endif

```

C

```

if (istep.eq.0.and.ntimes.gt.0) then
    print *, 'My Reader warning: File does not specify a time step.'
    print *, 'Using the first time step as the default.'
    istep=times(1,1)
    time=times(2,1)
endif
endif

```

C Continue reading the solution file. The step number
C istep has been selected.

Support for Cartesian Grids

To enable optimization for Cartesian grids:

Using C:

```
void set_cartesian_grid_flag (int grid, int flag);
```

where:

grid=0 for the first grid in the data file being read, and
grid=1 for the second grid, etc.

and:

"flag" is any nonzero value if the grid is Cartesian (rectangular).
flag=0 if the grid is not Cartesian

Using Fortran:

```
call ftn_set_cartesian_grid_flag (igrid, iflag)
```

where:

igrid=1 for the first grid in the data file being read, and
igrid=2 for the second grid, etc.

and:

"iflag" is any nonzero value if the grid is Cartesian (rectangular).
iflag=0 if the grid is not Cartesian

Only structured grids can be Cartesian. If you set the Cartesian grid flag for an unstructured grid, **FieldView** will issue a warning and remove the flag.

Grids flagged as Cartesian skip all of the grid preprocessing (for point probe and coord surfaces), so they read in faster and use less memory. Probing and coord surfaces are much faster for Cartesian grids. Other things that use probing, such as using a Cartesian dataset as the "results target" for dataset sampling, will also be much faster.

If all grids are flagged as Cartesian, **DataGuide™** is disabled.

Using User-Defined Plugins with a FieldView Server

In the shared library approach used by **FieldView**, there is no need to build separate client and server shared libraries. The same shared library created for the client can be used for the server.

If you've built a shared library plugin for the client, and copied it (as instructed) into the `bin/plugins` directory of your **FieldView** installation, then the local server will automatically use the plugin you created. If you are not using any servers other than the local server, you can skip to step 4 of this section.

1. Installing the Server Toolkit for Your Server

If you chose to install the directory of compressed servers during the **FieldView** installation, you will find a subdirectory in your **FieldView** installation called `servers`. If you did not choose to install these, you will find this same directory on the **FieldView** DVD in the `unix/servers` directory.

In the `servers` directory, locate the appropriate compressed file for your platform. Unpack the file as follows:

```
zcat fvsvr_<platform>_tar.Z | tar xvf -
```

This will unpack the server for the specified platform, as well as a directory called `user`, which contains the files needed for building a server with user-defined routines.

2. Edit `make_fvsvr` for Shared Library Toolkit

You should edit the file `make_fvsvr` in the `user` directory, so the Fortran and C compiler names and flags match those for your compilers.

3. Bind Your Routines into the Server

Data readers and functions can be written for a **FieldView** server in the same manner as they are written for **FieldView**. If you have already written these for a **FieldView** client, you should be able to use the same C and Fortran files in the server, except for any platform-specific changes in your data readers and functions.

Once you have added your changes into the `user` directory, build a server plugin by issuing the command:

```
make_fvsrv
```

Install the plugin as instructed by `make_fvsrv`.

4. Create or Edit a **FieldView** Server Config File for Your Server

Go to the `sconfig` subdirectory of your **FieldView** installation directory. Create a server config file for the new server in this directory. If you previously created a server config file for this server, you can optionally edit this file instead of creating a new one.

Make sure to update the `ServerProgram` line with the full path of the server executable.

See “Installing **FieldView** Servers” in the **Installation Guide** for more information on server config files.

For each user-defined data reader you have added to your server, you must add a corresponding line to the server config file. Each line must begin with `"UserDefinedReader:"`.

If your reader reads grid geometry and results from separate files, then add the word `"Separate"`. Otherwise, add the word `"Combined"` for readers which read all data from a single file.

If your reader is for structured grids, add the word `"Structured"`. Otherwise, add the word `"Unstructured"` for unstructured grid readers.

Finally, add the title exactly as it appears in your registration function.

For example, to add a reader called `"My Data Reader"` which reads structured data with separate grid and results files, add the following line:

```
UserDefinedReader: Separate Structured My Data Reader
```

Unlike user-defined *readers*, user-defined *functions* do not require additional entries in the server config file. For complete information on how to set up Server Configuration (`.srv`) files for FieldView, [Step 3 – Set up a Server Configuration File](#) in the Installation Guide.

If you use a **FieldView** client on the same platform as the server, then the `"local"` server has the best performance. You can edit the server config file `local.srv` to include any user-defined data readers you have added to the local server. Alternatively, you can create a new local server config file for your new server; it will run in local mode as long as the client and server are on the same machine and the `"ServerName:"` line is not present in the config file.

5. Use Your New Server from **FieldView**

When you run **FieldView**, you should see an entry for your new server config file on the "Choose Server..." sub-menu. When you choose the new or edited server config file, your user-defined readers and/or functions should be available.

Frequently Asked Questions

How can I configure my Plugin Reader to run on a different platform?

The Plugin Toolkit reader(s) you may obtain or create yourself may be executable on only specific operating system(s). As an example, you may want to access your reader built for Linux, while running a FieldView Windows Client. In order to do that, you will need to use a *Server Configuration* file (.srv) to provide the reader name to the FieldView Client program, and read your data via *client-server*. For instance, if you register a reader name *My Data Reader* in your plugin for a *Separate Structured* style data format, you'll need to place the following line in the file:

```
UserDefinedReader: Separate Structured My Data Reader
```

This will allow the FieldView Client to produce a *Data Input* option for *My Data Reader*, given of course that you have placed the plugin for that reader in the *plugins* sub-folder where the FieldView Server executable resides on that other OS. This is explained in steps 3 through 5 of the above section [Using User-Defined Plugins with a FieldView Server](#). For complete information on how to set up Server Configuration (.srv) files for FieldView, [Step 3 – Set up a Server Configuration File](#) in the Installation Guide.

How many plugins can be used?

There is no limit on the number of plugins.

*How does **FieldView** locate the plugin?*

If the environment variable `FV_PLUGINS` is set then the directory defined by that variable is searched for a file ending in `.so` (Unix/Linux) or `.dll` (Windows). The first file found is used. When the environment variable is set no other directory is searched. If the environment variable is not set the directory `$FV_HOME/plugins` is searched while trying to locate plugin (`.so` or `.dll`) components. Note that the server does not use the environment variable `FV_PLUGINS`.

*How does the **FieldView** Server locate the plugin?*

The server executable will look for a `/plugins` directory (in the servers local directory) in trying to locate plugin (`.so` or `.dll`) components.

Is a FORTRAN compiler required?

No. But three Fortran symbols must be stubbed with the following C code:

```
void FTNSYM(ftn_register_data_readers) {}  
void FTNSYM(ftn_register_functions) {}  
void FTNSYM(ftn_fv_close) {}
```

Note: The `FTNSYM` macro is defined in `toolkit.c`.

What changes are needed to accommodate different compilers?

For External names:

Different FORTRAN compilers may or may not append an underscore to external names. To resolve this, change the `FTNSYM` macro in `toolkit.c` to match your FORTRAN compiler.

On Fortran string lengths:

Different FORTRAN compilers may use different size integers to hold the length of a string. This length argument is seen as a separate argument when FORTRAN calls C. The length seen by C is defined by the `typedef` for `ftn_strlen_t` in `toolkit.c`. To resolve this, change the `typedef` to match your FORTRAN compiler.

Compiler name and options:

Compiler name and options may need to be changed in the file `make_fv` which creates `Make-file.fv`.

Linker name and options:

Linker name and options may need to be changed in the file `ld_fv` which creates `fv_toolkit.so`.

Writing and using Parallel User-Defined Data Readers

FieldView supports two kinds of parallelization for data readers. These are "grid-parallel" and "partitioned-file parallel" (PFPR). Both of these require that the **FieldView** user is licensed for parallel operation.

Grid-Parallel Data Readers

Grid parallelization is for data that is organized (in the data file, or by the data reader) into **FieldView** grids, all of which are stored together in a single grid file or database. These can be structured grids or unstructured grids.

In grid parallelization, **FieldView** assigns different subsets of the grids to different parallel processes (different worker server processes inside a **FieldView** parallel server). In this way, the work of reading and post-processing the data is distributed and load-balanced.

FieldView data readers are divided into a "query" phase (which returns certain summary information such as the number of nodes in each grid), and a "data read" phase which reads one grid at a time (the grid number is supplied to the data read phase). In order to perform grid parallelization, a data reader must be able to read a subset of the grids in a dataset. These grids will be in ascending order, but there will be gaps. For example, the data read phase may be asked to read grid 3 (skipping over grids 1 and 2), then grid 5, and then grid 9.

If your data reader supports reading selected grids like this, you can enable grid parallelization by setting the `FV_GRID_PARALLEL_READER` option as described in `register_data_readers.c` and `ftn_register_data_readers.f`.

If your data reader is slow to skip over grids (such as having to read significant portions of grid 4 in order to skip from grid 3 to grid 5), then you may not get any parallel speed-up during the grid read phase. However, you should still get parallel speed-up during many post-processing operations, such as creating surfaces.

The query phase is called on all parallel (worker) processes (although this may change in a future version of **FieldView**). Therefore, if your query phase is slow, you may not get any parallel speed-up during your data read. However, you should still get parallel speed-up during many post-processing operations.

Certain features of user-defined readers are not supported; see the section [Features Unsupported in Parallel Data Readers](#).

Partitioned-File Parallel Data Readers

Partitioned-file parallelization is for datasets that are split into "partition files", each of which contains a subset of the entire dataset. For example, a parallel solver may split the dataset into partitions, assign a partition to each sub-process in the solver, and then write each partition into a separate file.

In partitioned-file parallelization, **FieldView** assigns each partition to a different parallel process (a different worker process inside the **FieldView** parallel server). The assignment of partitions to server processes is controlled by a "layout" file, which is simply a text file that lists the partition files and which host machines should process each partition. The format of the layout file is described under [Description of Layout File Format](#) in [Chapter 1](#) of this **Reference Manual**.

Unlike grid-file parallelization, all user-defined data readers automatically support partitioned-file parallelization. There is no need to set any special data reader registration options. However, certain features of user-defined readers are not supported; see the section [Features Unsupported in Parallel Data Readers](#).

Each **FieldView** server worker process only sees the single partition assigned to that process, so it behaves like an ordinary non-parallel data reader. The extra work of splitting or merging the operations on the dataset is done automatically by **FieldView** using the information in the layout file.

There are restrictions on the partition files. All of the partition files in a single dataset must have the same variable names (including boundary variable names if these are present). However, the partition files can have different boundary types; the boundary types will be automatically merged by **FieldView**.

Partitioned-file parallel does not support grid subsetting by the user during the data read. If the reader has enabled this, it is forced off.

There are no special requirements for efficiency. However, if all of the partition files are located in the same filesystem, then parallel speed-up during data reads can be hurt by competition for access to the filesystem.

Features Unsupported in Parallel Data Readers

The following features are not supported for grid-parallel or partitioned-file parallel data readers.

The following functions cannot be called from inside a parallel data reader:

```
fetch_element
fetch_element_ex
ftn_fetch_element
ftn_fetch_element_ex
```

If they are called from a parallel data reader, they will return an error code (-1 for failure, instead of 0 for success). These functions can be called from inside user-defined functions (parallel or not), just not from parallel data readers. Grid numbers inside each parallel process are local to that process; they are not the same as the grid numbers seen in the **FieldView** user interface. The grid number passed as an input argument to the user-defined functions is this kind of localized grid number. However, you can pass this localized grid number to the `fetch_element` family of functions, and they will return correct values for the grid. Be careful about using this grid number for anything except calling the `fetch_element` family.

Temporary region files created by the data reader are not supported. Therefore, if a parallel data reader calls:

```
open_tmp_fvreg
ftn_open_tmp_fvreg
```

then these functions will return an error code (-1 for failure, instead of 0 for success).

Parallel data readers, including the PLOT3D and **FieldView** Unstructured readers provided with **FieldView**, do not support the following **FieldView** features:

DataGuide™ (supported for partitioned-file parallel, but not for grid-parallel)

Dataset Sampling

Create Wall Boundaries

Create Exterior Boundaries

Appendix A Built-In Functions

Geometric Functions

```
X
Y
Z
Rcyl: (X^2+Y^2) ^ .5
Theta: atan(Y/X)
Rsphere: (X^2+Y^2+Z^2) ^ .5
Phi: acos(z/Rsphere)
(X^2+Z^2) ^ .5
atan(Z/X)
(Y^2+Z^2) ^ .5
atan(Y/Z)
I
J
K
IBLANK
```

Scalar Functions Available with PLOT3D Q Files



Note: All functions defined using the PLOT3D equations are appended with " [PLOT3D] "

Menu Name	Full Name
Density (Q1)	
Normalized density [PLOT3D]	
Stagnation density [PLOT3D]	
Norm. stag. density [PLOT3D]	Normalized stagnation density
Log (norm. density) [PLOT3D]	Log (normalized density)
Pressure [PLOT3D]	
Norm. pressure [PLOT3D]	Normalized temperature
Stagnation press. [PLOT3D]	Stagnation pressure
Norm. stag. press. [PLOT3D]	Normalized stagnation pressure
Cp [PLOT3D]	Pressure Coefficient
Stagnation Cp [PLOT3D]	Stagnation Pressure Coefficient
Pitot pressure [PLOT3D]	
Pitot press. ratio [PLOT3D]	Pitot pressure ratio
Dynamic pressure [PLOT3D]	
Log (norm. pressure) [PLOT3D]	Log (normal pressure)

Temperature [PLOT3D]
 Norm. temperature [PLOT3D]
 Stag. Temperature [PLOT3D]
 Norm. stag. temp. [PLOT3D]
 Log (norm. temp.) [PLOT3D]
 Enthalpy [PLOT3D]
 Norm. enthalpy [PLOT3D]
 Stag. Enthalpy [PLOT3D]

Norm. stag. enthalpy [PLOT3D]
 (Internal) energy [PLOT3D]
 Norm. int. energy [PLOT3D]
 Stagnation energy [PLOT3D]
 Norm. stag. Energy [PLOT3D]
 Kinetic energy [PLOT3D]
 Norm. kin. Energy [PLOT3D]
 u-velocity [PLOT3D]
 v-velocity [PLOT3D]
 w-velocity [PLOT3D]
 Velocity Magnitude [PLOT3D]
 Mach number [PLOT3D]
 Speed of sound [PLOT3D]
 Cross flow velocity [PLOT3D]
 Div. of velocity [PLOT3D]
 x-momentum (Q2)
 y-momentum (Q3)
 z-momentum (Q4)
 Stag. energy (Q5)
 Entropy [PLOT3D]
 Entropy measure s1 [PLOT3D]
 vorticity (x-dir) [PLOT3D]
 vorticity (y-dir) [PLOT3D]
 vorticity (z-dir) [PLOT3D]
 Vorticity Magnitude [PLOT3D]
 Swirl [PLOT3D]
 Vel. x Vort. mag. [PLOT3D]
 Helicity density [PLOT3D]
 Relative helicity [PLOT3D]
 Filter. rel. helicity [PLOT3D]
 Shock function [PLOT3D]
 Filter. shock func. [PLOT3D]
 Press. gradient mag. [PLOT3D]
 Dens. gradient mag. [PLOT3D]

Normalized temperature
 Stagnation temperature
 Normalized stagnation temperature
 Log (normalized temperature) Enthalpy

Normalized enthalpy
 Stagnation enthalpy

Normalized stagnation enthalpy

Normalized internal energy

Normalization stagnation energy

Normalized kinetic energy

Divergence of velocity

Stagnation energy (Q5) (per unit volume)

Velocity x Vorticity magnitude

Filtered relative helicity

Filtered shock function
 Pressure gradient magnitude
 Density gradient magnitude

Vector Functions Available with PLOT3D Q Files

Menu Name	Full Name
Velocity Vectors [PLOT3D]	
Vorticity Vectors [PLOT3D]	
Momentum Vectors [PLOT3D]	
Pert. vel. Vectors [PLOT3D]	Perturbation Velocity Vectors
Vel. x Vort. Vectors [PLOT3D]	Velocity x Vorticity Vectors
Press. grad. Vectors [PLOT3D]	Pressure gradient Vectors
Dens. grad. Vectors [PLOT3D]	Density gradient Vectors

Appendix B PLOT3D Formats

Introduction

PLOT3D supports four types of formats: Formatted, Unformatted, Double Precision (DP) Unformatted and Binary. Formatted files are simple ASCII text files that can be read and written using either FORTRAN or C. Unformatted files can only be read or written using FORTRAN, provided that the `OPEN` statement contains the argument `FORM=UNFORMATTED`. Unformatted files store floating point data in single precision (32 bit). DP Unformatted files are unformatted files that store floating point data in double precision (64 bit) and integer data in single precision. Binary files can only be written and read using C. SGI machines support writing and reading binary files in FORTRAN using the `FORM=SYSTEM` option on the open statement.

Formatted, Binary, Unformatted and DP Unformatted files generated on UNIX machines are readable on Windows and LINUX machines and visa-versa. Byte-swapping (when it is necessary) is done automatically.

For transient data, each time step is stored in a separate file. The file name must have a time step number embedded in it using a naming convention described in [Transient Data](#).

The following subroutines show how to write out PLOT3D unformatted files. To change these to write formatted files, change `(IUNIT)` to `(IUNIT, *)`.



Note: All Unformatted and Binary files must have floating point and integer values in single precision in order for **FieldView** to be able to correctly read in the files. DP Unformatted files must have floating point values stored in double precision and integer values stored in single precision in order for **FieldView** to be able to correctly read in the files.

Face Data and PLOT3D Format

FieldView supports face-based results on boundary surfaces of PLOT3D data. In order to provide face results for a PLOT3D dataset, *three* additional files will need to be created. One is a 2D Function File, which contains the face results for those boundary surfaces that have them. This file is described in the Function Files section of this appendix, below. In addition, a Function Name file which communicates the names of the face result variables to **FieldView** is needed, and described in the next appendix. The last required file is a special form of the Structured Boundary file (`*.fvbnd`) which communicates the boundary surface definitions, the surface normal direction, and whether there are face results for each of the boundary surfaces. The format that accommodates this is described in [Appendix H](#) of this **Reference Manual**.



Warning: When using `UNFORMATTED` statements, each line of information must be written with a single `WRITE` statement, as shown. That is, you cannot write out all `X` values with one `WRITE` statement, all `Y` values with a different `WRITE` statement, and all `Z` values with a third `WRITE` statement.

Grid XYZ Files

The grid file (XYZ file) defines the coordinates of the grid points in the structured mesh. The points are stored in I, J, K order with all of the X coordinates stored first, the Y coordinates second, and the Z coordinates last. If the file contains IBlank information, it is stored after the Z coordinates. See the section on IBlank Usage for additional details.

XYZ_File Single_Grid

```
WRITE(IUNIT) IDIM, JDIM, KDIM
WRITE(IUNIT) ((X(I, J, K), I=1, IDIM), J=1, JDIM), K=1, KDIM),
& ((Y(I, J, K), I=1, IDIM), J=1, JDIM), K=1, KDIM),
& ((Z(I, J, K), I=1, IDIM), J=1, JDIM), K=1, KDIM)
```

XYZ_File Single_Grid with IBlank

```
WRITE(IUNIT) IDIM, JDIM, KDIM
WRITE(IUNIT) ((X(I, J, K), I=1, IDIM), J=1, JDIM), K=1, KDIM),
& ((Y(I, J, K), I=1, IDIM), J=1, JDIM), K=1, KDIM),
& ((Z(I, J, K), I=1, IDIM), J=1, JDIM), K=1, KDIM),
& ((IBLANK(I, J, K), I=1, IDIM), J=1, JDIM), K=1, KDIM)
```

XYZ_File with Multiple_Grid

```
WRITE(IUNIT) NGRID
WRITE(IUNIT) (IDIM(IGRID), JDIM(IGRID), KDIM(IGRID), IGRID=1, NGRID)
DO 10 IGRID = 1, NGRID
  WRITE(IUNIT)
& ((X(I, J, K),
& I=1, IDIM(IGRID), J=1, JDIM(IGRID), K=1, KDIM(IGRID)),
& ((Y(I, J, K),
& I=1, IDIM(IGRID), J=1, JDIM(IGRID), K=1, KDIM(IGRID)),
& ((Z(I, J, K),
& I=1, IDIM(IGRID), J=1, JDIM(IGRID), K=1, KDIM(IGRID))
10 CONTINUE
```



Note: In the above WRITE statements for the grid information, "X", "Y" and "Z" stand for the particular arrays that hold the XYZ values for the Nth grid.

XYZ_File with Multiple_Grid and IBlank

```
WRITE(IUNIT) NGRID
WRITE(IUNIT) (IDIM(IGRID), JDIM(IGRID), KDIM(IGRID), IGRID=1, NGRID)
```

```

DO 10 IGRID = 1, NGRID
  WRITE (IUNIT)
&      ((X(I,J,K),
&      I=1, IDIM(IGRID)), J=1, JDIM(IGRID)), K=1, KDIM(IGRID)),
&      ((Y(I,J,K),
&      I=1, IDIM(IGRID)), J=1, JDIM(IGRID)), K=1, KDIM(IGRID)),
&      ((Z(I,J,K),
&      I=1, IDIM(IGRID)), J=1, JDIM(IGRID)), K=1, KDIM(IGRID)),
&      ((IBLANK(I,J,K),
&      I=1, IDIM(IGRID)), J=1, JDIM(IGRID)), K=1, KDIM(IGRID))
10 CONTINUE

```



Note: In the above `WRITE` statements for the grid information, "X", "Y" and "Z" stand for the particular arrays that hold the XYZ values for the Nth grid.

2D - XYZ_File Single_Grid

```

WRITE(IUNIT) IDIM, JDIM
WRITE(IUNIT) ((X(I, J), I=1, IDIM), J=1, JDIM),
&      ((Y(I, J), I=1, IDIM), J=1, JDIM)

```

2D - XYZ_File Single_Grid with IBlank

```

WRITE(IUNIT) IDIM, JDIM
WRITE(IUNIT) ((X(I, J), I=1, IDIM), J=1, JDIM),
&      ((Y(I, J), I=1, IDIM), J=1, JDIM),
&      ((IBLANK(I, J), I=1, IDIM), J=1, JDIM)

```

2D - XYZ_File Multiple_Grid

```

WRITE(IUNIT) NGRID
WRITE(IUNIT) (IDIM(IGRID), JDIM(IGRID) IGRID=1, NGRID)
DO 10 IGRID = 1, NGRID
  WRITE (IUNIT)
&      ((X(I, J), I=1, IDIM(IGRID)), J=1, JDIM(IGRID)),
&      ((Y(I, J), I=1, IDIM(IGRID)), J=1, JDIM(IGRID))
10 CONTINUE

```



Note: In the above `WRITE` statements for the grid information, "X", "Y" and "Z" stand for the particular arrays that hold the XYZ values for the Nth grid.

2D - XYZ_File with Multiple_Grid and IBlank

```

WRITE(IUNIT) NGRID
WRITE(IUNIT) (IDIM(IGRID), JDIM(IGRID), IGRID=1, NGRID)
DO 10 IGRID = 1, NGRID
    WRITE(IUNIT)
    &      ((X(I, J), I=1, IDIM(IGRID)), J=1, JDIM(IGRID)),
    &      ((Y(I, J), I=1, IDIM(IGRID)), J=1, JDIM(IGRID)),
    &      ((IBLANK(I, J), I=1, IDIM(IGRID)), J=1, JDIM(IGRID))
10 CONTINUE

```



Note: In the above `WRITE` statements for the grid information, "X", "Y" and "Z" stand for the particular arrays that hold the `XYZ` values for the N^{th} grid.

Note: All Unformatted and Binary files must have floating point and integer values in single precision in order for **FieldView** to be able to correctly read in the files. DP Unformatted files must have floating point values stored in double precision and integer values stored in single precision in order for **FieldView** to be able to correctly read in the files.

Solution Q Files

The PLOT3D Q files contain flow quantities for the grid points defined in `XYZ` files. In order to properly use the PLOT3D functions, the variables in the `Q` file must be properly normalized. The free-stream density is assumed to be 1. So, the `Density (Q1)` has been divided by the free-stream density. Similarly, the free-stream speed of sound set equal to 1. Thus, `Momentum (Q2, Q3, Q4)` is divided by free-stream density and free-stream speed of sound.

The free-stream velocity magnitude can be calculated from the free-stream Mach number, `FSMACH`. The direction of the free-stream velocity is computed from the angle of attack, `ALPHA` (in degrees). (The angle of attack is currently used only in the computation of the Perturbation Velocity vectors.)

The other two values, `RE` and `TIME`, are not currently used in the PLOT3D functions. The time value is used for transient data input (see [Chapter 14](#) of **Working with FieldView**).

In computing the PLOT3D functions, the fluid is assumed to be air, and behave as a perfect gas. The ratio of specific heats (γ) is assumed to be 1.4 and the gas constant (R) is assumed to be 1.

Values of the ratio of specific heats (γ) and the gas constant (R) can be changed using command line options when running **FieldView**. Refer to [Chapter 1](#) of this **Reference Manual** and [Chapter 1](#) of the **User's Guide** for more information about this feature.

The following subroutines show how to write out PLOT3D unformatted files. To change these to read in formatted files, change `(IUNIT)` to `(IUNIT, *)`.

Q_File

Freestream Mach number (`FSMACH`)

Angle of attack	(ALPHA)
Reynolds number	(RE)
Time	(TIME)

Q_File Single_Grid

```
WRITE (IUNIT) IDIM, JDIM, KDIM
WRITE (IUNIT) FSMACH, ALPHA, RE, TIME
WRITE (IUNIT) ( ( (Q(I, J, K, NX), I=1, IDIM), J=1, JDIM), K=1, KDIM),
&    NX=1, 5)
```

Q_File Multiple_Grid

```
WRITE (IUNIT) NGRID
WRITE (IUNIT) (IDIM(IGRID), JDIM(IGRID), KDIM(IGRID), IGRID=1, NGRID)
DO 10 IGRID = 1, NGRID
    WRITE (IUNIT) FSMACH, ALPHA, RE, TIME
    WRITE (IUNIT) ( ( (Q(I, J, K, NX),
&    I=1, IDIM(IGRID)), J=1, JDIM(IGRID)),
&    K=1, KDIM(IGRID)), NX=1, 5)
10 CONTINUE
```

2D - Q_File Single_Grid

```
WRITE (IUNIT) IDIM, JDIM
WRITE (IUNIT) FSMACH, ALPHA, RE, TIME
WRITE (IUNIT) ( ( (Q(I, J, NX), I=1, IDIM), J=1, JDIM), NX=1, 4)
```

2D - Q_File Multiple_Grid

```
WRITE (IUNIT) NGRID
WRITE (IUNIT) (IDIM(IGRID), JDIM(IGRID), IGRID=1, NGRID)
DO 10 IGRID = 1, NGRID
    WRITE (IUNIT) FSMACH, ALPHA, RE, TIME
    WRITE (IUNIT) ( ( (Q(I, J, K, NX),
&    I=1, IDIM(IGRID)), J=1, JDIM(IGRID)), NX=1, 4)
10 CONTINUE
```



Note: All Unformatted and Binary files must have floating point and integer values in single precision in order for **FieldView** to be able to correctly read in the files. DP Unformatted files must have floating point values stored in double precision and integer values stored in single precision in order for **FieldView** to be able to correctly read in the files.

Function Files

The Function Files contain a list of variables on the mesh defined in the grid file. These quantities are stored in **I**, **J**, **K** order with the names of the variables being defined in the Function Name File (see [Appendix C](#) of this **Reference Manual**). Function Files may be used instead of, or in conjunction with **Q** files.

The following subroutines show how to write out PLOT3D unformatted files. To change these to read in formatted files, change (IUNIT) to (IUNIT, *).

Function_File Single_Grid

```
WRITE (IUNIT) IDIM, JDIM, KDIM, NVAR
WRITE (IUNIT) ( ( (F(I, J, K, NX), I=1, IDIM),
&      J=1, JDIM), K=1, KDIM), NX=1, NVAR)
```

Function_File Multiple_Grid

```
WRITE (IUNIT) NGRID
WRITE (IUNIT) (IDIM(IGRID), JDIM(IGRID), KDIM(IGRID), NVAR(IGRID),
&      IGRID=1, NGRID)
DO 10 IGRID=1, NGRID
    WRITE (IUNIT) ( ( (F(I, J, K, NX),
&      I=1, IDIM(IGRID)), J=1, JDIM(IGRID)), K=1, KDIM(IGRID)),
&      NX=1, NVAR(IGRID))
10 CONTINUE
```

2D - Function_File Single_Grid

```
WRITE (IUNIT) IDIM, JDIM, NVAR
WRITE (IUNIT) ( ( (F(I, J, NX), I=1, IDIM), J=1, JDIM), NX=1, NVAR)
```

2D - Function_File Multiple_Grid

```
WRITE (IUNIT) NGRID
WRITE (IUNIT) (IDIM(IGRID), JDIM(IGRID), NVAR(IGRID), IGRID=1, NGRID)
DO 10 IGRID=1, NGRID
    WRITE (IUNIT) ( ( (F(I, J, NX),
&      I=1, IDIM(IGRID)), J=1, JDIM(IGRID)), NX=1, NVAR(IGRID))
10 CONTINUE
```



Note: For any vector field in the **Q** File, a dummy **W** component of the vector will automatically be created upon read in, with all of the values set to zero. However, vectors defined in the Function File must still contain 3 components.

2D - Function_File Multiple_Grid

```

WRITE (IUNIT) NGRID
WRITE (IUNIT) (IDIM(IGRID), JDIM(IGRID), NVAR(IGRID), IGRID=1, NGRID)
DO 10 IGRID=1, NGRID
    WRITE (IUNIT) ((F(I,J,NX),
&          I=1, IDIM(IGRID)), J=1, JDIM(IGRID)), NX=1, NVAR(IGRID))
10 CONTINUE

```

The file format *must* be multi-grid because each boundary ‘patch’ will need to have its own results (grid). A boundary surface may consist of faces from several different grids, each grid requiring at least one entry (and perhaps more) in the Structured Boundary File. An example of this is given in the Face Data section of [Appendix H](#) of this **Reference Manual**.

Face Data and Function Files

FieldView supports face-based results on boundary surfaces of PLOT3D data. In order to provide face results for a PLOT3D dataset, one of the additional files needed is a standard 2D Function File which contains the face results for those boundary surfaces that have them. That is, the face data file for a 3D dataset is a 2D, not a 3D, file. The Function file should have the same file name as the results file *plus* have an additional extension: *.fv_{surf}. Information about all of the additional files necessary in order to use face data is given in [Appendix H](#) of this **Reference Manual**.

IBlack Usage

IBlanking is a technique which permits definition of walls and holes within a mesh. It can also be used to allow integration (for streamline calculations) in models with multiple-grids.

The IBlank array, if present in the XYZ file, contains a single integer value per grid point. If one grid has IBlank values defined, then all grids in a multi-grid model must have IBlank values. Recognized values are:

- 0 = a point not in the computational domain (a hole), to be ignored in processing grids and results. This is typically used for non-fluid regions.
- 1 = an ordinary grid point.
- 2 = a wall point. Integration in the Streamlines panel is forced away from walls (grid cell faces which have IBLANK = 2 on all vertices), by limiting velocity and position very near the wall.

Negative = negative IBlank values are used by the integration procedure in the Streamlines panel as grid pointers to allow the integration to continue from one grid to another. If this value is not set, it is still possible to have the integration continue across grids (see the [Chapter 6](#) of **Working with FieldView** for more information).

In order to alert **FieldView** that you want to integrate through both grids, you must set the IBlanking to be negative in those common areas. The negative (-) sign indicates to **FieldView** that it should try to integrate to another grid. The number following the negative (-) sign tells the software which grid to integrate into.

If grid *M* and grid *N* have a joining boundary, you would set the IBlank of grid *M* to *-N* at the boundary. This tells grid *M* to integrate across boundaries into the *N* grid. In addition, you must also set

the IBlank of grid N to $-M$ at the boundary. This tells grid N to integrate across boundaries into the M grid.

PLOT3D

The Data Type of each input file must be specified. The files may either be standard PLOT3D Grid files (XYZ), Results files (Q) or Function Files (which allows for any number of input variables). Pressing one of these Data read buttons will bring up a file selector.

By pressing Merge Series, a series of 2D files (or 3D files with one of the dimensions equal to 1) may be read in as time steps of a solution. The files will be appended together in the K dimension (or in the dimension that is equal to 1, for 3D solutions).

When the data is read in, it may either Replace the data currently in memory or be Appended to the current data.

File Format is used to specify the format of the input data. The supported formats are: Formatted, FORTRAN Unformatted, Double Precision FORTRAN Unformatted and Binary.

These buttons indicate whether the input data contains Multiple Grids or IBlanks. The default is off for both options.

This area is used to specify whether the PLOT3D file to be read is in the 2D or 3D format.

The screenshot shows the 'Data Input' dialog box for PLOT3D. It includes sections for FORMAT (set to PLOT3D), INPUT MODE (Replace and Append), OPTIONS (Merge Series), FILE FORMAT (Formatted, Unformatted, DP Unformatted, Binary), DATA FORMAT (IBlanks, Multi-Grid), COORDS (2-D, 3-D), and GRID PROCESSING (Less, Balanced, More). Annotations with arrows point to various elements: 'FORMAT' points to the dropdown menu; 'When the data is read in...' points to the 'Replace' and 'Append' radio buttons; 'File Format is used to specify...' points to the 'Formatted', 'Unformatted', 'DP Unformatted', and 'Binary' radio buttons; 'These buttons indicate whether...' points to the 'IBlanks' and 'Multi-Grid' checkboxes; and 'This area is used to specify...' points to the '2-D' and '3-D' radio buttons under 'COORDS'.

Figure 141 PLOT3D Data Input Panel

PLOT3D Constants

The PLOT3D equations which are functions of the Q (solution) vector (such as Temperature [PLOT3D], Enthalpy [PLOT3D], etc.) are computed with the assumptions of γ (gamma) and R (gas constant) as constants equal to 1.4 and 1.0, respectively. For those users who have Q files resulting from solutions involving perfect gases (but not necessarily air), the internal value of these two constants can be changed so that the PLOT3D functions calculated will be correct. That is, if you have a solution where the gas is pure N_2 (with $\gamma = 1.2$), you can enter in this value for gamma and adjust the gas constant, R , appropriately so that the PLOT3D functions that are available in the Function Specification panel are valid.

Acceptable values of these constants are: $\gamma > 1$ and $R > 0.0$. In order to change the internal values of the gas constant (R) and gamma (γ), command line switches will have to be used. The command line switches are:

```
-gasconstant value
```

and

```
-gamma value
```

Please refer to [Chapter 1](#) of the **User's Guide** for more information about how to apply these values.



Note: These constants (γ and R), will be written into **DataGuide™** files. For more information on the impact of this on **DataGuide™** processing, see [Chapter 1](#) of the **User's Guide**.

Using the PLOT3D Data File Input Panel

What are the steps for reading in a PLOT3D file?

To read in a PLOT3D file, you must first read in an XYZ file (grid file) and then a results file (either Q or Function). Since the PLOT3D format does not contain any flags indicating the type of file, you must specify if the file is Formatted, Unformatted, Binary, or DP Unformatted, whether it is in 2D or 3D format, and whether it contains Multiple grids and IBlanks.

After the XYZ file is selected, you will be presented with the grid subset selection menu (see **Figure 142**). This menu gives you the option of selecting only certain grids to read in, or allows you to skip grids points on input. This second option will increase performance on smaller machines.

After selecting a Function file and pressing OK, you will be given the Function Name Input panel (**Figure 143**). This allows you to input a file containing the names of the variables in the Function file. If you do not have such a file, you can simply press the Use Defaults button. A detailed description of the name file is given in [Appendix C](#) of this **Reference Manual**.

What is the difference between XYZ , Q , and Function Files?

An XYZ file defines the grid itself. It contains all of the grid points and their associated x , y , and z coordinates. A Q file is a standard PLOT3D file that contains several constants and the 5 specially non-dimensional variables: density ($Q1$), 3 momentum components ($Q2$, $Q3$, $Q4$) and stagnation

energy (Q5). A function file is similar to a Q file but can contain as many variables as desired, with specific names.

Can I give specific names to my functions in a Function File?

The Function name file is used to give names to each of the variables found in the Function file. For more information of the Function Name File, see [Appendix C](#) of this **Reference Manual**.

Can I use a Q and Function File Together?

Both Q and Function files may be used together or independently.

Can I read in Q or Function Files in any order?

No. The only method which gives the desired result is to read Grid1, Results1, Grid2, Results2, etc. Any other ordering will not function.

What is the difference between Formatted, Unformatted and Binary files?

Formatted files are simple ASCII text files that can be read and written using either FORTRAN or C. Unformatted files can only be read or written using FORTRAN, provided that the OPEN statement contains the argument FORM=UNFORMATTED. Binary files can only be read using C (although some machines support reading binary files in FORTRAN using the FORM=BINARY option on the OPEN statement).

What are IBlanks?

For a complete description of IBlanks, please see the IBlank section earlier in this appendix.

How can I get a list of files with a certain extension (or prefix)?

Using the File filter you can get a list of files with a certain extension (or prefix). To do this, simply type in the part of the file that you would like, using a wildcard for the rest of the file name, and either perform a carriage return (hit the Enter key on keyboard), or press the filter button. For example, to find all files that end in .bin, you would type: *.bin.

What about 3D transient data?

The Merge Series option (see below) should not be used for true 3D transient data. Instead, you should simply select one of the files for input. If the proper naming conventions have been followed (see [Chapter 1](#) of this **Reference Manual**), **FieldView** will determine that a set of transient data has been selected, and give you the choice of reading in only one time step, or using the transient options. If you select the transient option, the Transient Data Controls panel will become active. For information on how to read in a set of transient PLOT3D data, see [Chapter 1](#) of this **Reference Manual**. For information on how to use the Transient Data Controls panel, see [Chapter 14](#) of **Working with FieldView**.

How can I read in a series of 2D time-dependent files?

A group of 2D files may be read in to **FieldView** in order to display a time series. These files must be either 3D (with one of the I, J, or K dimensions equal to 1), or 2D. The files must have the same prefix and suffix, with only the step number being changed. For example, if time step 1 is called test001.bin, all the other time steps must have a name of the form test###.bin, where "###" indicates the numbering of the files.

To read in the time series, you will need to turn on the Merge Series button, and then use the PLOT3D panel as described above. Next, select any one of the `XYZ` files from the list. You will then be presented with the Merge Series File Selection panel. This panel will list all files with the same prefix and suffix as defined above. You may then choose which of the files you wish to read in. You will be presented with the same panel upon reading in your solution file, and should make sure to select the same group of time steps as you did when reading the grid file. See [Chapter 1](#), and the end of this appendix of this **Reference Manual** for more information.

After responding to the Merge Series File Selection panel, you will be presented with the Grid Subset Selection panel and may proceed as usual.

How can I display my series once it has been read?

Upon reading in a group of files to show a series, **FieldView** will append all of these datasets together in memory. The data will be appended in the `K` dimension, for 2D datasets, or in the dimension that is equal to 1, for 3D datasets. For example, if you read in 10 time steps of a problem that is 20×40 , **FieldView** will set `I` = 20, `J` = 40 and `K` = 10. If, on the other hand, the 10 time steps were for a grid defined to be $30 \times 1 \times 40$, the `I`, `J` and `K` dimensions would be 30, 10, and 40 respectively.

To display the series, press the Create button on the Computational Surface panel. The default surface will be the dimension that was used for the time series. This would be a `K` surface for the first problem described above, and a `J` surface for the second. When you then press the Sweep button on the Computation Surface panel, the surface will be moved through the time steps. For example, in the first problem `K` = 1 would be time step 1, `K` = 2 would be time step 2, etc.

What if I have a single grid file but a different results file for each dataset?

FieldView supports the reading of multiple grid and solution files to show a series of steps. If you only have one grid file but multiple results files, **FieldView** will match up the single grid file with the set of time-dependent results files.

Possible Problems:

Because PLOT3D data files have no internal type tags and there are no naming conventions, the user must know what the format is for each dataset and whether or not it has Multi-Grids or IBlanks. See Data Input Constraints below.

Data Input Constraints:

`XYZ` data for a given grid must be read before results data for the same grid. You will get an error message if you try to read results data for a grid which has no `XYZ` data yet.

Each of the results files must have the same number of grids as the corresponding `XYZ` file, and the grids must be the same size in the results file and the `XYZ` file.

FieldView attempts to detect mismatches between the settings for File Format, Data Type, Data Format and the data in the file, as best it can given the limitations of the PLOT3D formats. Many such mismatches cannot be detected; it is the user's responsibility to know the correct settings for each file.

The Data Restart file will reproduce the input subsetting chosen by the user. However, Computational Surface, Iso-Surface or Streamlines Restart files that were created with a particular input subsetting may not be usable with a Data Restart file or data which has a different input subsetting. Old restarts may be unusable, but more recent restarts are more flexible and an attempt is made to create the surface in question. See the section [Restart Flexibility in Chapter 5](#) of this **Reference Manual**.

Grid File Input

Every time an XYZ file is read (whether in Replace or Append mode), the file header will be read. If no errors are detected, the Grid Subset Selection sub-panel will appear. This sub-panel can be used to reduce the amount of data to read by selecting from the total number of grids and the total number of points. This can be done to enhance performance or to focus the visualization to certain areas.

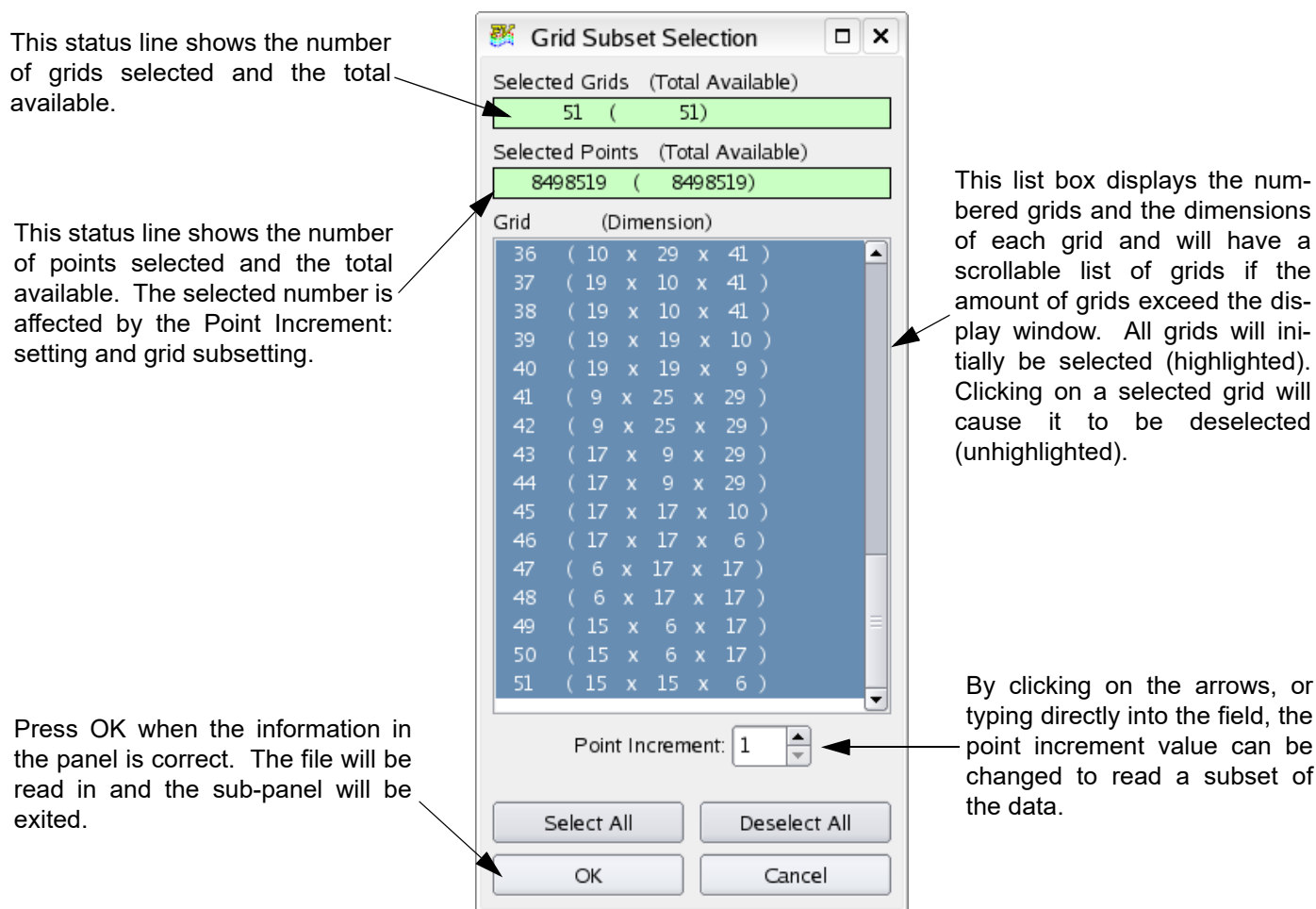


Figure 142 Grid File Input Panel

Using the Grid File Input Panel

How can I select the grids I want to read?

When a grid file has been selected, a list of all of the grids, and their dimensions, will appear in the grid area of this panel. All of the grids will be selected by default (highlighted). To turn off a grid for read in, simply click the mouse on the grid you do not want to read in. Once this decision has been made, press the OK button at the bottom of the panel.

How can I read in a subset of my data?

By changing the Point Increment, **FieldView** will read in a subset of the data. For example, setting the Point Increment to 2 will cause every other grid point in each direction to be read in. This will result in $1/8^{\text{th}}$ of the total data being read in.

*Why are the **FieldView** grid numbers different than my input data?*

If you use the Grid File Input panel to skip some of your grids, the grid will be re-numbered so that if grid 2 is skipped, then grid 3 becomes grid 2.

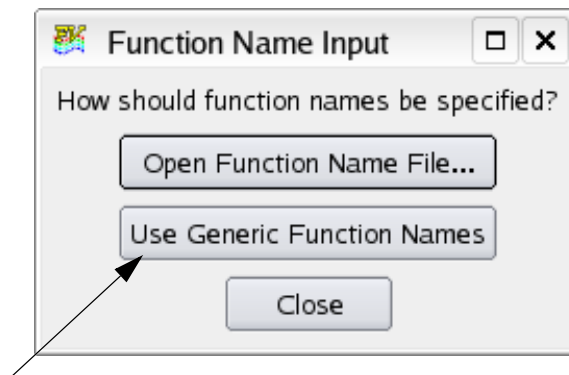
*Why are the **FieldView** point numbers different than my input data?*

If you use the Grid File Input panel to eliminate some of your grid points, IJK values will be re-numbered so that if Point Increment = 2, then $I = 5$ becomes $I = 3$. If Point Increment = 2, then only odd IJK values are read. This means if the I values in the XYZ file go from 1 to 40, then the original maximum value ($I = 40$) will not be read.

Function File and Function Name File Input

The **FieldView** Function File, as described earlier in this appendix, contains PLOT3D results. The names of the functions need to be communicated to **FieldView** through use of the Function Name file. When a Function File is selected for read-in, **FieldView** will prompt you to select a Function Name file. This file is used to define scalar and vector function names for the variables in the Function files. A Function Name file is required to have the extension `.nam`. See [Appendix C](#) of this **Reference Manual** for a description of the format. If you wish to read a Function file without supplying names for the variables, the system will supply default names if you select Use Defaults described below.

This sub-panel will appear every time a Function file is read.



By pressing Use Generic Function Names, the Function Name file will be ignored and the entry will be cleared. The Functions will be listed as the letter F followed by a number for as many functions in the Function File. A Function File with ten functions will be listed as F1 through F10. Note that no vectors will be created.

Figure 143 Function Name Input Panel

Function Name Mismatch

Function Name files that contain a different number of entries than the corresponding Function File can be used, but a Warning pop-up is first issued.

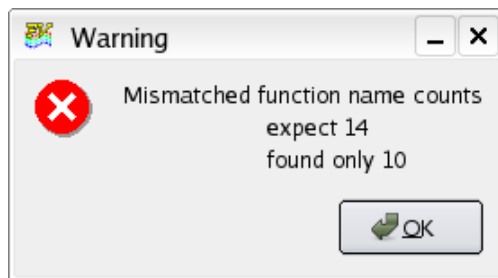
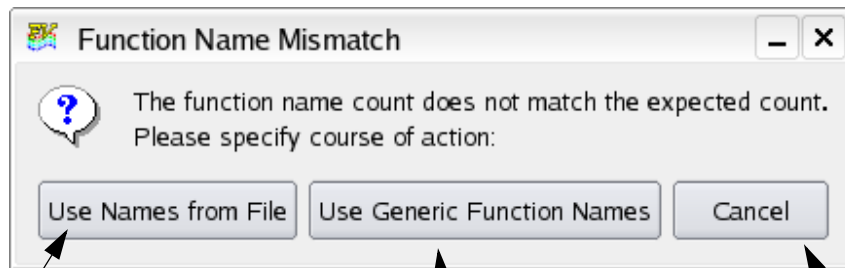


Figure 144 Function Name Warning

After OK is pressed, the mismatch panel is brought up giving you several options:



By pressing Use Names when too few names are found, the remaining functions will be named using the default names (the letter F followed by the line number). If too many names are found, the Name file will be used for the available functions and the remaining names will be discarded.

By pressing Use Defaults, the system will ignore the Function Name file and the Function Names list will be numbered as F1 through Fn, where n is the number of functions in the Function File. Default names do not include any vector function definitions.

Pressing Cancel will abort the operation. The Function Name file will not be read and the pop-up will be removed.

Figure 145 Function Name Mismatch Panel

Merge Series File Selection

The Merge Series is a special type of transient PLOT3D solution. Normally, transient data is handled by reading in one of the files of the series into memory at a time, with **FieldView** keeping an internal list of the transient filenames for reference. Different time steps are accessed through the Transient Data Controls panel (see [Chapter 14](#) of **Working with FieldView** for details). Merge Series is also discussed in [Chapter 1](#) of this **Reference Manual**.

It is possible to read in an entire transient dataset and have access to any particular time step for a particular type of data. The files must be either 2D or a special form of 3D data - with one of the I , J , or K dimensions equal to 1. In either case, the K dimension (in the case of 2D data), or the "1" dimension (in the case of this special type of 3D data) will contain the number of time steps. Sweeping this dimension will be the same as doing a transient sweep.

Example:

You have a transient series of 2D data where the grid is 21×50 . After reading into **FieldView** as a Merge Series solution (see below), the Computational Surface panel will show that I ranges from 1 to 21, J ranges from 1 to 50, and K will range from 1 to N , where N is the number of files that make up the transient series.

Example:

You have a transient series of 3D data where the grid is $33 \times 1 \times 70$. After reading into **FieldView** as a Merge Series solution (see below), the Computational Surface panel will show that I ranges from 1 to 33, J ranges from 1 to N , where N is the number of files that make up the transient series, and K will range from 1 to 70.

The files must have the same prefix and suffix, with only the step number being changed. For example, if time step 1 is called `test001.bin`, all the other time steps must have a name of the form `test###.bin`.

Warning: **FieldView** will not recognize the series if there are more than one dot (".") in the filename. For example, `test.1020.bin`, and files like it, will not be recognized.

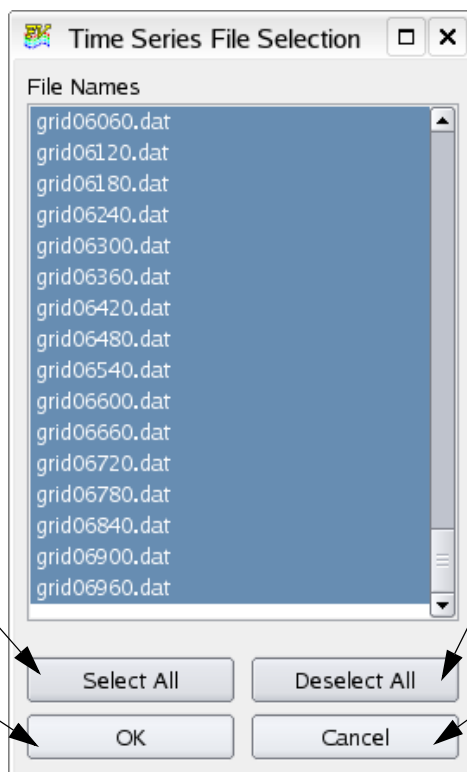
When the Merge Series button is turned on, the Merge Series File Selection panel will be presented. This panel is used to select the files you wish to use as a series. All files with the same prefix and suffix as the selected file will be displayed in this list. Note that using the Merge Series option will append all of the data together in memory, to allow sweeping through all the data by sweeping in one of the computational directions.

Note: This option should not be used for true transient data. Instead, simply read in one file of the transient data, and use the Transient Data Controls panel. See the section on Transient PLOT3D Data in [Chapter 1](#) of this **Reference Manual**.

This list box contains the names of all the files with the same prefix and suffix as the file selected on the PLOT3D Data Input panel. Any file in this window can be selected or unselected by clicking on it with M1.

Press Select All to select (highlight) all the files.

Press OK when the information in the panel is correct. The files will be read in and the sub-panel will be exited.



Press Unselect All to deselect (unhighlight) all the files.

Press Cancel and the files will not be read and the sub-panel will be exited.

Figure 146 Merge Series File Selection

Appendix C Function File Name Format

Function Name files specify scalar and vector function names for the variables in the Function files.

File Naming Convention

The general form for the Function Name file is:

```
file_name.extension
```

The first part of the filename may contain up to 255 alpha characters, numbers, or underscores (Note: spaces are not permitted), although it is suggested that the user limit the filename portion to 22 characters so that the entire name will be visible in the function selection list. The portion of the filename designated as the extension is used to identify the *type* of the file. In this case the required extension is `.nam`.

File Format

The Function Name file is an ASCII text file. Each line contains a scalar variable name. Vector variables are defined by adding a semi-colon and the name of a vector function after a scalar variable name. The components of this vector are defined by the scalar variable on the same line, together with the scalar variables on the next two lines.

Example:

The following shows the contents of a typical Function Name file:

```
pressure
u-velocity; velocity
v-velocity
w-velocity
kinetic energy
```

This file, since it has five lines, supplies names for five variables per grid point. The first variable in a Function File will be assigned the name `pressure`, and so forth. Since there is a semicolon on the second line, the name after the semicolon will be used to define a vector function called `velocity`. Since this vector function begins on the second line, the vector function components include the second variable (`u-velocity`) and the variables on the next two lines (`v-velocity`, `w-velocity`).

Face Data and Function Name Files

FieldView supports face-based results on boundary surfaces of PLOT3D data. In order to provide face results for a 3D dataset, one of the additional files needed is a standard 2D Function File which contains the face results for those boundary surfaces that have them. In order to communicate the names of the face data functions to **FieldView**, a Function Name file needs to be used. Details on the implementation of the additional files needed for face data is contained in [Appendix H](#) of this **Reference Manual**.

Error Conditions

A vector name not followed by at least two more lines in the name file will result in an error. **FieldView** will issue a pop-up message and the vector name will not be used and the vector will not be created. No data will be read. The user will have to designate a different Function Name file or Use Defaults (which will cause the functions to have the names: F1, F2, ... , FN).

If a vector name is defined, another vector name (another semicolon) cannot appear on either of the next two lines in the name file. In other words, overlapping vector definitions are not allowed.

The user is warned if the number of Function Name entries does not correspond to the number of variables in the Function File. Pop-ups will prompt the user for a course of action, either to use default names (F1, etc.), or to use the Function Name file. If the (incomplete) Function Name File is used, it will be padded by default names. That is, if the Name file contains 10 names but the Function file contains 12 functions, the 10 names will be used + F11 and F12.

Blank names will give an error. Either a different Name file will need to be indicated or the default function names will be used.

Although the PLOT3D Function File format allows different numbers of variables on each grid, this is not allowed. **FieldView** validates the file before proceeding.

Appendix D Unstructured Grid Format

General Remarks on Unstructured Data Format

Introductory note

FieldView has a reader for an unstructured file format: FV-UNS. This appendix describes how to create data files that can be read using this reader. Using this file format, you can specify and describe:

1. 3D elements (or cells) including convex arbitrary polyhedrons
2. Multiple grids (or blocks) containing groups of 3D elements or cells
3. Groups of grids (or blocks) defining regions
4. 2D elements (or faces) including arbitrary polygons
5. Surfaces containing groups of 2D elements (or faces), called boundary types
6. Nodal based data for nodes within the volume and on surfaces
7. Surface based results, restricted to boundary types, for face-based data
8. Dataset specific constants for Time, Reynolds No, Free Stream Mach No. and Angle of Attack

Specifically, the format permits you to create files which can contain 1) the grid only, 2) the results only or 3) the combined grid and results file. The current file format version is designated as 3.0. Note that all FV-UNS files based on version 2.7 and earlier can still be read into **FieldView**; there is no need to make changes to existing FV-UNS files as the reader is backward compatible.

The arbitrary polyhedron support was introduced in **FieldView** 9.0 (FV-UNS file version 3.0). The support for arbitrary polyhedrons was expanded in **FieldView** 12.0 to allow non-convex polyhedrons, if they do not have center nodes or hanging face nodes. The **FieldView**-Unstructured-split file format (grid info and results info are stored in different files) was introduced in **FieldView** 8.2 (FV-UNS file version 2.7). The **FieldView**-Unstructured-combined grid and results file pre-dates **FieldView** 8.2 release.

The **FieldView**-Unstructured-split file format permits separation of the grid and result information into different files. For transient simulations in which the grid is invariant (that is, does not change with time), the **FieldView**-Unstructured-split file format will provide a significant savings in terms of disk space, since the grid does not need to be written with the results for each time step. This format can also be used for a series of datasets that are based on the same grid, but are run at differing boundary conditions. The user must explicitly state whether the FV-UNS file contains either grid information, results information or a combination of grid and results data. [Unstructured Data Input panel on page 503](#) of this appendix provides additional instruction on reading FV-UNS files into **FieldView**.

Both **FieldView**-Unstructured-split format and **FieldView**-Unstructured-combined file formats allow for specifying arbitrary polyhedron elements and arbitrary polygon faces (see [Arbitrary Polyhedron Cells on page 417](#)), as well as four standard element types (hexahedron, tetrahedron, prism and pyramid) described in [Standard 3D element types on page 415](#) of this appendix.

The FV-UNS reader can accommodate 3 different file types: Binary, Fortran Unformatted and ASCII. The content of the Binary and Unformatted files is identical, except that the Fortran Unformatted files are organized as a series of Fortran records (one record per Fortran `READ` or `WRITE`). The record structure means that data in Unformatted files must be written with exactly the required Fortran `WRITE` statements, while the data in Binary files can be written any number of words at a time.

Note: Some Fortran compilers can write Binary files using a special option in the Fortran `OPEN` statement, such as `FORM="BINARY"`. Check your compiler documentation to see if your Fortran compiler supports this. Also, some Fortran compilers write Unformatted files using an internal format not supported by **FieldView**. In particular, 64-bit g77 and some versions of 64-bit gfortran use a special 64-bit record structure that is not compatible with 32-bit g77 and gfortran, and is not supported by **FieldView**.

Note that the ASCII file content differs from the Binary/Unformatted files. Consequently some care and planning should be exercised when attempting to change from one type to the other. ASCII files are also slower to read into **FieldView** than Binary or Unformatted files. Information about creating Binary or Unformatted FV-UNS data with FORTRAN 77 can be found in [Creating FV-UNS files with FORTRAN 77 and C for different OS on page 504](#) of this appendix. The FV-UNS reader automatically determines the type of file being read. No byte-swapping is needed for reading FV-UNS files created on a different platform.

The data within FV-UNS files pertaining to both the grid and results must be single precision (4 byte integers or floats). There is currently no support for double precision (8 byte integers or floats). For the case of Binary and Unformatted types, **FieldView** will fail to read the files if any numbers are written using the double precision format. For the case of ASCII files, the double precision format can be read, however the numbers will be truncated to single precision at the time of being read into **FieldView**.

Results are stored on the nodes (at the nodal coordinates). Currently, there is no support for cell centered data. However, cell centered nodes and variables assigned to these nodes may be used for arbitrary polyhedral cells. In other words, cell centered data may be brought into **FieldView** in addition to vertex based data rather than instead of vertex data.

There is support for face based data for boundary surfaces only. This is described in further detail below.

Supported Element Types

General Remarks on Supported Element Types

The FV-UNS data format supports the following standard 3D element types: hexahedron, tetrahedron, prism and pyramid. These elements are described in more detail in [Standard 3D element types on page 415](#).

In addition to the standard 3D elements, **FieldView** 9.0 (and above) supports arbitrary polyhedron 3D elements (cells) that may have cell-centered values and hanging nodes on faces. These elements are described in more detail in [Arbitrary Polyhedron Cells on page 417](#).

The FV-UNS data format supports the following standard 2D element types: quadrilateral and triangle. In addition to the standard 2D elements, **FieldView** 9.0 (and above) supports arbitrary polygon 2D elements (faces). An arbitrary polygon 2D element may have from 3 to 256 vertices. These elements are described in more detail in [Arbitrary Polygon Boundary Faces on page 420](#).

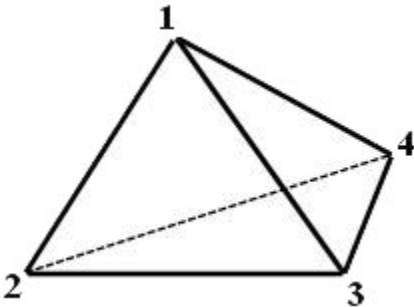
Generally speaking, quadrilaterals and triangles may be considered as particular cases of arbitrary polygon faces. Hexahedron, tetrahedron, prism and pyramid may be considered as particular cases of arbitrary polyhedrons. Therefore, there are two ways to describe these elements in FV-UNS file: as standard elements, and as arbitrary polyhedrons and arbitrary polygon faces.

In binary and unformatted FV-UNS files standard elements and arbitrary polyhedrons / arbitrary polygon faces are assembled into their own sections within the file. In ASCII FV-UNS files standard elements and arbitrary polyhedrons / arbitrary polygon faces share the same sections within the file. The arbitrary polyhedron elements can appear before, after, or in-between standard 3D elements in ASCII FV-UNS file `Elements` section. On the other hand, arbitrary polygon faces should always be written in the end of the ASCII FV-UNS file `Boundary Faces` section, after all standard 2D elements (faces) are written. See the full description of binary, unformatted and ASCII FV-UNS formats below for details.

Standard 3D element types

Four standard 3D element types supported by **FieldView** are hexahedron, tetrahedron, prism and pyramid. Node numbering and face numbering for these elements is shown in **Figure 150** below.

Tetrahedron

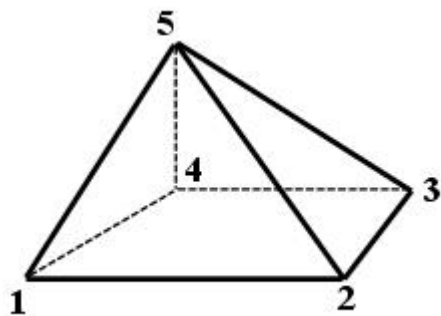


Tetrahedron

face	nodes
1	1 2 3
2	2 3 4
3	3 4 1
4	4 1 2

Figure 147 Face/Node numbering for-

Pyramid

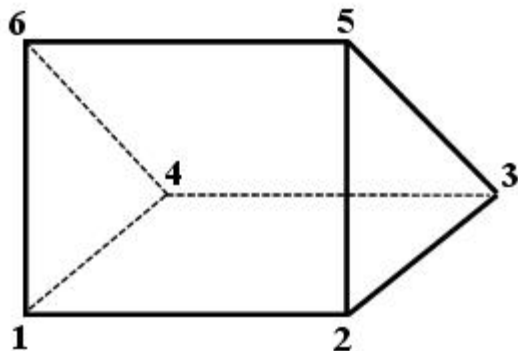


Pyramid

face	nodes
1	1 2 3 4
2	2 3 5
3	3 4 5
4	4 5 1
5	5 1 2

Figure 148 Face/Node numbering for Pyramid Cell type

Prism

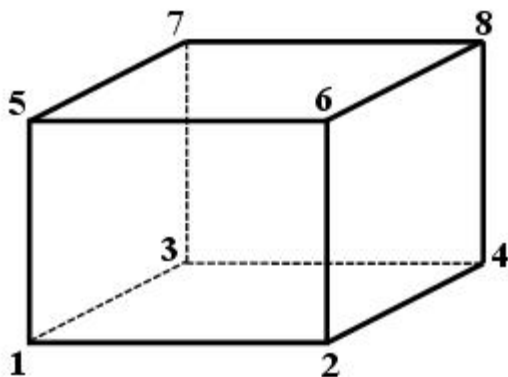


Prism

face	nodes
1	1 2 3 4
2	2 5 6 1
3	3 4 6 5
4	4 6 1
5	5 2 3

Figure 149 Face/Node numbering for Prism Cell type

Hexahedron



Hexahedron

face	nodes
1	1 3 7 5
2	2 6 8 4
3	1 2 4 3
4	5 7 8 6
5	1 5 6 2
6	3 4 8 7

Figure 150 Face/Node numbering for Hexahedron Cell Type

Arbitrary Polyhedron Cells

The Arbitrary Polyhedron 3D element (cell) type adds flexibility to **FieldView**'s handling of unstructured grid files. These cells can have up to 256 faces, each of which can have up to 256 vertices. The faces do not have to be flat (planar). Arbitrary polyhedron cells accommodate trimmed versions of the standard cell types, hanging nodes on grids (for grids that have been refined in certain areas), and polyhedrons such as dodecahedrons. They provide a way to emulate cells with quadratic basis functions. In fact, using cell center values in addition to the nodal values (which are always required) can increase the accuracy of interpolation within the cell. Note that these cells can be freely intermixed with the standard cell types.

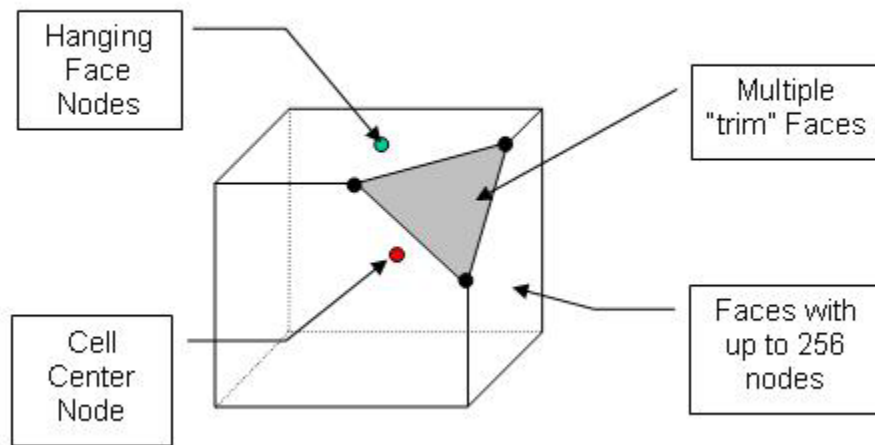


Figure 151 Overview of Arbitrary Polyhedron Cell type

Cell center nodes and hanging face nodes are optional. Arbitrary Polyhedron Cells are described in **FieldView** by providing the following data:

For each cell

- Number of Faces in the cell
- Number of Nodes in the cell (including any cell center or hanging nodes)
- Node ID of cell center data (or a negative integer that is less than 0, if there is no cell center data)

For each Face in the cell

- Wall Flag (just like the standard element types)
- Number of Regular Nodes on the Face
- List of Regular Node IDs on the Face *
- Number of Hanging Nodes on the Face (or 0 if there are none)
- List of Hanging Node IDs (only if preceding value is greater than 0)

* - The ordering of node IDs for each face must be consistent with the other faces of the cell. If one face is clockwise, then all faces of the cell must be clockwise. Otherwise, **FieldView** will reject the cell.

There are some limitations on the types of arbitrary polyhedrons that are accepted by **FieldView**:

- If the cell has a center node or hanging face nodes, then it is converted into tetrahedrons. This causes **FieldView** to use more memory. It also causes additional edges and polygons to appear in surfaces created from these cells. If you want all Arbitrary Polyhedron cells to be converted into tetrahedrons, you can force this by setting the environment variable `FV_TET_CONV`.
- If an Arbitrary Polyhedron cell is converted into tetrahedrons, and you also set the environment variable `FV_ARB_POLY`, then **FieldView** will use a slower but more careful conversion that incorporates any hanging face nodes. The `FV_ARB_POLY` environment variable has no effect on cells that are not converted into tetrahedrons.
- Cells that are converted to tetrahedrons must be convex or "slightly" non-convex (for a cell to be "convex", it must satisfy the criterion that a line connecting any two nodes must be contained entirely within the cell).
- Cells that are converted to tetrahedrons cannot have degenerate nodes on a face (i.e., no two nodes of a face can occupy the same point in space).
- No hanging nodes are permitted on a face edge. Nodes on a face edge should be defined as face nodes rather than hanging nodes, i.e., all hanging nodes are to be in the interior of a face.

Here are two examples of Arbitrary Polyhedron cells. The node numbers in the examples correspond to the node numbers of two Arbitrary Polyhedron cells generated in the code examples located in the `/uns` subdirectory of the directory where **FieldView** is installed.

Example 1 has a center node, so this cell will be converted to tetrahedrons. Example 2 has a hanging face node (not connected to the other nodes of the face). Therefore, Example 2 will also be converted to tetrahedrons.

Example 1: Hex Cell with Trimmed Face and Center Value

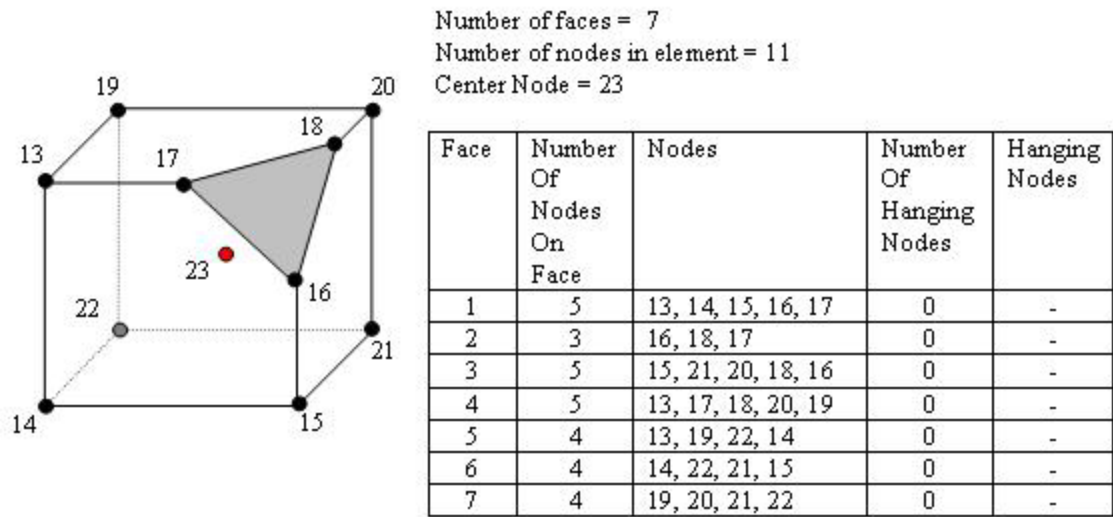


Figure 152 Hex Cell with Trimmed Face and Center Value

Example 2: Hex Cell with Hanging Node, No Center Value

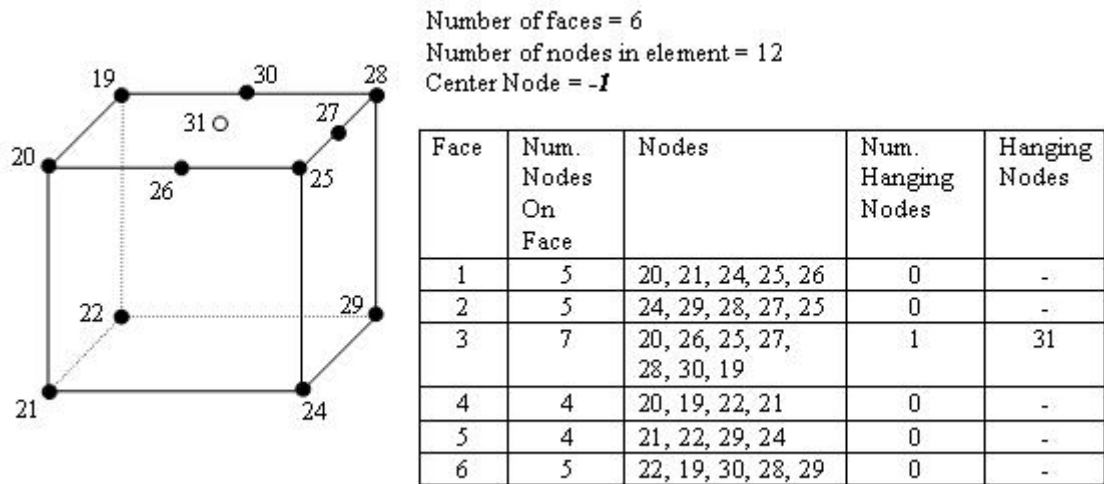


Figure 153 Hex Cell with Hanging Node, No Center Value

The node numbering in the examples above corresponds to the node numbering in code examples located in the /uns subdirectory of the directory where **FieldView** is installed.

Arbitrary Polygon Boundary Faces

The Arbitrary Polygon Boundary Face type adds flexibility to **FieldView**'s handling of boundary surfaces in unstructured grid files. Arbitrary Polygon Boundary Faces are defined by providing a count of the number of nodes that bound the face and the node IDs. The number of nodes that bound an Arbitrary Polygon Boundary Face may vary from 3 to 256. Hanging nodes are not permitted on Arbitrary Polygon Boundary Faces.

Arbitrary Polygon Boundary Faces are handled natively by **FieldView**, and are not converted into triangles. The faces do not have to be convex, and do not have to be flat (planar).

Arbitrary Polygon Boundary Faces support Surface Based results. Surface Based results for Arbitrary Polygon Boundary faces (as well as for standard triangular and quadrilateral faces) are function values that exist only on the boundary faces and not within the flow field of the underlying triangles. No subdivision or distribution is done.

For each grid in a Binary, Unformatted, or ASCII FV-UNS file Arbitrary Polygon Boundary Faces must always be written after all standard triangular and quadrilateral faces have been written. Obviously, triangular and quadrilateral faces that are described as Arbitrary Polygon Boundary Faces (rather than as standard faces) are written with other Arbitrary Polygon Boundary Faces and treated as Arbitrary Polygon Boundary Faces.



Note: An Arbitrary Polygon Boundary Face and a face of an Arbitrary Polyhedron volume element are different entities in FV-UNS file format. Hanging nodes are only permitted on the interior of faces of Arbitrary Polyhedron volume elements.

Arbitrary Polygons are limited to 256 nodes per face and up to 256 faces per cell.

FieldView Compliance for Unstructured Data

The idea of **FieldView** Compliance is that well written FV-UNS files will support both regions (grid or block grouping definitions) and face-based data. Regions allow for independent manipulation of different volume elements of your model, and can be used very effectively in turbomachinery or rotating blade applications. Regions can be made up of one or more grids. The inclusion of one or more grids into a region is established through a corresponding region file, `*.fvreg` (see [Chapter 3](#)). In order to specify regions, multiple grids must be set up within the FV-UNS file first.

Face based data lets you assign a variable per face, instead of per node, on boundary types within the FV-UNS model definition. This offers the benefit of being able to store "surface-only" data for a small part of the entire model. Surface only data, which is often associated with surface fluxes (momentum, heat, mass) can be stored in the format which is more native to the output from finite volume solvers. Subsequent integration of these results will therefore match those determined from a direct analysis of the solver output. Note that face based data can only be stored for faces belonging to a boundary type.

To establish the correspondence between the boundary types and face based data, specific flags are used. Each 2D element or face in a boundary type has a surface results flag and a clockness flag. For the surface results flag, a value of 1 means surface results will be present and a value of 0 means no surface results will be present. Clockness is needed to calculate surface normals. For the clockness flag, a value of 1 means that the face follows a "right hand rule". In other words, if the vertices are written counter-clockwise as in **Figure 154**,

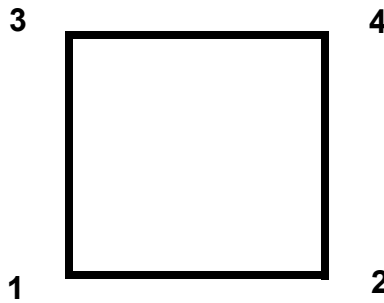


Figure 154 Surface Normal Clockness

the normal to the face is pointing towards you (not away from you). A value of 0 means that the faces do not have any consistent clockness. The clockness of surface normals is only used for calculating surface integral components that involve surface normals. If the clockness flag is 0, these special integral results will not be available.



Important note for users working with older FV-UNS file formats: FV-UNS files in ASCII format (all versions), or in Binary/Unformatted (version 2.2 and earlier) would allow the vertices of quadrilateral faces to be given in arbitrary order. The faces would then be matched up to the element or elements they were attached to and the vertices were taken from the element - not the face - for boundary type definitions in **FieldView**. The Binary/Unformatted FV-UNS file formats (version 2.3 and above) have separate face and element sections, so files that previously read without problems may now result in degenerate boundary faces. Typically they will resemble "bow-ties" when displayed with Display Type set to Mesh. Switching the order of the last two vertices for any degenerate quad faces is all that is required to fix them.

In order to make the task of writing FV-UNS files easier, several example code fragments are supplied as part of the regular **FieldView** installation:

```
write_ascii_uns.f
write_split_ascii_uns.f
write_binary_uns.c
write_split_binary_uns.c
write_unformatted_uns.F
write_split_unformatted_uns.F
fv_reader_tags.h
```

`ftn_fv_reader_tags.h`

These files are located in the `/uns` subdirectory of the directory where **FieldView** is installed. The following sections of this appendix describe the specific detail of each file type, for the split (grid or results only) and combined FV-UNS file formats. References are made to each of the files mentioned above.

Tip: All integer sections (face and element specifications sections) of the FV-UNS file will be scanned for valid data. For exceptionally large files, this may take some time. To speed up input, you may set the environment variable `FV_NO_DATA_CHECK` to a value of 1. However, when this variable is set, no error checking will be done for integer sections of the file. Thus, you should only set this variable if you are very confident of your data integrity.

Binary Format

General Remarks on Binary Format

Binary files can only be created with C and certain FORTRAN compilers (see [Creating FV-UNS files with FORTRAN 77 and C for different OS on page 504](#) for more information). All strings must be written as a record of 80 characters. All characters after the first null character (if any) are discarded. There are no explicit end of file characters. Comment lines are not allowed in binary format.

Split Binary Format

General Remarks on Split Binary Format

Sample C code, called `write_split_binary_uns.c`, for writing the **FieldView**-Unstructured Binary data format has been included in the subdirectory `/uns` of the **FieldView** installation. This sample file provides a framework for use with your own writer and includes tips for easier application. All parameter definitions (`FV_MAGIC`, `FV_ELEMENTS`, etc.) are found in the header file `fv_reader_tags.h` in the `/uns` subdirectory. The sample file also contains two useful functions: `fwrite_str80` and `fv_encode_elem_header`. These are used in the code samples below.

Grid File in Split Binary Format

The first section contains an `open` statement for the grid file:

```
/*Open the grids file for binary write access. */
if ((outfp = fopen(grids_file_name, "wb")) ==
NULL)
{   perror ("Cannot open output file");
    exit(1);}
```

Next section contains a bit pattern to identify the file to **FieldView**. The section must be as follows:

```
/* Output the magic number. */
ibuf[0] = FV_MAGIC;
fwrite(ibuf, sizeof(int), 1, outfp);
```

66051

The next section contains a file type string as shown (strings must be written as a record of 80 characters):

```
/* Output file header and version number. */
fwrite_str80("FIELDVIEW", outfp);

FIELDVIEW
```

Next, the version numbers for the format must be written as shown:

```
/* This version of the FieldView unstructured file is "3.0".
This is written as two integers.*/
ibuf[0] = 3;
ibuf[1] = 0;
fwrite(ibuf, sizeof(int), 2, outfp);

3 0
```

Next, an integer number, the file type code is written:

```
/* File type code - new in version 2.7 */
ibuf[0] = FV_GRIDS_FILE;
fwrite(ibuf, sizeof(int), 1, outfp);

1
```

Next, an integer number, a zero, is written:

```
/* Reserved field, always write a zero - new in version 2.6. */
ibuf[0] = 0;
fwrite(ibuf, sizeof(int), 1, outfp);

0
```

Next, the number of grids must be written. Separating your data into multiple grids is needed in order to use **FieldView** region grouping capabilities. One or more grids may be associated with a region via **FieldView** Region File (see [Chapter 3](#) of the **Reference Manual** for more information on region files). If regions are not to be used, writing multiple grids is still beneficial, as the Grid File will then be suitable for the Grid-Parallel FieldView Server Input options.

```
/* Output the number of grids. */
ibuf[0] = num_grids;
fwrite(ibuf, sizeof(int), 1, outfp);
```

1

Each boundary face (2D element) must have an integer number, the boundary face type, assigned to it. The integer numbers may range from one to the number of different boundary types. This is used to associate the face with a boundary. The next section contains the number of different boundary types (which in the sample source file is equal to 5):

```
ibuf[0] = num_face_types;
fwrite(ibuf, sizeof(int), 1, outfp);
```

5

The next section contains Boundary Types (names of boundaries). The number of Boundary Types should be equal to the number of different boundary types written in the previous section. Each Boundary Type name is preceded by two integer numbers: Surface Results Flag and Clockness Flag. The clockness of surface normals is only used for calculating certain boundary surface integrals that involve surface normals. If the surface normals flag is 0, these special integrals will not be available. Boundary Type names may contain blanks. Boundary Type names should always start with a letter. Each boundary type name should be different from all other boundary type names. The comparison of names is case-insensitive.

Surface Results Flag	= 1	implies face based results present
	= 0	implies no face based results present
Clockness Flag	= 1	implies consistent clockness (for component integral results output)
	= 0	implies no consistent clockness

Character strings must be written as records of 80 characters; this is accomplished by using the `fwrite_str80` function from the sample source.

```
for (i = 0; i < num_face_types; i++) {
  ibuf[0] = results_flag[i];
  ibuf[1] = normals_flag[i];
  fwrite(ibuf, sizeof(int), 2, outfp);
  fwrite_str80(face_type_names[i], outfp);
}
```

```
1 1 bottom
1 1 top
0 0 wall
1 1 trimmed cell
1 1 hanging node cell
```



Note: If you specified more than one grid, the following sections of this file must be repeated for each grid.

The next section contains the node information for the grid being written. First a header word with a numeric field signifies the section number. Next, the number of nodes in this grid is written.

```
/* Output the node definition section for this grid. */
ibuf[0] = FV_NODES;
ibuf[1] = nnodes;
fwrite(ibuf, sizeof(int), 2, outfp);

1001 31
```

This is followed by all of *x*, all of *y*, and all of *z* coordinates for all nodes in this grid. Note that all *x* coordinates are output first, then all *y* and finally all *z*:

```
/* Output the X, then Y, then Z node coordinates.*/
fwrite(x, sizeof(float), nnodes, outfp);
fwrite(y, sizeof(float), nnodes, outfp);
fwrite(z, sizeof(float), nnodes, outfp);

-1. -1. 1. 1. -1. -1. 1. 1. -1. -1. 1. 1. 2. 2. 3. 3. 2.5 3. 2. 3. 3.
2. 2.5 3. 3. 3. 2.5 2. 2. 2.0 2.5
-1. -1. -1. -1. 1. 1. 1. 1. 3. 3. 3. 3. 0. 0. 0. 0. 0. .5 1. 1. 1. 1.
.5 2. 2. 1.5 2. 2. 2. 1.45 1.5
-1. 1. -1. 1. -1. 1. -1. 1. -1. 1. -1. 1. 1. 0. 0. .5 1. 1. 1. 1. 0. 0.
.5 0. 1. 1. 1. 1. 0. 1. 1
```

The Standard Boundary Face section(s) for standard 2D elements is (are) next, starting with a header word `FV_FACES`. Next, you must specify the boundary type (based upon the table above), the number of faces of this type, and then the vertices of each face. In the example presented herein, one face belongs to the boundary type `bottom`. Its Standard Boundary Face section is as follows:

```
ibuf[0] = FV_FACES;
ibuf[1] = 1; /* first boundary type */
ibuf[2] = 1; /* number of faces of this type */
fwrite(ibuf, sizeof(int), 3, outfp);
fwrite(bot_faces, sizeof(int), 4, outfp);

1002 1 1
1 2 3 4
```

where

1002	header signifying the start of the face section
1	specifies boundary type <code>bottom</code> , the first type in the table above
1	the number of separate faces that have this type
1 2 3 4	nodes 1, 2, 3 and 4 make up a quadrilateral face



Note: If the face is triangular (only has 3 nodes), you must specify the fourth vertex as zero.

Caution: A single boundary type can be broken into several sections if you prefer. Also, boundary face sections do not have to be in order. You may have a section of 10 faces of type 3, followed by a section of 20 faces of type 2, followed by a section of 15 more faces of type 3. However, note that breaking a boundary type into very many short sections is less efficient. The boundaries will require more memory and be somewhat slower to calculate in **FieldView**. Also note that you cannot mix standard (triangular and quadrilateral) faces and arbitrary polygon boundary faces in the same section.

Next, a section for the boundary surface of type `top` is written.

```
ibuf[0] = FV_FACES;
ibuf[1] = 2; /* second boundary type */
ibuf[2] = 1; /* number of faces of this type */
fwrite(ibuf, sizeof(int), 3, outfp);
fwrite(top_faces, sizeof(int), 4, outfp);

1002 2 1
9 10 12 11
```

Next, a section for the boundary surface of type `wall` is written.

```
ibuf[0] = FV_FACES;
ibuf[1] = 3; /* third boundary type */
ibuf[2] = 8; /* number of faces of this type */
fwrite(ibuf, sizeof(int), 3, outfp);
fwrite(wall_faces, sizeof(int), 8*4, outfp);

1002 3 8
1 2 6 5
5 6 10 9
3 4 8 7
7 8 12 11
1 3 7 5
5 7 11 9
2 4 8 6
6 8 12 10
```

The Arbitrary Polygon Boundary Face section(s) is (are) next. The semantics are the same as for standard boundary faces. If you are not specifying arbitrary polygon boundary faces, you can skip specifying this section completely. There are one or more separate sections for each boundary face type, as in the case for standard boundary faces. The node ordering for specifying faces should follow the right-handed rule (see [FieldView Compliance for Unstructured Data on page 420](#) for more information). In other words, nodes should be given by walking around the perimeter of the face in a counter-clockwise manner. Hanging nodes are not permitted on boundary faces. In the example presented herein (see code in `/uns` subdirectory of the directory where **FieldView** is installed), there are two Arbitrary Polygon Boundary Face sections, one for the trimmed node cell, and one for the hanging node cell (the cells are shown in the section on [Arbitrary Polyhedron Cells on page 417](#)).

All Standard Boundary Face sections must be written before any Arbitrary Boundary Face sections.

Each Arbitrary Polygon Boundary Face section starts with a header word `FV_ARB_POLY_FACES`, boundary face type and the number of faces for the section. The boundary face type for trimmed cell is 4, and the number of faces of this type is 7.

```
1007          ibuf[0] = FV_ARB_POLY_FACES;
4            ibuf[1] = 4; /* boundary face type */
7            ibuf[2] = 7; /*num faces for the trimmed cell*/
          fwrite(ibuf, sizeof(int), 3, outfp);
```

Next, there is a loop over the faces of type trimmed cell to write cell nodes:

```
for(i = 0; i < num_faces_trim_cell; ++i) /* loop over the faces */
    fwrite(trim_cell_face[i], sizeof(int), trim_cell_face[i][0] + 1,
           outfp);

5 13 14 15 16 17
3 16 18 17
5 15 21 20 18 16
5 13 17 18 20 19
4 13 19 22 14
4 14 22 21 15
4 19 20 21 22
```

The boundary face type for hanging node cell is 5, and the number of faces of this type is 6.

```
ibuf[0] = FV_ARB_POLY_FACES;
ibuf[1] = 5; /* boundary face type */
ibuf[2] = 6; /* num faces for the hanging node cell */
fwrite(ibuf, sizeof(int), 3, outfp);

1007 5 6
```

Next, there is a cycle over the faces of type hanging node cell to write cell nodes:

```
for(i = 0; i < num_faces_hang_cell; ++i) /* loop over the faces */
    fwrite(hang_cell_face[i], sizeof(int), hang_cell_face[i][0] + 1,
           outfp);
```

```
5 20 21 24 25 26
5 24 29 28 27 25
7 20 26 25 27 28 30 19
4 20 19 22 21
4 21 22 29 24
5 22 19 30 28 29
```

The following sections are 3D element sections. There may be as many element sections as needed. Each section may consist of as many elements as needed. There are two kinds of 3D element sections: Standard 3D Element section and Arbitrary Polyhedron section. Each Standard 3D Element section may contain a single element type or a mixture of standard element types (tetrahedron, hexahedron, prism, pyramid). Standard Element sections may be written before, after, or in-between Arbitrary Polyhedron sections. For maximum efficiency, each Standard 3D Element section should contain a significant percentage of the elements of the grid.

The next section is a Standard 3D Element section. It starts with the `FV_ELEMENTS` keyword and includes the number of standard elements of each type (tetrahedron, hexahedron, prism, pyramid).

```
ibuf[0] = FV_ELEMENTS;
ibuf[1] = 0; /* tet count */
ibuf[2] = 2; /* hex count */
ibuf[3] = 0; /* prism count */
ibuf[4] = 0; /* pyramid count */
fwrite(ibuf, sizeof(int), 5, outfp);
```

```
1003 0 2 0 0
```

where

1003	header signifying the start of the elements section
0	number of tetrahedrons
2	number of hexahedrons
0	number of prisms
0	number of pyramids

The standard elements within the Standard 3D Element section can be written in any order, without regard to element type. For each element, a header for that element is written. The header is followed by the node definition for the element. The proper header value is generated by a call to `fv_encode_elem_header` (found in the sample source `write_split_binary_uns.c` in the `/uns` directory). This C routine packs the information about the element type and wall flags into a four byte word that is called the element header. The 3D element face wall flags indicate whether a particular

element face should be treated as a wall in streamline computations. Note that the wall flag data contained in the header is only used during streamline calculation; the data is *not* used as boundary types.

The 3D element types are: 1-tetrahedron, 2-hexahedron, 3-prism, 4-pyramid. See [Standard 3D element types on page 415](#) for node numbering information.

```
/* Write header for first element. */
elem_header = fv_encode_elem_header(FV_HEX_ELEM_ID, hex1_walls);
if (elem_header == 0)
{
    fprintf (stderr, "fv_encode_elem_header failed for first hex.\n");
    exit(1);
}
fwrite (&elem_header, sizeof(elem_header), 1, outfp);
```

First element header – four byte word

```
/* Write node definition for first element. */
fwrite(hex1, sizeof(int), 8, outfp);

1 2 3 4 5 6 7 8

/* Write header for second element. */
elem_header = fv_encode_elem_header(FV_HEX_ELEM_ID, hex2_walls);
if (elem_header == 0)
{
    fprintf (stderr, "fv_encode_elem_header failed for second hex.\n");
    exit(1);
}
fwrite (&elem_header, sizeof(elem_header), 1, outfp);
```

Second element header – four byte word

```
/* Write node definition for second element. */
fwrite(hex2, sizeof(int), 8, outfp);

5 6 7 8 9 10 11 12
```



Important Note: The ordering of the nodes within an element is important. See **Figure 151** in this appendix for details.

The next section is for specifying Arbitrary Polyhedron Elements. This section is not required to be present. If you are not specifying arbitrary polyhedron elements, you can skip this section altogether.

The section consists of one or more arbitrary polyhedron elements. The section starts with the keyword `FV_ARB_POLY_ELEMENTS`.

The wall flag for arbitrary polyhedron elements has the same meaning as for standard elements (in standard element headers), i.e. `A_WALL` or `NOT_A_WALL` (see `fv_reader_tags.h` file in `/uns` subdirectory of the directory where **FieldView** is installed). The wall flag information is used in stream-line computations. The node ordering for specifying faces should be consistent; if one face is clockwise, then all faces of the cell must be clockwise. Hanging nodes are associated with a face interior and should not be on an edge (hanging nodes on an edge should be specified as regular arbitrary polygon boundary face nodes).

In the example presented herein (see code in `/uns` subdirectory of the directory where **FieldView** is installed) there are two arbitrary polyhedron elements: a trimmed cell element and a hanging node element (the cells are shown in [Arbitrary Polyhedron Cells on page 417](#)). The section starts with the `FV_ARB_POLY_ELEMENTS` keyword and the number of elements.

```
ibuf[0] = FV_ARB_POLY_ELEMENTS;
ibuf[1] = 2; /* have 2 elements here */
fwrite(ibuf, sizeof(int), 2, outfp);

1008 2
```

After that, the number of faces, the number of nodes (including the center node if there is one), and the center node number are written for the trimmed cell element. The center node number should be specified as a negative number if there is not a center node.

```
/* trimmed face element */
ibuf[0] = 7; /* num faces for the trimmed cell */
ibuf[1] = 11; /* number of nodes including a center node */
ibuf[2] = 23; /* the center node */
fwrite(ibuf, sizeof(int), 3, outfp);

7 11 23
```

Next, there is a cycle over the faces of the trimmed cell element to write for each face: wall value, number of vertices for the face, node numbers for vertices of the face. In this example, all faces for the element are assumed to be walls. All faces for the element are assumed not to have hanging nodes on them.

```
ibuf[0] = A_WALL; /* wall value */
ibuf[1] = 0; /* number of hanging nodes */

for(i = 0; i < num_faces_trim_cell; ++i) /* write out face info */
{
    fwrite(ibuf, sizeof(int), 1, outfp); /* write wall value */
    fwrite(trim_cell_face[i], sizeof(int), trim_cell_face[i][0] + 1,
```

```

    outfp); /* write num verts and verts */
fwrite(&ibuf[1], sizeof(int), 1, outfp); /* write num hang nodes */
}

7 5 13 14 15 16 17 0
7 3 16 18 17 0
7 5 15 21 20 18 16 0
7 5 13 17 18 20 19 0
7 4 13 19 22 14 0
7 4 14 22 21 15 0
7 4 19 20 21 22 0

```

After that, number of faces, number of provided nodes (no center node exists for the cell), and "-1" for the center node number are written for the hanging node element. A negative integer number for the center node tells **FieldView** that there is no center node specified for this cell.

```

/* hanging node element */
ibuf[0] = 6; /* num faces for the hanging node cell */
ibuf[1] = 12; /* number of nodes including a center node */
            /* (if center node exists) */
ibuf[2] = -1; /* the center node, this indicates that there
            ** is no centernode */
fwrite(ibuf, sizeof(int), 3, outfp);

6 12 -1

```

Next, the following data is written for each face of the trimmed cell element: wall value, number of vertices for the face, node numbers for vertices of the face. All faces for the element are assumed to be walls. All faces for the element except face 3 are assumed not to have hanging nodes at them. Face 3 has one hanging node. Node number for the hanging node is 31.

```

ibuf[0] = A_WALL; /* wall value */
ibuf[1] = 0; /* number of hanging nodes */
ibuf[2] = 1; /* number of hanging nodes for face 3 */
ibuf[3] = 31; /* the node number for the hanging node on face 3 */
for(i = 0; i < 2; ++i) /* write out face info for first 2 faces */
{
    fwrite(ibuf, sizeof(int), 1, outfp); /* write wall value */
    fwrite(hang_cell_face[i], sizeof(int), hang_cell_face[i][0] + 1,
           outfp); /* write num verts and verts */
    fwrite(&ibuf[1], sizeof(int), 1, outfp); /* write num hang nodes */
}

7 5 20 21 24 25 26 0
7 5 24 29 28 27 25 0

```

```

/* this face has a hanging node */
fwrite(ibuf, sizeof(int), 1, outfp);
fwrite(hang_cell_face[2], sizeof(int), hang_cell_face[2][0] + 1,
outfp);
fwrite(&ibuf[2], sizeof(int), 2, outfp);

7 7 20 26 25 27 28 30 19 1 31

/* write out face info for last 3 faces */
for(i = 3; i < num_faces_hang_cell; ++i)
{
fwrite(ibuf, sizeof(int), 1, outfp); /* write wall value */
fwrite(hang_cell_face[i], sizeof(int), hang_cell_face[i][0] + 1,
outfp); /* write num verts and verts */
fwrite(&ibuf[1], sizeof(int), 1, outfp); /* write num hang nodes */
}

7 4 20 19 22 21 0
7 4 21 22 29 24 0
7 5 22 19 30 28 29 0

```



Note: The arbitrary polyhedron element section(s) can appear before, after, or in-between standard 3D element (tetrahedron, pyramid, prism, hexahedron) sections. There can be any number of both arbitrary polyhedron element sections and standard 3D element sections for any grid. The only requirement is to start each section of standard 3D elements with

FV_ELEMENTS keyword, and to start each section of arbitrary polyhedron elements with FV_ARB_POLY_ELEMENTS keyword.

Closing of grid file:

```

if (fclose(outfp) != 0)
{
perror ("Cannot close output file");
exit(1);
}

```

Results File in Split Binary Format

The first section contains an `open` statement for the results file:

```

/*Open the results file for binary write access. */
if ((outfp = fopen(results_file_name, "wb"))
== NULL)
{
perror ("Cannot open output file");
exit(1);}

```

The next section contains a bit pattern to identify the file to **FieldView**. The section must be as follows:

```
/* Output the magic number. */
ibuf[0] = FV_MAGIC;
fwrite(ibuf, sizeof(int), 1, outfp);

66051
```

The next section contains a file type string as shown (strings must be written as a record of 80 characters):

```
/* Output file header and version number. */
fwrite_str80("FIELDVIEW", outfp);

FIELDVIEW
```

Next, the version numbers for the format must be written as shown:

```
/* This version of the FieldView unstructured file is "3.0".
This is written as two integers.*/
ibuf[0] = 3;
ibuf[1] = 0;
fwrite(ibuf, sizeof(int), 2, outfp);

3 0
```

Next, an integer number, the file type code is written:

```
/* File type code - new in version 2.7 */
ibuf[0] = FV_RESULTS_FILE;
fwrite(ibuf, sizeof(int), 1, outfp);

2
```

Next, an integer number, a zero, is written:

```
/* Reserved field - new in version 2.6 */
/* Always write a zero. */
ibuf[0] = 0;
fwrite(ibuf, sizeof(int), 1, outfp);

0
```

Next is the solution time, `TIME`, and 3 constants, `FSMACH`, `ALPHA` and `RE` that may be used by the "CFD Calculator". If your results are not transient, you should put a floating point zero for the time value. Similarly, if you do not wish to use these constants, use zero for these values as well.

The solution time and the 3 constants are all floating point numbers.

```
/* Output constants for time, fsmach, alpha, re */
fwrite(consts, sizeof(float), 4, outfp);

1.0 0.0 0.0 0.0
```

Next, the number of grids must be written:

```
/* Output the number of grids. */
ibuf[0] = num_grids;
fwrite(ibuf, sizeof(int), 1, outfp);

1
```

The next section contains the number of volume (nodal) variables in the file, followed by the names of the variables. When listing the names of the variables, a vector is indicated by following the first component of the vector with a semicolon and the name of the vector. This will indicate that this variable and the next two listed are the three components of the vector (note that a vector is counted as three variables). The variable names may contain blanks.



Note: The number of variables can be zero, meaning the file contains no information on volume variables. If this is the case, the number of variables, "0", still has to be present in the file.

```
/* Output the table of variable names. */
/* The number of variables can be zero. */
ibuf[0] = num_vars;
fwrite(ibuf, sizeof(int), 1, outfp);
for (i = 0; i < num_vars; i++)
    fwrite_str80(var_names[i], outfp);

4
pressure
uvel; velocity
vvel
wvel
```

The next section contains the number and names of boundary variables in the file. Boundary variables are associated with boundary faces, rather than with grid nodes. **FieldView** will automatically append `[BNDRY]` to each name so boundary variables can be easily distinguished from volume (nodal) vari-

ables. The number of boundary variables can be different from the number of volume variables. The number of boundary variables can also be zero. If this is the case, the number of boundary variables, "0", still has to be present in the file.

```
ibuf[0] = num_bvars;
fwrite(ibuf, sizeof(int), 1, outfp);
for (i = 0; i < num_bvars; i++)
    fwrite_str80(bvar_names[i], outfp);
```

```
4
temperature
uvel; velocity
vvel
wvel
```



Note: If you specified more than one grid, the following sections of this file must be repeated for each grid.

The next section contains some node information for the grid. First the header word `FV_NODES` is written. Next, the number of nodes in this grid is written. It should be the same as the number of nodes in the corresponding grid file.

```
/* Output node count only for this grid. */
ibuf[0] = FV_NODES;
ibuf[1] = nnodes;
fwrite(ibuf, sizeof(int), 2, outfp);
```

```
1001
31
```

Next, the variable section is listed. This begins with a header, followed by the results of each variable. Note that the results are in single precision. This section header is required even if the number of variables is zero.

```
ibuf[0] = FV_VARIABLES;
fwrite(ibuf, sizeof(int), 1, outfp);
for (i = 0; i < num_vars; i++)
    fwrite(vars[i], sizeof(float), nnodes, outfp);
```

```
1004
1.0 0.1 1.2 0.1
1.1 0.2 1.1 0.2
...
1.18 1.18 1.18 1.18
```

(124 real numbers - 4 variables for 31 nodes)



Note: One needs to write out all of the results, in node order, for variable 1 (in this case `pressure`), then all of the results, in node order, for variable 2 (in this case `u-velocity`), etc. All of the data for the first variable is output before any of the data for the second variable. The total number of real numbers should match the number of nodes times the number of variables.

Next, the section that contains boundary variable data for standard boundary faces (quadrilaterals and triangles) is written. Remember that the Boundary Table above has a surface results flag indicating which boundary types have face data (surface results). The data should be written in the same order as the faces in the Boundary Faces sections, skipping over faces whose boundary type has a surface results flag of zero (false). For each variable, you should write one number per boundary face. You must write the section header even if the number of boundary variables is zero.

```
ibuf[0] = FV_BNDRY_VARS;
fwrite(ibuf, sizeof(int), 1, outfp);
for (i = 0; i < num_bvars; i++) {
    int num_faces;
    num_faces = 1; /* number of bottom faces */
    fwrite(&bot_bvars[i], sizeof(float), num_faces, outfp);
    num_faces = 1; /* number of top faces */
    fwrite(&top_bvars[i], sizeof(float), num_faces, outfp);
}

1006
1002.11.0
5.5 2.0
5.6 4.0
3.0 2.5
```

For each boundary variable, the boundary variable values for all standard faces are written. In the example presented herein, the first boundary variable value is written for the boundary of the type `bottom`, since the boundary was written first in the Boundary Types section. The first boundary variable value for the boundary of the type `top` follows variable value for the boundary of the type `bottom`. The boundary variable values for the boundary of the type `wall` are skipped, since the surface results flag for the `wall` boundary type was 0 (false) in the Boundary Types section. All of the data for the first boundary variable (at standard faces) is output before any of the data for the second variable. After that, the data for the third variable is written. Finally, the data for the fourth variable is written. The total number of real numbers in the section should match the number of boundary variables times the number of standard boundary faces (quadrilaterals and triangles) that belong to a boundary of a particular type times the number of boundary types that have surface results flag 1 (true).

Next, the section that contains boundary variable data for arbitrary polygon boundary faces is written. This section should always appear after the section that contains boundary variable data for the standard boundary faces (if faces of both standard and arbitrary polygon boundary faces are present in the dataset). Remember that the Boundary Table above has a surface results flag indicating which bound-

ary types have face data (surface results). The data should be written in the same order as the faces in the Arbitrary Polygon Boundary Face sections, skipping over faces whose boundary type has a surface results flag of zero (false). For each variable, you should write one number per arbitrary polygon boundary face.

```
ibuf[0] = FV_ARB_POLY_BNDRY_VARS;
fwrite(ibuf, sizeof(int), 1, outfp);
for (i = 0; i < num_bvars; ++i)
{
    int num_faces;
    num_faces = 7; /* num faces for the trimmed cell */
    fwrite(trim_cell_bvars[i], sizeof(float), num_faces, outfp);
    num_faces = 6; /* num faces for the hanging node cell */
    fwrite(hang_cell_bvars[i], sizeof(float), num_faces, outfp);
}
```

```
1009
1.0  1.1  1.2  1.3  1.4  1.5 1.6
1.1  1.11 1.12 1.13 1.14 1.15
1.7  1.8  1.9  1.1 1.11 1.12 1.13
1.16 1.17 1.18 1.19 1.2  1.21
1.14 1.15 1.16 1.17 1.18 1.19 1.2
1.22 1.23 1.24 1.25 1.26 1.27
1.21 1.22 1.23 1.24 1.25 1.26 1.27
1.28 1.29 1.30 1.31 1.32 1.33
```

Closing of results file:

```
if (fclose(outfp) != 0)
{
    perror ("Cannot close output file");
    exit(1);
}
```

Combined (Grid & Results) Binary Format

Sample C code, called `write_binary_uns.c`, for writing the **FieldView**-Unstructured Binary data format has been included in the subdirectory `/uns` of the **FieldView** installation. This sample file provides a framework for use with your own writer and includes tips for easier application. All parameter definitions (`FV_MAGIC`, `FV_ELEMENTS`, etc.) can be found in the header file `fv_reader_tags.h` in the `uns` directory. The sample file also contains two useful functions: `fwrite_str80` and `fv_encode_elem_header`. These will be used in the code samples below.

The first section contains an `open` statement for the grid file:

```
/* Open the file for binary write access. */
```

```
if ((outfp = fopen(file_name, "wb")) == NULL)
{
    perror ("Cannot open output file");
    exit(1);
}
```

Next section contains a bit pattern to identify the file to **FieldView**. The section must be as follows:

```
/* Output the magic number. */
ibuf[0] = FV_MAGIC;
fwrite(ibuf, sizeof(int), 1, outfp);

66051
```

The next section contains a file type string as shown (strings must be written as a record of 80 characters):

```
/* Output file header and version number. */
fwrite_str80("FIELDVIEW", outfp);

FIELDVIEW
```

Next, the version numbers for the format must be written as shown:

```
/* This version of the FieldView unstructured file is "3.0".
This is written as two integers.*/
ibuf[0] = 3;
ibuf[1] = 0;
fwrite(ibuf, sizeof(int), 2, outfp);

3 0
```

Next, an integer number, the file type code, is written:

```
/* File type code - new in version 2.7 */
ibuf[0] = FV_COMBINED_FILE;
fwrite(ibuf, sizeof(int), 1, outfp);

3
```

Next, an integer number, a zero, is written:

```
/* Reserved field, always write a zero - new in version 2.6. */
ibuf[0] = 0;
fwrite(ibuf, sizeof(int), 1, outfp);
```

0

Next is the solution time, `TIME`, and 3 constants, `FSMACH`, `ALPHA` and `RE` that may be used by the "CFD Calculator". If your results are not transient, you should put a floating point zero for the time value. Similarly, if you do not wish to use these constants, use zero for these values as well.

The solution time and the 3 constants are all floating point numbers.

```
/* Output constants for time, fsmach, alpha, re */
fwrite(consts, sizeof(float), 4, outfp);
```

```
1.0 0.0 0.0 0.0
```

Next, the number of grids must be written. Separating your data into multiple grids is needed in order to use **FieldView** region grouping capabilities. One or more grids may be associated with a region via **FieldView** Region File (see [Chapter 3](#) of the **Reference Manual** for more information on region files). If regions are not to be used, writing multiple grids is still beneficial, as the Grid File will then be suitable for the Grid-Parallel FieldView Server Input options.

```
/* Output the number of grids. */
ibuf[0] = num_grids;
fwrite(ibuf, sizeof(int), 1, outfp);
```

```
1
```

Each boundary face (2D element) must have an integer number, the boundary face type, assigned to it. The integer number may range from one to the number of different boundary types. This is used to associate the face with a boundary. The next section contains the number of different Boundary Types (which in the sample source file is equal to 5):

```
ibuf[0] = num_face_types;
fwrite(ibuf, sizeof(int), 1, outfp);
```

```
5
```

The next section contains Boundary Types (names of boundaries). The number of Boundary Types should be equal to the number of different boundary types written in the previous section. Each Boundary Type name is preceded by two integer numbers: Surface Results Flag and Clockness Flag. The clockness of surface normals is only used for calculating certain boundary surface integrals that involve surface normals. If the surface normals flag is 0, these special integrals will not be available. Boundary Type names may contain blanks. Boundary Type names should always start with a letter. Each boundary type name should be different from all other boundary type names. The comparison of names is case-insensitive.

Surface Results Flag	= 1	implies face based results present
	= 0	implies no face based results present

Clockness Flag	= 1	implies consistent clockness (for component integral results output)
	= 0	implies no consistent clockness

Character strings must be written as records of 80 characters; this is accomplished by using the `fwrite_str80` function from the sample source.

```
for (i = 0; i < num_face_types; i++) {
    ibuf[0] = results_flag[i];
    ibuf[1] = normals_flag[i];
    fwrite(ibuf, sizeof(int), 2, outfp);
    fwrite_str80(face_type_names[i], outfp);
}
```

```
1 1 bottom
1 1 top
0 0 wall
1 1 trimmed cell
1 1 hanging node cell
```

The next section contains the number of volume (nodal) variables in the file, followed by the names of the variables. When listing the names of the variables, a vector is indicated by following the first component of the vector with a semicolon and the name of the vector. This will indicate that this variable and the next two listed are the three components of the vector (note that a vector is counted as three variables). The variable names may contain blanks.



Note: The number of variables can be zero, meaning the file contains no information on volume variables. If this is the case, the number of variables, "0", still has to be present in the file.

```

/* Output the table of variable names. */
/* The number of variables can be zero. */
ibuf[0] = num_vars;
fwrite(ibuf, sizeof(int), 1, outfp);
for (i = 0; i < num_vars; i++)
    fwrite_str80(var_names[i], outfp);

4
pressure
uvel; velocity
vvel
wvel

```

The next section contains the number and names of boundary variables in the file. Boundary variables are associated with boundary faces, rather than with grid nodes. **FieldView** will automatically append [BNDRY] to each name so boundary variables can be easily distinguished from volume (nodal) variables. The number of boundary variables can be different from the number of volume variables. The number of boundary variables can also be zero. If this is the case, the number of boundary variables, "0", still has to be present in the file.

```
ibuf[0] = num_bvars;
fwrite(ibuf, sizeof(int), 1, outfp);
for (i = 0; i < num_bvars; i++)
    fwrite_str80(bvar_names[i], outfp);
```

```
4
temperature
uvel; velocity
vvel
wvel
```



Note: If you specified more than one grid, the following sections of this file must be repeated for each grid.

The next section contains the node information for the grid being written. First a header word with a numeric field signifies the section number. Next, the number of nodes in this grid is written.

```
/* Output the node definition section for this grid. */
ibuf[0] = FV_NODES;
ibuf[1] = nnodes;
fwrite(ibuf, sizeof(int), 2, outfp);
```

```
1001 31
```

This is followed by all of *X*, all of *Y*, and all of *Z* coordinates for all nodes in this grid. Note that all *X* coordinates are output first, then all *Y* and finally all *Z*:

```
/* Output the X, then Y, then Z node coordinates.*/
fwrite(x, sizeof(float), nnodes, outfp);
fwrite(y, sizeof(float), nnodes, outfp);
fwrite(z, sizeof(float), nnodes, outfp);

-1. -1. 1. 1. -1. -1. 1. 1. -1. -1. 1. 1. 2. 2. 3. 3. 2.5 3. 2. 3. 3.
2. 2.5 3. 3. 3. 2.5 2. 2. 2.0 2.5
-1. -1. -1. -1. 1. 1. 1. 1. 3. 3. 3. 3. 0. 0. 0. 0. 0. .5 1. 1. 1. 1.
.5 2. 2. 1.5 2. 2. 2. 1.45 1.5
```

```

-1. 1. -1. 1. -1. 1. -1. 1. -1. 1. -1. 1. 1. 0. 0. .5 1. 1. 1. 1. 0. 0.
.5 0. 1. 1. 1. 1. 0. 1. 1.

```

The Standard Boundary Face section(s) for standard 2D elements is (are) next, starting with a header word `FV_FACES`. Next, you must specify the boundary type (based upon the table above), the number of faces of this type, and then the vertices of each face. In the example presented herein, one face belongs to the boundary type `bottom`. Its Standard Boundary Face section is as follows:

```

ibuf[0] = FV_FACES;
ibuf[1] = 1; /* first boundary type */
ibuf[2] = 1; /* number of faces of this type */
fwrite(ibuf, sizeof(int), 3, outfp);
fwrite(bot_faces, sizeof(int), 4, outfp);

1002 1 1
1 2 3 4

```

where

1002	header signifying the start of the face section
1	specifies boundary type <code>bottom</code> , the first type in the table above
1	the number of separate faces that have this type
1 2 3 4	nodes 1, 2, 3 and 4 make up a quadrilateral face



Note: If the face is triangular (only has 3 nodes), you must specify the fourth vertex as zero.

Caution: A single boundary type can be broken into several sections if you prefer. Also, boundary face sections do not have to be in order. You may have a section of 10 faces of type 3, followed by a section of 20 faces of type 2, followed by a section of 15 more faces of type 3. However, note that breaking a boundary type into very many short sections is less efficient. The boundaries will require more memory and be somewhat slower to calculate in **FieldView**. Also note that you cannot mix standard (triangular and quadrilateral) faces and arbitrary polygon boundary faces in the same section.

Next, a section for the boundary surface of type `top` is written.

```

ibuf[0] = FV_FACES;
ibuf[1] = 2; /* second boundary type */
ibuf[2] = 1; /* number of faces of this type */
fwrite(ibuf, sizeof(int), 3, outfp);
fwrite(top_faces, sizeof(int), 4, outfp);

1002 2 1
9 10 12 11

```

Next, a section for the boundary surface of type `wall` is written.

```

ibuf[0] = FV_FACES;
ibuf[1] = 3; /* third boundary type */
ibuf[2] = 8; /* number of faces of this type */
fwrite(ibuf, sizeof(int), 3, outfp);
fwrite(wall_faces, sizeof(int), 8*4, outfp);

1002 3 8
1 2 6 5
5 6 10 9
3 4 8 7
7 8 12 11
1 3 7 5
5 7 11 9
2 4 8 6
6 8 12 10

```

The Arbitrary Polygon Boundary Face section(s) is (are) next. The semantics are the same as for standard boundary faces. If you are not specifying arbitrary polygon boundary faces, you can skip specifying this section completely. There are one or more separate sections for each boundary face type, as in the case for standard boundary faces. The node ordering for specifying faces should follow the right-handed rule (see [FieldView Compliance for Unstructured Data on page 420](#) for more information). In other words, nodes should be given by walking around the perimeter of the face in a counter-clockwise manner. Hanging nodes are not permitted on boundary faces. In the example presented herein (see code in `/uns` subdirectory of the directory where **FieldView** is installed), there are two Arbitrary Polygon Boundary Face sections, one for the trimmed node cell, and one for the hanging node cell (the cells are shown in the section on [Arbitrary Polyhedron Cells on page 417](#)).

All Standard Boundary Face sections must be written before any Arbitrary Boundary Face sections.

Each Arbitrary Polygon Boundary Face section starts with a header word `FV_ARB_POLY_FACES`, boundary face type and the number of faces for the section. The boundary face type for trimmed cell is `4`, and the number of faces of this type is `7`.

```

1007          ibuf[0] = FV_ARB_POLY_FACES;
4            ibuf[1] = 4; /* boundary face type */
7            ibuf[2] = 7; /*num faces for the trimmed cell*/
              fwrite(ibuf, sizeof(int), 3, outfp);

```

Next, there is a loop over the faces of type trimmed cell to write cell nodes:

```

for(i = 0; i < num_faces_trim_cell; ++i) /* loop over the faces */
    fwrite(trim_cell_face[i], sizeof(int), trim_cell_face[i][0] + 1,
           outfp);

```

```

5 13 14 15 16 17
3 16 18 17
5 15 21 20 18 16
5 13 17 18 20 19
4 13 19 22 14
4 14 22 21 15
4 19 20 21 22

```

The boundary face type for hanging node cell is 5, and the number of faces of this type is 6.

```

ibuf[0] = FV_ARB_POLY_FACES;
ibuf[1] = 5; /* boundary face type */
ibuf[2] = 6; /* num faces for the hanging node cell */
fwrite(ibuf, sizeof(int), 3, outfp);

```

```

1007 5 6

```

Next, there is a cycle over the faces of type hanging node cell to write cell nodes:

```

for(i = 0; i < num_faces_hang_cell; ++i) /* loop over the faces */
    fwrite(hang_cell_face[i], sizeof(int), hang_cell_face[i][0] + 1,
           outfp);

```

```

5 20 21 24 25 26
5 24 29 28 27 25
7 20 26 25 27 28 30 19
4 20 19 22 21
4 21 22 29 24
5 22 19 30 28 29

```

The following sections are 3D element sections. There may be as many element sections as needed. Each section may consist of as many elements as needed. There are two kinds of 3D element sections: Standard 3D Element section and Arbitrary Polyhedron section. Each Standard 3D Element section may contain a single element type or a mixture of standard element types (tetrahedron, hexahedron, prism, pyramid). Standard Element sections may be written before, after, or in-between Arbitrary Polyhedron sections. For maximum efficiency, each Standard 3D Element section should contain a significant percentage of the elements of the grid.

The next section is a Standard 3D Element section. It starts with the `FV_ELEMENTS` keyword and includes the number of standard elements of each type (tetrahedron, hexahedron, prism, pyramid).

```

ibuf[0] = FV_ELEMENTS;
ibuf[1] = 0; /* tet count */
ibuf[2] = 2; /* hex count */
ibuf[3] = 0; /* prism count */

```

```
ibuf[4] = 0; /* pyramid count */
fwrite(ibuf, sizeof(int), 5, outfp);
```

```
1003 0 2 0 0
```

where

1003	header signifying the start of the elements section
0	number of tetrahedrons
2	number of hexahedrons
0	number of prisms
0	number of pyramids

The standard elements within the Standard 3D Element section can be written in any order, without regard to element type. For each element, a header for that element is written. The header is followed by the node definition for the element. The proper header value is generated by a call to `fv_encode_elem_header` (found in the sample source `write_binary_uns.c` in the `/uns` directory). This C routine packs the information about the element type and wall flags into a four byte word that is called the element header. The 3D element face wall flags indicate whether a particular element face should be treated as a wall in streamline computations. Note that the wall flag data contained in the header is only used during streamline calculation; the data is *not* used as boundary types.

The 3D element types are: 1-tetrahedron, 2-hexahedron, 3-prism, 4-pyramid. See [Standard 3D element types on page 415](#) for node numbering information.

```
/* Write header for first element. */
elem_header = fv_encode_elem_header(FV_HEX_ELEM_ID, hex1_walls);
if (elem_header == 0)
{
    fprintf (stderr, "fv_encode_elem_header failed for first hex.\n");
    exit(1);
}
fwrite (&elem_header, sizeof(elem_header), 1, outfp);
```

First element header – four byte word

```
/* Write node definition for first element. */
fwrite(hex1, sizeof(int), 8, outfp);

1 2 3 4 5 6 7 8

/* Write header for second element. */
elem_header = fv_encode_elem_header(FV_HEX_ELEM_ID, hex2_walls);
if (elem_header == 0)
{
    fprintf (stderr, "fv_encode_elem_header failed for second hex.\n");
```

```

exit(1);
    }
    fwrite (&elem_header, sizeof(elem_header), 1, outfp);

```

Second element header – four byte word

```

/* Write node definition for second element. */
fwrite(hex2, sizeof(int), 8, outfp);

5 6 7 8 9 10 11 12

```



Important Note: The ordering of the nodes within an element is important. See **Figure 151** in this appendix for details.

The next section is for specifying Arbitrary Polyhedron Elements. This section is not required to be present. If you are not specifying arbitrary polyhedron elements, you can skip this section altogether. The section consists of one or more arbitrary polyhedron elements. The section starts with the keyword `FV_ARB_POLY_ELEMENTS`.

The wall flag for arbitrary polyhedron elements has the same meaning as for standard elements (in standard element headers), i.e. `A_WALL` or `NOT_A_WALL` (see `fv_reader_tags.h` file in `/uns` subdirectory of the directory where **FieldView** is installed). The wall flag information is used in stream-line computations. The node ordering for specifying faces should be consistent; if one face is clockwise, then all faces of the cell must be clockwise. Hanging nodes are associated with a face interior and should not be on an edge (hanging nodes on an edge should be specified as regular arbitrary polygon boundary face nodes).

In the example presented herein (see code in `/uns` subdirectory of the directory where **FieldView** is installed) there are two arbitrary polyhedron elements: a trimmed cell element and a hanging node element (the cells are shown in [Arbitrary Polyhedron Cells on page 417](#)). The section starts with the `FV_ARB_POLY_ELEMENTS` keyword and the number of elements.

```

ibuf[0] = FV_ARB_POLY_ELEMENTS;
ibuf[1] = 2; /* have 2 elements here */
fwrite(ibuf, sizeof(int), 2, outfp);

1008 2

```

After that, the number of faces, the number of nodes (including the center node if there is one), and the center node number are written for the trimmed cell element. The center node number should be specified as a negative number if there is not a center node.

```

/* trimmed face element */
ibuf[0] = 7; /* num faces for the trimmed cell */

```

```
ibuf[1] = 11; /* number of nodes including a center node */
ibuf[2] = 23; /* the center node */
fwrite(ibuf, sizeof(int), 3, outfp);
```

```
7 11 23
```

Next, there is a cycle over the faces of the trimmed cell element to write for each face: wall value, number of vertices for the face, node numbers for vertices of the face. In this example, all faces for the element are assumed to be walls. All faces for the element are assumed not to have hanging nodes on them.

```
ibuf[0] = A_WALL; /* wall value */
ibuf[1] = 0; /* number of hanging nodes */

for(i = 0; i < num_faces_trim_cell; ++i) /* write out face info */
{
    fwrite(ibuf, sizeof(int), 1, outfp); /* write wall value */
    fwrite(trim_cell_face[i], sizeof(int), trim_cell_face[i][0] + 1,
           outfp); /* write num verts and verts */
    fwrite(&ibuf[1], sizeof(int), 1, outfp); /* write num hang nodes */
}
```

```
7 5 13 14 15 16 17 0
7 3 16 18 17 0
7 5 15 21 20 18 16 0
7 5 13 17 18 20 19 0
7 4 13 19 22 14 0
7 4 14 22 21 15 0
7 4 19 20 21 22 0
```

After that, number of faces, number of provided nodes (no center node exists for the cell), and "-1" for the center node number are written for the hanging node element. A negative integer number for the center node tells **FieldView** that there is no center node specified for this cell.

```
/* hanging node element */
ibuf[0] = 6; /* num faces for the hanging node cell */
ibuf[1] = 12; /* number of nodes including a center node */
           /* (if center node exists) */
ibuf[2] = -1; /* the center node, this indicates that there
              ** is no centernode */
fwrite(ibuf, sizeof(int), 3, outfp);
```

```
6 12 -1
```

Next, the following data is written for each face of the trimmed cell element: wall value, number of vertices for the face, node numbers for vertices of the face. All faces for the element are assumed to be

walls. All faces for the element except face 3 are assumed not to have hanging nodes at them. Face 3 has one hanging node. Node number for the hanging node is 31.

```

ibuf[0] = A_WALL; /* wall value */
ibuf[1] = 0; /* number of hanging nodes */
ibuf[2] = 1; /* number of hanging nodes for face 3 */
ibuf[3] = 31; /* the node number for the hanging node on face 3 */
for(i = 0; i < 2; ++i) /* write out face info for first 2 faces */
{
    fwrite(ibuf, sizeof(int), 1, outfp); /* write wall value */
    fwrite(hang_cell_face[i], sizeof(int), hang_cell_face[i][0] + 1,
           outfp); /* write num verts and verts */
    fwrite(&ibuf[1], sizeof(int), 1, outfp); /* write num hang nodes */
}

7 5 20 21 24 25 26 0
7 5 24 29 28 27 25 0

/* this face has a hanging node */
fwrite(ibuf, sizeof(int), 1, outfp);
fwrite(hang_cell_face[2], sizeof(int), hang_cell_face[2][0] + 1,
outfp);
fwrite(&ibuf[2], sizeof(int), 2, outfp);

7 7 20 26 25 27 28 30 19 1 31

/* write out face info for last 3 faces */
for(i = 3; i < num_faces_hang_cell; ++i)
{
    fwrite(ibuf, sizeof(int), 1, outfp); /* write wall value */
    fwrite(hang_cell_face[i], sizeof(int), hang_cell_face[i][0] + 1,
           outfp); /* write num verts and verts */
    fwrite(&ibuf[1], sizeof(int), 1, outfp); /* write num hang nodes */
}

7 4 20 19 22 21 0
7 4 21 22 29 24 0
7 5 22 19 30 28 29 0

```



Note: The arbitrary polyhedron element section(s) can appear before, after, or in-between standard 3D element (tetrahedron, pyramid, prism, hexahedron) sections. There can be any number of both arbitrary polyhedron element sections and standard 3D element sections for any grid. The only requirement is to start each section of standard 3D elements with

FV_ELEMENTS keyword, and to start each section of arbitrary polyhedron elements with FV_ARB_POLY_ELEMENTS keyword.

Next, the variable section is listed. This begins with a header, followed by the results of each variable. Note that the results are in single precision. This section header is required even if the number of variables is zero.

```
ibuf[0] = FV_VARIABLES;
fwrite(ibuf, sizeof(int), 1, outfp);
for (i = 0; i < num_vars; i++)
    fwrite(vars[i], sizeof(float), nnodes, outfp);
```

```
1004
1.0 0.1 1.2 0.1
1.1 0.2 1.1 0.2
...
1.18 1.18 1.18 1.18
```

(124 real numbers - 4 variables for 31 nodes)



Note: One needs to write out all of the results, in node order, for variable 1 (in this case pressure), then all of the results, in node order, for variable 2 (in this case u-velocity), etc. All of the data for the first variable is output before any of the data for the second variable. The total number of real numbers should match the number of nodes times the number of variables.

Next, the section that contains boundary variable data for standard boundary faces (quadrilaterals and triangles) is written. Remember that the Boundary Table above has a surface results flag indicating which boundary types have face data (surface results). The data should be written in the same order as the faces in the Boundary Faces sections, skipping over faces whose boundary type has a surface results flag of zero (false). For each variable, you should write one number per boundary face. You must write the section header even if the number of boundary variables is zero.

```
ibuf[0] = FV_BNDRY_VARS;
fwrite(ibuf, sizeof(int), 1, outfp);
for (i = 0; i < num_bvars; i++) {
    int num_faces;
    num_faces = 1; /* number of bottom faces */
    fwrite(&bot_bvars[i], sizeof(float), num_faces, outfp);
    num_faces = 1; /* number of top faces */
    fwrite(&top_bvars[i], sizeof(float), num_faces, outfp);
}
```

```
1006
5.1 1.0
5.6 2.0
5.1 4.0
3.0 2.5
```

For each boundary variable, the boundary variable values for all standard faces are written. In the example presented herein, the first boundary variable value is written for the boundary of the type

bottom, since the boundary was written first in the Boundary Types section. The first boundary variable value for the boundary of the type `top` follows variable value for the boundary of the type `bottom`. The boundary variable values for the boundary of the type `wall` are skipped, since the surface results flag for the `wall` boundary type was `0` (false) in the Boundary Types section. All of the data for the first boundary variable (at standard faces) is output before any of the data for the second variable. After that, the data for the third variable is written. Finally, the data for the fourth variable is written. The total number of real numbers in the section should match the number of boundary variables times the number of standard boundary faces (quadrilaterals and triangles) that belong to a boundary of a particular type times the number of boundary types that have surface results flag `1` (true).

Next, the section that contains boundary variable data for arbitrary polygon boundary faces is written. This section should always appear after the section that contains boundary variable data for the standard boundary faces (if faces of both standard and arbitrary polygon boundary faces are present in the dataset). Remember that the Boundary Table above has a surface results flag indicating which boundary types have face data (surface results). The data should be written in the same order as the faces in the Arbitrary Polygon Boundary Face sections, skipping over faces whose boundary type has a surface results flag of zero (false). For each variable, you should write one number per arbitrary polygon boundary face.

```
ibuf[0] = FV_ARB_POLY_BNDRY_VARS;
fwrite(ibuf, sizeof(int), 1, outfp);
for (i = 0; i < num_bvars; ++i)
{
    int num_faces;
    num_faces = 7; /* num faces for the trimmed cell */
    fwrite(trim_cell_bvars[i], sizeof(float), num_faces, outfp);
    num_faces = 6; /* num faces for the hanging node cell */
    fwrite(hang_cell_bvars[i], sizeof(float), num_faces, outfp);
}
```

```
1009
1.0  1.1  1.2  1.3  1.4  1.5 1.6
1.1  1.11 1.12 1.13 1.14 1.15
1.7  1.8  1.9  1.1 1.11 1.12 1.13
1.16 1.17 1.18 1.19 1.2  1.21
1.14 1.15 1.16 1.17 1.18 1.19 1.2
1.22 1.23 1.24 1.25 1.26 1.27
1.21 1.22 1.23 1.24 1.25 1.26 1.27
1.28 1.29 1.30 1.31 1.32 1.33
```

Closing of the file:

```
if (fclose(outfp) != 0)
{
    perror ("Cannot close output file");
    exit(1);
}
```

}

Unformatted (FORTRAN 77) Format

General Remarks on Unformatted (FORTRAN 77) Format

Unformatted files are created with FORTRAN. All of the information in this section assumes FORTRAN 77, unless otherwise specifically mentioned. All strings must be written as a record of 80 characters. All characters after the first null character (if any) are discarded. There are no explicit end of file characters. Comment lines are not allowed in unformatted format.

Split Unformatted (FORTRAN 77) Format

General Remarks on Split Unformatted (FORTRAN 77) Format

In the Split Unformatted (FORTRAN 77) format grid data and results data are stored separately in two files. Sample FORTRAN 77 code, called `write_split_unformatted_uns.F`, has been included in the subdirectory `/uns` of the directory where **FieldView** is installed. This sample file provides a framework for use with your own writer and includes tips for easier application. All parameter definitions (`FV_MAGIC`, `FV_ELEMENTS`, etc.) can be found in the header file `ftn_fv_reader_tags.h` in the subdirectory `/uns` of the directory where **FieldView** is installed. The sample file also contains the useful subroutine `ftn_encode_header`. This will be used in the code samples below.

Grid File in Split Unformatted (FORTRAN 77) Format

The first section contains an `open` statement for the grid file:

```
iunit = 16
open (unit=iunit,
+   file='four_hex_grids.uns',
+   status='UNKNOWN', form='UNFORMATTED',
+   iostat=istat)
if (istat .ne. 0) then
    print *, 'Cannot open file'
    stop 1
endif
```

The next section contains a bit pattern to identify the file to **FieldView**. The section must be as follows:

```
c Output the magic number.
    write(iunit) FV_MAGIC
```

```
66051
```

The next section contains a string as shown. Strings must be padded to 80 characters. This is accomplished by copying them into the 80-character string `txt` (in the sample code):

```
c Output file header
    txt = 'FieldView'
```

```
write(iunit) txt
```

FieldView

Next, the version numbers for the format must be written as shown:

```
c This version of the FieldView unstructured file is "3.0"
c This is written as two integers.
c File type code - new in version 2.7.
c Reserved field, always write a zero -
c new in version 2.6.
```

```
write(iunit) 3, 0, FV_GRIDS_FILE, 0
```

```
3 0 1 0
```

Next, the number of grids must be written. Separating your data into multiple grids is needed in order to use **FieldView** region grouping capabilities. One or more grids may be associated with a region via **FieldView** Region File (see [Chapter 3](#) of the **Reference Manual** for more information on region files). If regions are not to be used, writing multiple grids is still beneficial, as the Grid File will then be suitable for the Grid-Parallel FieldView Server Input options.

```
c Output the number of grids.
  ngrids=1
  write(iunit) ngrids
```

```
1
```

Each face of any element may have its own unique type for association with a boundary surface. The next section contains the number of different boundary types (which in this case is equal to 5):

```
c Output the table of boundary types, starting
c with the number of types.
  write(iunit) 5
```

```
5
```

The next section contains the name of each boundary type preceded by the two integer flags for Surface Results and Clockness respectively. The clockness of surface normals is only used for calculating certain boundary surface integrals that involve surface normals. If the surface normals flag is 0, these special integrals will not be available. There must be as many entries as were specified in the previous section. Strings must be written as a record of 80 characters. The boundary type names may contain blanks. Boundary type names should always start with a letter. Each boundary type name should be different from all other boundary type names. The comparison of names is case-insensitive.

Surface Results Flag = 1 implies face based results present

	= 0	implies no face based results present
Clockness Flag	= 1	implies consistent clockness (for component integral results output)
	= 0	implies no consistent clockness

c We insert a space between the flags and the type name,
c to make it easier to read the ASCII format unstructured file.

```
txt = 'bottom'
write(iunit) 1, 1, txt
txt = 'top'
write(iunit) 1, 1, txt
txt = 'wall'
write(iunit) 0, 0, txt
txt = 'trimmed cell'
write(iunit) 1, 1, txt
txt = 'hanging node cell'
write(iunit) 1, 1, txt
```

```
1 1 bottom
1 1 top
0 0 wall
1 1 trimmed cell
1 1 hanging node cell
```



Note: If you specified more than one grid, the following sections of this file must be repeated for each grid.

The next section contains the node information for the grid being written. First a header word `FV_NODES` with a numeric field signifies the section number. Next, the number of nodes in this grid is written.

```
c Output node definition section for this grid.
  write(iunit) FV_NODES, 31
```

```
1001 31
```

This is followed by all of `X`, all of `Y`, and all of `Z` coordinates for all nodes in this grid. Note that all `X` coordinates are output first, then all `Y` and finally all `Z`, and that all coordinates are output in a single unformatted write statement:

```
c Output node definition section for this grid.
  write(iunit)x, y, z
```

```

-1. -1. 1. 1. -1. -1. 1. 1. -1. -1. 1. 1. 2. 2. 3. 3. 2.5 3. 2. 3. 3. 2. 2.5
3. 3. 3. 2.5 2. 2. 2.0 2.5
-1. -1. -1. -1. 1. 1. 1. 1. 3. 3. 3. 3. 0. 0. 0. 0. 0. .5 1. 1. 1. 1. .5 2.
2. 1.5 2. 2. 2. 1.45 1.5
-1. 1. -1. 1. -1. 1. -1. 1. -1. 1. -1. 1. 1. 0. 0. .5 1. 1. 1. 1. 0. 0. .5
0. 1. 1. 1. 1. 0. 1. 1.

```

The Standard Boundary Face section(s) for standard 2D elements is (are) next, starting with a header word `FV_FACES`. Next, you must specify the boundary type (based upon the table above), the number of faces of this type, and then the vertices of each face. In the example presented herein, for the boundary types of `bottom` (1 face), `top` (1 face) and `wall` (8 faces), the Standard Boundary Face sections are as follows:

c Output boundary faces of the 3 types.

```

write(iunit) FV_FACES, 1, 1
write(iunit) bot_faces
write(iunit) FV_FACES, 2, 1
write(iunit) top_faces
write(iunit) FV_FACES, 3, 8
write(iunit) wall_faces

```

```

1002 1 1
1 2 4 3
1002 2 1
9 10 12 11
1002 3 8
1 2 6 5
5 6 10 9
3 4 8 7
7 8 12 11
1 3 7 5
5 7 11 9
2 4 8 6
6 8 12 10

```

where

1002	header signifying the start of the face section
1	specifies boundary type <code>bottom</code> , the first type in the table above
1	the number of separate faces that have this type
1 2 4 3	face node numbers
1002	header signifying the start of the face section
2	specifies boundary type <code>top</code> , the second type in the table above
1	the number of separate faces that have this type
9 10 12 11	face node numbers
1002	header signifying the start of the face section

3	specifies boundary type	<code>wall</code> , the third type in the table above
8	the number of separate faces that have this type	
1 2 6 5 (etc...)	face node numbers	



Note: If the face is triangular (only has 3 nodes), you must specify the fourth vertex as zero.

Caution: A single boundary type can be broken into several sections if you prefer. Also, boundary face sections do not have to be in order. You may have a section of 10 faces of type 3, followed by a section of 20 faces of type 2, followed by a section of 15 more faces of type 3. However, note that breaking a boundary type into very many short sections is less efficient. The boundaries will require more memory and be somewhat slower to calculate in **FieldView**. Also note that you cannot mix standard (triangular and quadrilateral) faces and arbitrary polygon boundary faces in the same section.

The Arbitrary Polygon Boundary Face section(s) is (are) next. The semantics are the same as for standard boundary faces. If you are not specifying arbitrary polygon boundary faces, you can skip specifying this section completely. There are one or more separate sections for each boundary face type, as in the case for standard boundary faces. The node ordering for specifying faces should follow the right-handed rule (see [FieldView Compliance for Unstructured Data on page 420](#) for more information). In other words, nodes should be given by walking around the perimeter of the face in a counter-clockwise manner. Hanging nodes are not permitted on boundary faces. In the example presented herein (see code in `/uns` subdirectory of the directory where **FieldView** is installed), there are two Arbitrary Polygon Boundary Face sections, one section for `trimmed node cell`, and another one for `hanging node cell` (the cells are shown in the section on [Arbitrary Polyhedron Cells on page 417](#)).

All Standard Boundary Face sections must be written before any Arbitrary Boundary Face sections.

Each Arbitrary Polygon Boundary Face section starts with a header word `FV_ARB_POLY_FACES`, boundary face type and the number of faces for the section. Boundary face type for `trimmed cell` is 4, and the number of faces of this type is 7.

```
write(iunit) FV_ARB_POLY_FACES,4,7
```

```
1007 4 7
```

Next, there is a loop over the faces of type `trimmed cell` to write cell nodes:

```
write(iunit) 5, (trim_cell_face(i,1), i=1,5),
+             3, (trim_cell_face(i,2), i=1,3),
+             5, (trim_cell_face(i,3), i=1,5),
+             5, (trim_cell_face(i,4), i=1,5),
+             4, (trim_cell_face(i,5), i=1,4),
+             4, (trim_cell_face(i,6), i=1,4),
```

```

+                4, (trim_cell_face(i,7), i=1,4)

5 13 14 15 16 17
3 16 18 17
5 15 21 20 18 16
5 13 17 18 20 19
4 13 19 22 14
4 14 22 21 15
4 19 20 21 22

```

Boundary face type for hanging node cell is 5, and the number of faces of this type is 6.

```
write(iunit) FV_ARB_POLY_FACES,5,6
```

```
1007 5 6
```

Next, there is a cycle over the faces of type hanging node cell to write cell nodes:

```

write(iunit) 5, (hang_cell_face(i,1), i=1,5),
+             5, (hang_cell_face(i,2), i=1,5),
+             7, (hang_cell_face(i,3), i=1,7),
+             4, (hang_cell_face(i,4), i=1,4),
+             4, (hang_cell_face(i,5), i=1,4),
+             5, (hang_cell_face(i,6), i=1,5)

5 20 21 24 25 26
5 24 29 28 27 25
7 20 26 25 27 28 30 19
4 20 19 22 21
4 21 22 29 24
5 22 19 30 28 29

```

The following sections are 3D element sections. There may be as many element sections as needed. Each section may consist of as many elements as needed. There are two kinds of 3D element sections: Standard 3D Element section and Arbitrary Polyhedron section. Each Standard 3D Element section may contain a single element type or a mixture of standard element types (tetrahedron, hexahedron, prism, pyramid). Standard Element sections may be written before, after, or in-between Arbitrary Polyhedron sections. For maximum efficiency, each Standard 3D Element section should contain a significant percentage of the elements of the grid.

The next section is a Standard 3D Element section. It starts with the `FV_ELEMENTS` keyword and includes the number of standard elements of each type (tetrahedron, hexahedron, prism, pyramid).

```

c This element section contains 2 hexes.
write(iunit) FV_ELEMENTS, 0, 2, 0, 0

```

1003 0 2 0 0

where

1003	header signifying the start of the elements section
0	number of tetrahedrons
2	number of hexahedrons
0	number of prisms
0	number of pyramids

The standard elements within the Standard 3D Element section can be written in any order, without regard to element type. For each element, a header for that element is written. The header is followed by the node definition for the element. The proper header value is generated by a call to `ftn_encode_header` (found in the sample source `write_split_unformatted_uns.F` in the `/uns` directory where **FieldView** is installed). This FORTRAN routine packs the information about the element type and wall flags into a four byte word that is called the element header. The 3D element face wall flags indicate whether a particular element face should be treated as a wall in streamline computations. Note that the wall flag data contained in the header is only used during streamline calculation; the data is *not* used as boundary types.

The 3D element types are: 1-tetrahedron, 2-hexahedron, 3-prism, 4-pyramid. See [Standard 3D element types on page 415](#) for node numbering information.

The element header and node definition information must be written with a single unformatted write statement for all of the elements belonging to a given section.

```
c The headers and node definitions of all the elements in the section.
c This must be written with a single unformatted write statement.
      write(iunit) (headers(i), (hexes(j,i),j=1,8), i=1,num_elems)
```

First element header – four byte word

1 2 3 4 5 6 7 8

Second element header – four byte word

5 6 7 8 9 10 11 12



Important Note: The ordering of the nodes within an element is important. See **Figure 150** in this appendix for details.

The next section is for specifying Arbitrary Polyhedron Elements. This section is not required to be present. If you are not specifying arbitrary polyhedron elements, you can skip this section altogether. The section consists of one or more arbitrary polyhedron elements. The section starts with the keyword `FV_ARB_POLY_ELEMENTS`.

The wall flag for arbitrary polyhedron elements has the same meaning as for standard elements (in standard element headers), i.e. `A_WALL` or `NOT_A_WALL` (see `ftn_fv_reader_tags.h` file in `/uns` subdirectory of the directory where **FieldView** is installed). The wall flag information is used in streamline computations. The node ordering for specifying faces should be consistent; if one face is clockwise, then all faces of the cell must be clockwise. Hanging nodes are associated with a face interior and should not be on an edge (hanging nodes on an edge should be specified as regular arbitrary polygon boundary face nodes).

In the example presented herein (see code in `/uns` subdirectory of the directory where **FieldView** is installed) there are two arbitrary polyhedron elements: a trimmed cell element and a hanging node element (the cells are shown in [Arbitrary Polyhedron Cells on page 417](#)). The section starts with the `FV_ARB_POLY_ELEMENTS` keyword and the number of elements.

```
write(iunit) FV_ARB_POLY_ELEMENTS, 2
```

```
1008 2
```

After that, the number of faces, the number of nodes (including the center node if there is one), and the center node number are written for the trimmed cell element. The center node number should be specified as a negative number if there is not a center node.

Next, for each face of the trimmed cell element the following values are written: wall value, number of vertices for the face, node numbers for vertices of the face. In this example, all faces for the element are assumed to be walls. All faces for the element are assumed not to have hanging nodes on them.

After that, the number of faces, the number of provided nodes (no center node exists for the cell), and a "-1" for the center node number are written for the hanging node element. A negative integer number for the center node tells **FieldView** to compute the center node coordinates and variable values associated with it.

Next, the following data is written for each face of the trimmed cell element: wall value, number of vertices for the face, node numbers for vertices of the face. All faces for the element are assumed to be walls. All faces for the element except face 3 are assumed not to have hanging nodes at them. Face 3 has one hanging node. Node number for the hanging node is 31.

```
write(iunit) 7, 11, 23,
+      A_WALL, 5, (trim_cell_face(i,1), i=1,5), 0,
+      A_WALL, 3, (trim_cell_face(i,2), i=1,3), 0,
+      A_WALL, 5, (trim_cell_face(i,3), i=1,5), 0,
+      A_WALL, 5, (trim_cell_face(i,4), i=1,5), 0,
+      A_WALL, 4, (trim_cell_face(i,5), i=1,4), 0,
+      A_WALL, 4, (trim_cell_face(i,6), i=1,4), 0,
+      A_WALL, 4, (trim_cell_face(i,7), i=1,4), 0,
+      6, 12, -1,
+      A_WALL, 5, (hang_cell_face(i,1), i=1,5), 0,
+      A_WALL, 5, (hang_cell_face(i,2), i=1,5), 0,
```

```

+      A_WALL, 7, (hang_cell_face(i,3), i=1,7), 1, 31,
+      A_WALL, 4, (hang_cell_face(i,4), i=1,4), 0,
+      A_WALL, 4, (hang_cell_face(i,5), i=1,4), 0,
+      A_WALL, 5, (hang_cell_face(i,6), i=1,5), 0

```

```

7 11 23
7  5 13 14 15 16 17 0
7  3 16 18 17 0
7  5 15 21 20 18 16 0
7  5 13 17 18 20 19 0
7  4 13 19 22 14 0
7  4 14 22 21 15 0
7  4 19 20 21 22 0
6 12 -1
7  5 20 21 24 25 26 0
7  5 24 29 28 27 25 0
7  7 20 26 25 27 28 30 19 1 31
7  4 20 19 22 21 0
7  4 21 22 29 24 0
7  5 22 19 30 28 29 0

```



Note: The arbitrary polyhedron element section(s) can appear before, after, or in-between standard 3D element (tetrahedron, pyramid, prism, hexahedron) sections. There can be any number of both arbitrary polyhedron element sections and standard 3D element sections for any grid. The only requirement is to start each section of standard 3D elements with

FV_ELEMENTS keyword, and to start each section of arbitrary polyhedron elements with FV_ARB_POLY_ELEMENTS keyword.

Closing of grid file:

```
close(iunit)
```

Results File in Split Unformatted (FORTRAN 77) Format

The first section contains an `open` statement for results file:

```

iunit = 16
open (unit=iunit, file='four_hex_results.uns',
+      status='UNKNOWN', form='UNFORMATTED',
+      iostat=istat)
if (istat .ne. 0) then
  print *, 'Cannot open file'
  stop 1
endif

```

The next section contains a bit pattern to identify the file to **FieldView**. The section must be as follows:

```
c  Output the magic number.
    write(iunit) FV_MAGIC
```

```
66051
```

The next section contains a string as shown. Strings must be padded to 80 characters. This is accomplished by copying them into the 80-character string `txt` (in the sample code):

```
c  Output file header
    txt = 'FieldView'
    write(iunit) txt
```

```
FieldView
```

Next, the version numbers for the format must be written as shown:

```
c  This version of the FieldView unstructured file is "3.0".
c  This is written as two integers.
c  File type code FV_RESULTS_FILE - to identify
c  the content of the file as results only -
c  new in version 2.7.
c  Reserved field, always write a zero -
c  new in version 2.6.
    write(iunit) 3, 0, FV_RESULTS_FILE, 0
```

```
3 0 2 0
```

Next is the solution time, `TIME`, and 3 constants, `FSMACH`, `ALPHA` and `RE` that may be used by the "CFD Calculator". If your results are not transient, you should put a floating point zero for the time value. Similarly, if you do not wish to use these constants, use zero for these values as well.

The solution time and the 3 constants are all floating point numbers.

```
c  Output constants for time, fsmach, alpha, re
    write(iunit) 1., 0., 0., 0.
```

```
1. 0. 0. 0.
```

Next, the number of grids must be written:

```
c  Output the number of grids.
    write(iunit) ngrids
```

```
1
```

The next section contains the number of volume (nodal) variables in the file, followed by the names of the variables. When listing the names of the variables, a vector is indicated by following the first component of the vector with a semicolon and the name of the vector. This will indicate that this variable and the next two listed are the three components of the vector (note that a vector is counted as three variables). The variable names may contain blanks.



Note: The number of variables can be zero, meaning the file contains no information on volume variables. If this is the case, the number of variables, "0", still has to be present in the file.

```
c  Output the table of variable names,
c  starting with the number of variables.
c  The number of variables can be zero.
```

```
    nvars = 4
    write(iunit) nvars
    txt = 'pressure'
    write(iunit) txt
    txt = 'uvel; velocity'
    write(iunit) txt
    txt = 'vvel'
    write(iunit) txt
    txt = 'wvel'
    write(iunit) txt
```

```
4
pressure
uvel; velocity
vvel
wvel
```

The next section contains the number and names of boundary variables in the file. Boundary variables are associated with boundary faces, rather than with grid nodes. FieldView will automatically append [BNDRY] to each name so boundary variables can be easily distinguished from ordinary (grid node) variables. The number of boundary variables can be different from the number of ordinary variables. The number of boundary variables can also be zero.

```
c  Output the table of boundary variable names,
c  starting with the number of boundary variables.
c  The number of boundary variables can be zero.
```

```
    nbvars = 4
    write(iunit) nbvars
    txt = 'temperature'
    write(iunit) txt
    txt = 'uvel; velocity'
```

```

write(iunit) txt
txt = 'vvel'
write(iunit) txt
txt = 'wvel'
write(iunit) txt

```

```

4
pressure
uvel; velocity
vvel
wvel

```



Note: If you specified more than one grid, the following sections of this file must be repeated for each grid.

The next section contains some node information for the grid. First the header word `FV_NODES` is written. Next, the number of nodes in this grid is written. It should be the same as the number of nodes in the corresponding grid file.

```

c Output node definition section for this grid.
  write(iunit) FV_NODES, 31

```

```

1001 31

```

Next, the variable section is listed. This begins with a header, followed by the results of each variable. Note that the results are in single precision. This section header is required even if the number of variables is zero. The variable data must be written with a single unformatted write statement for the entire grid.

```

c Output the variable data for this grid.
c This must be a single unformatted write statement.
c The variables must be in the same order as the "Variable Names"
  write(iunit) FV_VARIABLES
  if (nvars .gt. 0) then
    write(iunit) vars
  endif

```

```

1004
1.0 0.1 1.2 0.1
1.1 0.2 1.1 0.2
...
1.18 1.18 1.18 1.18
(124 real numbers - 4 variables for 31 nodes)

```



Note: One needs to write out all of the results, in node order, for variable 1 (in this case `pressure`), then all of the results, in node order, for variable 2 (in this case `u-velocity`), etc. All of the data for the first variable is output before any of the data for the second variable. The total number of real numbers should match the number of nodes times the number of variables.

Next, the section that contains boundary variable data for standard boundary faces (quadrilaterals and triangles) is written. Remember that the Boundary Table above has a surface results flag indicating which boundary types have face data (surface results). The data should be written in the same order as the faces in the Boundary Faces sections, skipping over faces whose boundary type has a surface results flag of zero (false). For each variable, you should write one number per boundary face. You must write the section header even if the number of boundary variables is zero.

The boundary face data must be written with a single unformatted write statement for the entire grid.

```

write(iunit) FV_BNDRY_VARS
  if (nbvars .gt. 0) then
c  NOTE: If this grid has no faces with surface results, then
c  do NOT write an empty record. For grids with no surface
c  results, you should skip the following write statement.
      write(iunit) (bot_bvars(i),top_bvars(i),i=1,nbvars)
  endif

1006
5.2 1.0
5.7 2.0
5.2 4.0
3.0 2.5

```

For each boundary variable, the boundary variable values for all standard faces are written. In the example presented herein, the first boundary variable value is written for the boundary of the type `bottom`, since the boundary was written first in the Boundary Types section. The first boundary variable value for the boundary of the type `top` follows variable value for the boundary of the type `bottom`. The boundary variable values for the boundary of the type `wall` are skipped, since the surface results flag for the `wall` boundary type was 0 (false) in the Boundary Types section. All of the data for the first boundary variable (at standard faces) is output before any of the data for the second variable. After that, the data for the third variable is written. Finally, the data for the fourth variable is written. The total number of real numbers in the section should match the number of boundary variables times the number of standard boundary faces (quadrilaterals and triangles) that belong to a boundary of a particular type times the number of boundary types that have surface results flag 1 (true).

Note: There is no header word '1005'.

Next, the section that contains boundary variable data for arbitrary polygon boundary faces is written. This section should always appear after the section that contains boundary variable data for the standard boundary faces (if faces of both standard and arbitrary polygon boundary faces are present in the

dataset). Remember that the Boundary Table above has a surface results flag indicating which boundary types have face data (surface results). The data should be written in the same order as the faces in the Arbitrary Polygon Boundary Face sections, skipping over faces whose boundary type has a surface results flag of zero (false). For each variable, you should write one number per arbitrary polygon boundary face.

```
write(iunit) FV_BNDRY_VARS
if (nbvars .gt. 0) then
    write(iunit) (bot_bvars(i),top_bvars(i),i=1,nbvars)
endif
```

```
1006
5.3 1.0
5.8 2.0
5.3 4.0
3.0 2.5
```

For each boundary variable, the boundary variable values for all standard faces are written. In the example presented herein, the first boundary variable value is written for the boundary of the type `bottom`, since the boundary was written first in the Boundary Types section. The first boundary variable value for the boundary of the type `top` follows variable value for the boundary of the type `bottom`. The boundary variable values for the boundary of the type `wall` are skipped, since the surface results flag for the `wall` boundary type was `0` (false) in the Boundary Types section. All of the data for the first boundary variable (at standard faces) is output before any of the data for the second variable. After that, the data for the third variable is written. Finally, the data for the fourth variable is written. The total number of real numbers in the section should match the number of boundary variables times the number of standard boundary faces (quadrilaterals and triangles) that belong to a boundary of a particular type times the number of boundary types that have surface results flag `1` (true).

Next, the section that contains boundary variable data for arbitrary polygon boundary faces is written. This section should always appear after the section that contains boundary variable data for the standard boundary faces (if faces of both standard and arbitrary polygon boundary faces are present in the dataset). Remember that the Boundary Table above has a surface results flag indicating which boundary types have face data (surface results). The data should be written in the same order as the faces in the Arbitrary Polygon Boundary Face sections, skipping over faces whose boundary type has a surface results flag of zero (false). For each variable, you should write one number per arbitrary polygon boundary face.

```
if (nbvars .gt. 0) then
    write(iunit) FV_ARB_POLY_BNDRY_VARS
    write(iunit) ((trim_cell_bvars(i,j),j=1,7),
+      (hang_cell_bvars(i,k),k=1,6),i=1,nbvars)
endif
```

```
1009
1.0 1.1 1.2 1.3 1.4 1.5 1.6
```

1.1 1.11 1.12 1.13 1.14 1.15
 1.7 1.8 1.9 1.1 1.11 1.12 1.13
 1.16 1.17 1.18 1.19 1.2 1.21
 1.14 1.15 1.16 1.17 1.18 1.19 1.2
 1.22 1.23 1.24 1.25 1.26 1.27
 1.21 1.22 1.23 1.24 1.25 1.26 1.27
 1.28 1.29 1.30 1.31 1.32 1.33

Closing of results file:

```
close(iunit)
```

Combined (Grid & Results) Unformatted (FORTRAN 77) Format

Sample FORTRAN 77 code, called `write_unformatted_uns.F`, has been included in the subdirectory `/uns` of the directory where **FieldView** is installed. This sample file provides a framework for your own writer and includes tips for easier application. The sample file also contains a useful subroutine `ftn_encode_header`. It will be necessary to refer to this file to correctly write out the element section as described below. All parameter definitions (`FV_MAGIC`, `FV_ELEMENTS`, etc.) can be found in the header file `ftn_fv_reader_tags.h` in the subdirectory `/uns` of the directory where **FieldView** is installed.

The first section contains an `open` statement for the file:

```
iunit = 16
open (unit=iunit, file='four_hex.uns',
+     status='UNKNOWN', form='UNFORMATTED',
+     iostat=istat)
if (istat .ne. 0) then
  print *, 'Cannot open file'
  stop 1
endif
```

The next section contains a bit pattern to identify the file to **FieldView**. The section must be as follows:

```
c Output the magic number.
  write(iunit) FV_MAGIC
```

```
66051
```

The next section contains a string as shown. Strings must be padded to 80 characters. This is accomplished by copying them into the 80-character string `txt` (in the sample code):

```
c Output file header
  txt = 'FieldView'
```

```
write(iunit) txt
```

FieldView

Next, the version numbers for the format must be written as shown:

```
c This version of the FieldView unstructured file is "3.0".
c This is written as two integers.
c File type code - new in version 2.7.
c Reserved field, always write a zero -
c new in version 2.6.
```

```
write(iunit) 3, 0, FV_COMBINED_FILE, 0
```

```
3 0 3 0
```

Next is the solution time, `TIME`, and 3 constants, `FSMACH`, `ALPHA` and `RE` that may be used by the "CFD Calculator". If your results are not transient, you should put a floating point zero for the time value. Similarly, if you do not wish to use these constants, use zero for these values as well.

The solution time and the 3 constants are all floating point numbers.

```
c Output constants for time, fsmach, alpha, re
write(iunit) 1., 0., 0., 0.
```

```
1. 0. 0. 0.
```

Next, the number of grids must be written. Separating your data into multiple grids is needed in order to use **FieldView** region grouping capabilities. One or more grids may be associated with a region via **FieldView** Region File (see [Chapter 3](#) of the **Reference Manual** for more information on region files). If regions are not to be used, writing multiple grids is still beneficial, as the Grid File will then be suitable for the Grid-Parallel FieldView Server Input options.

```
c Output the number of grids.
ngrid=1
write(iunit) ngrid
```

```
1
```

Each face of any element may have its own unique type for association with a boundary surface. The next section contains the number of different boundary types (which in this case is equal to 5):

```
c Output the table of boundary types, starting
c with the number of types.
write(iunit) 5
```

```
5
```

The next section contains the name of each boundary type preceded by the two integer flags for Surface Results and Clockness respectively. The clockness of surface normals is only used for calculating certain boundary surface integrals that involve surface normals. If the surface normals flag is 0, these special integrals will not be available. There must be as many entries as were specified in the previous section. Strings must be written as a record of 80 characters. The boundary type names may contain blanks. Boundary type names should always start with a letter. Each boundary type name should be different from all other boundary type names. The comparison of names is case-insensitive.

Surface Results Flag	= 1	implies face based results present
	= 0	implies no face based results present
Clockness Flag	= 1	implies consistent clockness (for component integral results output)
	= 0	implies no consistent clockness

c We insert a space between the flags and the type name,
c to make it easier to read the ASCII format unstructured file.

```
txt = 'bottom'
write(iunit) 1, 1, txt
txt = 'top'
write(iunit) 1, 1, txt
txt = 'wall'
write(iunit) 0, 0, txt
txt = 'trimmed cell'
write(iunit) 1, 1, txt
txt = 'hanging node cell'
write(iunit) 1, 1, txt
```

```
1 1 bottom
1 1 top
0 0 wall
1 1 trimmed cell
1 1 hanging node cell
```



Note: If you specified more than one grid, the following sections of this file must be repeated for each grid.

The next section contains the node information for the grid being written. First a header word `FV_NODES` with a numeric field signifies the section number. Next, the number of nodes in this grid is written.

c Output node definition section for this grid.

```
write(iunit) FV_NODES, 31
```

```
1001 31
```

This is followed by all of *x*, all of *y*, and all of *z* coordinates for all nodes in this grid. Note that all *x* coordinates are output first, then all *y* and finally all *z*, and that all coordinates are output in a single unformatted write statement:

```
c Output node definition section for this grid.
```

```
write(iunit)x, y, z
```

```
-1. -1. 1. 1. -1. -1. 1. 1. -1. -1. 1. 1. 2. 2. 3. 3. 2.5 3. 2. 3. 3. 2. 2.5
3. 3. 3. 2.5 2. 2. 2.0 2.5
-1. -1. -1. -1. 1. 1. 1. 1. 3. 3. 3. 3. 0. 0. 0. 0. 0. .5 1. 1. 1. 1. .5 2.
2. 1.5 2. 2. 2. 1.45 1.5
-1. 1. -1. 1. -1. 1. -1. 1. -1. 1. -1. 1. 1. 0. 0. .5 1. 1. 1. 1. 0. 0. .5
0. 1. 1. 1. 1. 0. 1. 1.
```

The Standard Boundary Face section(s) for standard 2D elements is (are) next, starting with a header word *FV_FACES*. Next, you must specify the boundary type (based upon the table above), the number of faces of this type, and then the vertices of each face. In the example presented herein, for the boundary types of *bottom* (1 face), *top* (1 face) and *wall* (8 faces), the Standard Boundary Face sections are as follows:

```
c Output boundary faces of the 3 types.
```

```
write(iunit) FV_FACES, 1, 1
```

```
write(iunit) bot_faces
```

```
write(iunit) FV_FACES, 2, 1
```

```
write(iunit) top_faces
```

```
write(iunit) FV_FACES, 3, 8
```

```
write(iunit) wall_faces
```

```
1002 1 1
1 2 4 3
1002 2 1
9 10 12 11
1002 3 8
1 2 6 5
5 6 10 9
3 4 8 7
7 8 12 11
1 3 7 5
5 7 11 9
2 4 8 6
6 8 12 10
```

where

```

1002      header signifying the start of the face section
1         specifies boundary type bottom, the first type in the table above
1         the number of separate faces that have this type
1 2 4 3    face node numbers
1002      header signifying the start of the face section
2         specifies boundary type top, the second type in the table above
1         the number of separate faces that have this type
9 10 12 11 face node numbers
1002      header signifying the start of the face section
3         specifies boundary type wall, the third type in the table above
8         the number of separate faces that have this type
1 2 6 5 (etc...) face node numbers

```



Note: If the face is triangular (only has 3 nodes), you must specify the fourth vertex as zero.

Caution: A single boundary type can be broken into several sections if you prefer. Also, boundary face sections do not have to be in order. You may have a section of 10 faces of type `3`, followed by a section of 20 faces of type `2`, followed by a section of 15 more faces of type `3`. However, note that breaking a boundary type into very many short sections is less efficient. The boundaries will require more memory and be somewhat slower to calculate in **FieldView**. Also note that you cannot mix standard (triangular and quadrilateral) faces and arbitrary polygon boundary faces in the same section.

The Arbitrary Polygon Boundary Face section(s) is (are) next. The semantics are the same as for standard boundary faces. If you are not specifying arbitrary polygon boundary faces, you can skip specifying this section completely. There are one or more separate sections for each boundary face type, as in the case for standard boundary faces. The node ordering for specifying faces should follow the right-handed rule (see [FieldView Compliance for Unstructured Data on page 420](#) for more information). In other words, nodes should be given by walking around the perimeter of the face in a counter-clockwise manner. Hanging nodes are not permitted on boundary faces. In the example presented herein (see code in `/uns` subdirectory of the directory where **FieldView** is installed), there are two Arbitrary Polygon Boundary Face sections, one section for `trimmed node cell`, and another one for `hanging node cell` (the cells are shown in the section on [Arbitrary Polyhedron Cells on page 417](#)).

All Standard Boundary Face sections must be written before any Arbitrary Boundary Face sections.

Each Arbitrary Polygon Boundary Face section starts with a header word `FV_ARB_POLY_FACES`, boundary face type and the number of faces for the section. Boundary face type for `trimmed cell` is `4`, and the number of faces of this type is `7`.

```
write(iunit) FV_ARB_POLY_FACES,4,7
```

1007 4 7

Next, there is a loop over the faces of type trimmed cell to write cell nodes:

```

write(iunit) 5, (trim_cell_face(i,1), i=1,5),
+             3, (trim_cell_face(i,2), i=1,3),
+             5, (trim_cell_face(i,3), i=1,5),
+             5, (trim_cell_face(i,4), i=1,5),
+             4, (trim_cell_face(i,5), i=1,4),
+             4, (trim_cell_face(i,6), i=1,4),
+             4, (trim_cell_face(i,7), i=1,4)

```

```

5 13 14 15 16 17
3 16 18 17
5 15 21 20 18 16
5 13 17 18 20 19
4 13 19 22 14
4 14 22 21 15
4 19 20 21 22

```

The boundary face type for hanging node cell is 5, and the number of faces of this type is 6.

```
write(iunit) FV_ARB_POLY_FACES,5,6
```

1007 5 6

Next, there is a cycle over the faces of type hanging node cell to write cell nodes:

```

write(iunit) 5, (hang_cell_face(i,1), i=1,5),
+             5, (hang_cell_face(i,2), i=1,5),
+             7, (hang_cell_face(i,3), i=1,7),
+             4, (hang_cell_face(i,4), i=1,4),
+             4, (hang_cell_face(i,5), i=1,4),
+             5, (hang_cell_face(i,6), i=1,5)

```

```

5 20 21 24 25 26
5 24 29 28 27 25
7 20 26 25 27 28 30 19
4 20 19 22 21
4 21 22 29 24
5 22 19 30 28 29

```

The following sections are 3D element sections. There may be as many element sections as needed. Each section may consist of as many elements as needed. There are two kinds of 3D element sections: Standard 3D Element section and Arbitrary Polyhedron section. Each Standard 3D Element

section may contain a single element type or a mixture of standard element types (tetrahedron, hexahedron, prism, pyramid). Standard Element sections may be written before, after, or in-between Arbitrary Polyhedron sections. For maximum efficiency, each Standard 3D Element section should contain a significant percentage of the elements of the grid.

The next section is a Standard 3D Element section. It starts with the `FV_ELEMENTS` keyword and includes the number of standard elements of each type (tetrahedron, hexahedron, prism, pyramid).

```
c This element section contains 2 hexes.
      write(iunit) FV_ELEMENTS, 0, 2, 0, 0
```

```
1003 0 2 0 0
```

where

1003	header signifying the start of the elements section
0	number of tetrahedrons
2	number of hexahedrons
0	number of prisms
0	number of pyramids

The standard elements within the Standard 3D Element section can be written in any order, without regard to element type. For each element, a header for that element is written. The header is followed by the node definition for the element. The proper header value is generated by a call to `ftn_encode_header` (found in the sample source `write_unformatted_uns.F` in the `/uns` directory where **FieldView** is installed). This FORTRAN routine packs the information about the element type and wall flags into a four byte word that is called the element header. The 3D element face wall flags indicate whether a particular element face should be treated as a wall in streamline computations. Note that the wall flag data contained in the header is only used during streamline calculation; the data is *not* used as boundary types.

The 3D element types are: 1-tetrahedron, 2-hexahedron, 3-prism, 4-pyramid. See [Standard 3D element types on page 415](#) for node numbering information.

The element header and node definition information must be written with a single unformatted write statement for all of the elements belonging to a given section.

```
c The headers and node definitions of all the elements in the section.
c This must be written with a single unformatted write statement.
      write(iunit) (headers(i), (hexes(j,i),j=1,8), i=1,num_elems)
```

First element header – four byte word

```
1 2 3 4 5 6 7 8
```

Second element header – four byte word

```
5 6 7 8 9 10 11 12
```



Important Note: The ordering of the nodes within an element is important. See **Figure 150** in this appendix for details.

The next section is for specifying Arbitrary Polyhedron Elements. This section is not required to be present. If you are not specifying arbitrary polyhedron elements, you can skip this section altogether. The section consists of one or more arbitrary polyhedron elements. The section starts with the keyword `FV_ARB_POLY_ELEMENTS`.

The wall flag for arbitrary polyhedron elements has the same meaning as for standard elements (in standard element headers), i.e. `A_WALL` or `NOT_A_WALL` (see `ftn_fv_reader_tags.h` file in `/uns` subdirectory of the directory where **FieldView** is installed). The wall flag information is used in streamline computations. The node ordering for specifying faces should be consistent; if one face is clockwise, then all faces of the cell must be clockwise. Hanging nodes are associated with a face interior and should not be on an edge (hanging nodes on an edge should be specified as regular arbitrary polygon boundary face nodes).

In the example presented herein (see code in `/uns` subdirectory of the directory where **FieldView** is installed) there are two arbitrary polyhedron elements: a trimmed cell element and a hanging node element (the cells are shown in [Arbitrary Polyhedron Cells on page 417](#)). The section starts with the `FV_ARB_POLY_ELEMENTS` keyword and the number of elements.

```
write(iunit) FV_ARB_POLY_ELEMENTS, 2
```

```
1008 2
```

After that, the number of faces, the number of nodes (including the center node if there is one), and the center node number are written for the trimmed cell element. The center node number should be specified as a negative number if there is not a center node.

Next, for each face of the trimmed cell element the following values are written: wall value, number of vertices for the face, node numbers for vertices of the face. In this example, all faces for the element are assumed to be walls. All faces for the element are assumed not to have hanging nodes on them.

After that, the number of faces, the number of provided nodes (no center node exists for the cell), and a "-1" for the center node number are written for the hanging node element. A negative integer number for the center node tells **FieldView** that there is no center node for this cell.

Next, the following data is written for each face of the trimmed cell element: wall value, number of vertices for the face, node numbers for vertices of the face. All faces for the element are assumed to be walls. All faces for the element except face 3 are assumed not to have hanging nodes at them. Face 3 has one hanging node. Node number for the hanging node is 31.

```
write(iunit) 7, 11, 23,
```

```

+      A_WALL, 5, (trim_cell_face(i,1), i=1,5), 0,
+      A_WALL, 3, (trim_cell_face(i,2), i=1,3), 0,
+      A_WALL, 5, (trim_cell_face(i,3), i=1,5), 0,
+      A_WALL, 5, (trim_cell_face(i,4), i=1,5), 0,
+      A_WALL, 4, (trim_cell_face(i,5), i=1,4), 0,
+      A_WALL, 4, (trim_cell_face(i,6), i=1,4), 0,
+      A_WALL, 4, (trim_cell_face(i,7), i=1,4), 0,
+      6, 12, -1,
+      A_WALL, 5, (hang_cell_face(i,1), i=1,5), 0,
+      A_WALL, 5, (hang_cell_face(i,2), i=1,5), 0,
+      A_WALL, 7, (hang_cell_face(i,3), i=1,7), 1, 31,
+      A_WALL, 4, (hang_cell_face(i,4), i=1,4), 0,
+      A_WALL, 4, (hang_cell_face(i,5), i=1,4), 0,
+      A_WALL, 5, (hang_cell_face(i,6), i=1,5), 0

```

```

7 11 23
7 5 13 14 15 16 17 0
7 3 16 18 17 0
7 5 15 21 20 18 16 0
7 5 13 17 18 20 19 0
7 4 13 19 22 14 0
7 4 14 22 21 15 0
7 4 19 20 21 22 0
6 12 -1
7 5 20 21 24 25 26 0
7 5 24 29 28 27 25 0
7 7 20 26 25 27 28 30 19 1 31
7 4 20 19 22 21 0
7 4 21 22 29 24 0
7 5 22 19 30 28 29 0

```



Note: The arbitrary polyhedron element section(s) can appear before, after, or in-between standard 3D element (tetrahedron, pyramid, prism, hexahedron) sections. There can be any number of both arbitrary polyhedron element sections and standard 3D element sections for any grid. The only requirement is to start each section of standard 3D elements with

FV_ELEMENTS keyword, and to start each section of arbitrary polyhedron elements with FV_ARB_POLY_ELEMENTS keyword.

Next, the variable section is listed. This begins with a header, followed by the results of each variable. Note that the results are in single precision. This section header is required even if the number of variables is zero. The variable data must be written with a single unformatted write statement for the entire grid.

- c Output the variable data for this grid.
- c This must be a single unformatted write statement.
- c The variables must be in the same order as the "Variable Names"

```

write(iunit) FV_VARIABLES
if (nvars .gt. 0) then
    write(iunit) vars
endif

```

```

1004
1.0 0.1 1.2 0.1
1.1 0.2 1.1 0.2
...
1.18 1.18 1.18 1.18
(124 real numbers - 4 variables for 31 nodes)

```



Note: One needs to write out all of the results, in node order, for variable 1 (in this case pressure), then all of the results, in node order, for variable 2 (in this case u-velocity), etc. All of the data for the first variable is output before any of the data for the second variable. The total number of real numbers should match the number of nodes times the number of variables.

Next, the section that contains boundary variable data for standard boundary faces (quadrilaterals and triangles) is written. Remember that the Boundary Table above has a surface results flag indicating which boundary types have face data (surface results). The data should be written in the same order as the faces in the Boundary Faces sections, skipping over faces whose boundary type has a surface results flag of zero (false). For each variable, you should write one number per boundary face. You must write the section header even if the number of boundary variables is zero.

The boundary face data must be written with a single unformatted write statement for the entire grid.

```

write(iunit) FV_BNDRY_VARS
if (nbvars .gt. 0) then
c  NOTE: If this grid has no faces with surface results, then
c  do NOT write an empty record. For grids with no surface
c  results, you should skip the following write statement.
    write(iunit) (bot_bvars(i),top_bvars(i),i=1,nbvars)
endif

```

```

1006
5.4 1.0
5.9 2.0
5.4 4.0
3.0 2.5

```

For each boundary variable, the boundary variable values for all standard faces are written. In the example presented herein, the first boundary variable value is written for the boundary of the type `bottom`, since the boundary was written first in the Boundary Types section. The first boundary variable value for the boundary of the type `top` follows variable value for the boundary of the type `bottom`. The boundary variable values for the boundary of the type `wall` are skipped, since the surface

results flag for the `wall` boundary type was `0` (false) in the Boundary Types section. All of the data for the first boundary variable (at standard faces) is output before any of the data for the second variable. After that, the data for the third variable is written. Finally, the data for the fourth variable is written. The total number of real numbers in the section should match the number of boundary variables times the number of standard boundary faces (quadrilaterals and triangles) that belong to a boundary of a particular type times the number of boundary types that have surface results flag `1` (true).

Note: There is no header word `'1005'`.

Next, the section that contains boundary variable data for arbitrary polygon boundary faces is written. This section should always appear after the section that contains boundary variable data for the standard boundary faces (if faces of both standard and arbitrary polygon boundary faces are present in the dataset). Remember that the Boundary Table above has a surface results flag indicating which boundary types have face data (surface results). The data should be written in the same order as the faces in the Arbitrary Polygon Boundary Face sections, skipping over faces whose boundary type has a surface results flag of zero (false). For each variable, you should write one number per arbitrary polygon boundary face.

```
write(iunit) FV_BNDRY_VARS
if (nbvars .gt. 0) then
    write(iunit) (bot_bvars(i), top_bvars(i), i=1, nbvars)
endif
```

```
1006
5.5  1.0
5.10 2.0
5.5  4.0
3.0  2.5
```

For each boundary variable, the boundary variable values for all standard faces are written. In the example presented herein, the first boundary variable value is written for the boundary of the type `bottom`, since the boundary was written first in the Boundary Types section. The first boundary variable value for the boundary of the type `top` follows variable value for the boundary of the type `bottom`. The boundary variable values for the boundary of the type `wall` are skipped, since the surface results flag for the `wall` boundary type was `0` (false) in the Boundary Types section. All of the data for the first boundary variable (at standard faces) is output before any of the data for the second variable. After that, the data for the third variable is written. Finally, the data for the fourth variable is written. The total number of real numbers in the section should match the number of boundary variables times the number of standard boundary faces (quadrilaterals and triangles) that belong to a boundary of a particular type times the number of boundary types that have surface results flag `1` (true).

Next, the section that contains boundary variable data for arbitrary polygon boundary faces is written. This section should always appear after the section that contains boundary variable data for the standard boundary faces (if faces of both standard and arbitrary polygon boundary faces are present in the dataset). Remember that the Boundary Table above has a surface results flag indicating which boundary types have face data (surface results). The data should be written in the same order as the faces

in the Arbitrary Polygon Boundary Face sections, skipping over faces whose boundary type has a surface results flag of zero (false). For each variable, you should write one number per arbitrary polygon boundary face.

```

if (nbvars .gt. 0) then
  write(iunit) FV_ARB_POLY_BNDRY_VARS
  write(iunit) ((trim_cell_bvars(i,j),j=1,7),
+      (hang_cell_bvars(i,k),k=1,6),i=1,nbvars)
endif

```

```

1009
1.0  1.1  1.2  1.3  1.4  1.5  1.6
1.1  1.11 1.12 1.13 1.14 1.15
1.7  1.8  1.9  1.1 1.11 1.12 1.13
1.16 1.17 1.18 1.19 1.2  1.21
1.14 1.15 1.16 1.17 1.18 1.19 1.2
1.22 1.23 1.24 1.25 1.26 1.27
1.21 1.22 1.23 1.24 1.25 1.26 1.27
1.28 1.29 1.30 1.31 1.32 1.33

```

Closing of the file:

```
close(iunit)
```

ASCII Format

General Remarks on ASCII Format

The ASCII format and the Binary/Unformatted formats are not the same. Specifically, the flagging of faces as walls is done in the Boundary Table section in the ASCII format, where each boundary type is reported as a wall or not. In the Binary/Unformatted formats, each face of a 3D element is assigned a wall flag in the Elements section. The Binary/Unformatted format is thus more flexible, but requires a little more bookkeeping. The ASCII version also differs in the way that the `XYZ` values of the nodes are output, how the Boundary Faces are output, etc.

Comments can be used in the ASCII format (but not in binary and unformatted format). Any line starting with an exclamation point (!) will be considered a comment and ignored. All keywords in the ASCII file are case-insensitive. There are no explicit end of file characters.

Split ASCII Format

General Remarks on Split ASCII Format

This section will describe the Split ASCII format in detail. In the Split ASCII format, grid data and results data are stored separately in two files. See the sample code `write_split_ascii_uns.f` in the `/uns` sub-directory of the **FieldView** installation. This sample code is used in the following description.

Grid File in Split ASCII Format

The first section contains an `open` statement for the grid file:

```

iunit = 16
open (unit=iunit,
+   file='four_hex_ascii_grids.uns',
+   status='UNKNOWN', form='FORMATTED',
+   iostat=istat)
if (istat .ne. 0) then
    print *, 'Cannot open file'
    stop 1
endif

```

The first line of the file must contain the keyword `FieldView_Grids` followed by two integers as shown. No symbols or spaces are allowed before the keyword `FieldView_Grids`.

```

write(iunit, 1000)
+ 'FieldView_Grids 3 0'
1000 format(13a)

```

FieldView_Grids 3 0



Note: Comments can be used in the ASCII format. Any line starting with an exclamation point (!) will be considered a comment and ignored. All keywords in the ASCII file are case-insensitive.

Next, the keyword `Grids`, followed by the number of grids in the file must be displayed.

```

n grids = 1
write(iunit, *) 'Grids', n grids

```

Grids 1

Separating your data into multiple grids is needed in order to use **FieldView** region grouping capabilities. One or more grids may be associated with a region via **FieldView** Region File (see [Chapter 3](#) of the **Reference Manual** for more information on region files). If regions are not to be used, writing multiple grids is still beneficial, as the Grid File will then be suitable for the Grid-Parallel FieldView Server Input options..

Each boundary face (2D element) must have an integer number, the boundary face type, assigned to it. The integer numbers may range from one to the number of different boundary types (which in the sample source file is equal to 5). This is used to associate the face with a boundary. The next few lines specify the wall and face data results flags and name of each boundary type. A space (' ') is written before each boundary name to separate it from the integer flags. The boundary type names can be up to 80 characters, beginning with a letter. They can contain blanks. Each boundary type

name must be different from all other boundary type names. The comparison of names is case-insensitive.

The section contains the table of boundary types, starting with the key phrase `Boundary Table` and the number of different boundary types. Each boundary type name in the table is preceded by 3 integer flags. Note that this differs from the binary/unformatted specification which has 2 flags.

The first flag indicates whether this boundary type is a wall. A flag value of `1` indicates a wall, and a value of `0` indicates a non-wall. Walls are significant for streamline calculation.

The second flag indicates whether the boundary type has face data (surface results). A value of `1` means face data will be present for this boundary type (if any boundary variables are specified in the Boundary Variable Names section below). A value of `0` means no face data will be present.

The third flag indicates whether boundary faces of this type have consistent clockness for the purpose of calculating a surface normal. A value of `1` means that all faces of this type are written following the “right hand rule” for clockness. In other words, if the face vertices are written in the order that implies walking around the perimeter of the face in counter-clockwise direction (see **Figure 154**) then the normal to the face is pointing towards you (not away from you). A value of `0` means that the faces do not have any consistent clockness. The clockness of surface normals is only used for calculating certain boundary surface integrals that involve surface normals. If the surface normals flag is `0`, these special integrals will not be available.

```
write(iunit, *) 'Boundary Table', 3
write(iunit, *) 0, 1, 1, ' ', 'bottom'
write(iunit, *) 0, 1, 1, ' ', 'top'
write(iunit, *) 1, 0, 0, ' ', 'wall'
write(iunit, *) 1, 1, 1, ' ', 'trimmed cell'
write(iunit, *) 1, 1, 1, ' ', 'hanging node cell'
```

Boundary Table 3

```
0 1 1 bottom
0 1 1 top
1 0 0 wall
1 1 1 trimmed cell
1 1 1 hanging node cell
```



Note: If you specified more than one grid, the following sections of this file must be repeated for each grid.

The next section of the file defines the vertices of the grid. It begins with the keyword `Nodes`, followed by the number of nodes in the grid. Subsequent lines contain the `X`, `Y` and `Z` coordinates of each vertex. The ordering of the vertices is important since a node number will be calculated for each node

corresponding to the order of the vertex in the list. This node number (which is not entered in this section, and which starts with 1) is used later to define the elements.

```

      nnodes = 31
      write(iunit, *) 'Nodes', nnodes
c   Output the X, Y, Z coordinates of successive nodes.
c   Note that this differs from the binary/unformatted specification.
      do 200 i = 1, nnodes
          write(iunit, *) x(i), y(i), z(i)
200 continue

```

```

-1. -1. -1.
-1. -1. 1.
1. -1. -1.
1. -1. 1.
-1. 1. -1.
-1. 1. 1.
1. 1. -1.
1. 1. 1.
-1. 3. -1.
-1. 3. 1.
1. 3. -1.
1. 3. 1.
2. 0. 1.
2. 0. 0.
3. 0. 0.
3. 0. .5
2.5 0. 1.
3. .5 1.
2. 1. 1.
3. 1. 1.
3. 1. 0.
2. 1. 0.
2.5 .5 .5
3. 2. 0.
3. 2. 1.
3. 1.5 1.
2.5 2. 1.
2. 2. 1.
2. 2. 0.
2.0 1.45 1.
2.5 1.5 1.

```

Next, the boundary faces are defined. Each face is preceded by its type number (an index in the the Boundary Table described above) and the number of face vertices. (Note that this differs from the

Binary/Unformatted specification.) The number of face vertices may be 3 or 4 for Standard Faces. The number of face vertices may be from 5 to 256 for Arbitrary Polygon Boundary Faces.

You should have at least one face (either Standard Boundary Face or Arbitrary Polygon Boundary Face) for each non-empty boundary type listed in the boundary table above.

Caution: A single boundary type can be broken into several sections if you prefer. Also, boundary face sections do not have to be in order. You may have a section of 10 faces of type 3, followed by a section of 20 faces of type 2, followed by a section of 15 more faces of type 3. Also note that it is allowed to mix standard (triangular and quadrilateral) faces and arbitrary polygon boundary faces. Hanging nodes are not permitted on boundary faces.

The node ordering for specifying faces should follow the right-handed rule (see [FieldView Compliance for Unstructured Data on page 420](#)). In other words, nodes should be given by walking around the perimeter of the face in a counter-clockwise manner. Hanging nodes are not permitted on boundary faces. In the example presented herein (see code in `/uns` subdirectory of the directory where **FieldView** is installed), there are 10 Standard Faces that belong to `bottom`, `top` and `wall` boundary types, 9 Standard Faces that belong to trimmed node cell and hanging node cell boundary types, and 4 Arbitrary Polygon Boundary Faces that belong to trimmed node cell and hanging node cell boundary types (the cells are shown in the [Arbitrary Polyhedron Cells on page 417](#)). The first boundary face type is `bottom`, with a rectangular face (4 nodes) consisting of nodes 1, 2, 4 and 3. The second boundary face type is `top`, with a rectangular face consisting of nodes 9, 10, 12 and 11. Eight rectangular faces type `wall` are written after that. Nine Standard Faces that belong to trimmed node cell and hanging node cell boundary types are written after that. Four Arbitrary Polygon Boundary Faces that belong to trimmed node cell and hanging node cell boundary types are written after all Standard Boundary Faces are written.

If a face is not a boundary, it is not necessary to define it in this section. The section must start with the keywords `Boundary Faces`, followed by the number of faces that will be defined, as shown below.

```
nbfaces = 10 + 13
write(iunit, *) 'Boundary Faces', nbfaces
```

Boundary Faces 23

```
write(iunit, *) 1, 4, (bot_faces(i,1), i=1,4)
```

```
1 4 1 2 4 3
```

```
write(iunit, *) 2, 4, (top_faces(i,1), i=1,4)
```

```
2 4 9 10 12 11
```

```
do 300 j = 1, 8
```

```
    write(iunit, *) 3, 4, (wall_faces(i,j), i=1,4)
```

```
300 continue
```

```

3 4 1 2 6 5
3 4 5 6 10 9
3 4 3 4 8 7
3 4 7 8 12 11
3 4 1 3 7 5
3 4 5 7 11 9
3 4 2 4 8 6
3 4 6 8 12 10

```

c First, write the triangular and quadrilateral (i.e., 3 or 4 node faces)
c that are a part of the trimmed and hanging node cell.

```

write(iunit, *) 4, 3, (trim_cell_face(i,2), i=1,3)
write(iunit, *) 4, 4, (trim_cell_face(i,5), i=1,4)
write(iunit, *) 4, 4, (trim_cell_face(i,6), i=1,4)
write(iunit, *) 4, 4, (trim_cell_face(i,7), i=1,4)
write(iunit, *) 5, 4, (hang_cell_face(i,4), i=1,4)
write(iunit, *) 5, 4, (hang_cell_face(i,5), i=1,4)

```

```

4 3 16 18 17
4 4 13 19 22 14
4 4 14 22 21 15
4 4 19 20 21 22
5 4 20 19 22 21
5 4 21 22 29 24

```

```

write(iunit, *) 4, 5, (trim_cell_face(i,1), i=1,5)
write(iunit, *) 4, 5, (trim_cell_face(i,3), i=1,5)
write(iunit, *) 4, 5, (trim_cell_face(i,4), i=1,5)

```

```

4 5 13 14 15 16 17
4 5 15 21 20 18 16
4 5 13 17 18 20 19

```

```

write(iunit, *) 5, 5, (hang_cell_face(i,1), i=1,5)
write(iunit, *) 5, 5, (hang_cell_face(i,2), i=1,5)
write(iunit, *) 5, 7, (hang_cell_face(i,3), i=1,7)
write(iunit, *) 5, 5, (hang_cell_face(i,6), i=1,5)

```

```

5 5 20 21 24 25 26
5 5 24 29 28 27 25
5 7 20 26 25 27 28 30 19
5 5 22 19 30 28 29

```

3D elements (cells) are defined in the next section. The section must begin with the keyword `Elements`.

```
write(iunit, *) 'Elements'
```

Elements

The elements are described as follows. Each element will get a type and a subtype (currently always set to one). The Standard 3D element types are: 1-tetrahedron, 2-hexahedron, 3-prism, 4-pyramid. Type 5 is assigned to Arbitrary Polyhedron 3D elements.

Standard 3D elements require between 4 and 8 node numbers (depending on the element type) to define the element. The ordering of the nodes is important. See [Standard 3D element types on page 415](#) for node numbering information. The node numbers follow the element type.

The Arbitrary Polyhedron 3D elements are described as follows. First, the element type, 5, is written. Next, the element subtype, 1 is written (at present, subtype is 1 for all elements; it is a reserved field). After that, the number of faces, the number of nodes including the center node, and the center node number are written. A negative integer number for the center node tells **FieldView** there is no center node. Next, each face of the Arbitrary Polyhedron 3D element is described. For each face, the number of face vertices, the vertex numbers, the number of hanging nodes and the hanging node numbers are written.

In the example presented herein (see code in /uns subdirectory of the directory where **FieldView** is installed) there are two hexahedral 3D elements (cells) and two arbitrary polyhedron elements: a trimmed cell element and a hanging node element (the cells are shown in [Arbitrary Polyhedron Cells on page 417](#)).

Two hexahedral cells are written as follows:

```
write(iunit, *) 2, 1
write(iunit, *) (hexes(i,1), i=1,8)
write(iunit, *) 2, 1
write(iunit, *) (hexes(i,2), i=1,8)
```

```
2 1
1 2 3 4 5 6 7 8
2 1
5 6 7 8 9 10 11 12
```

The first element is a hexahedron, consisting of nodes 1 through 8. The second element is also a hexahedron, consisting of nodes 5 through 12.

Next, for the trimmed cell element we have:

```
write(iunit, *) 5, 1
write(iunit, *) 7, 11, 23
write(iunit, *) 5, (trim_cell_face(i,1), i=1,5), 0
write(iunit, *) 3, (trim_cell_face(i,2), i=1,3), 0
write(iunit, *) 5, (trim_cell_face(i,3), i=1,5), 0
```

```

write(iunit, *) 5, (trim_cell_face(i,4), i=1,5), 0
write(iunit, *) 4, (trim_cell_face(i,5), i=1,4), 0
write(iunit, *) 4, (trim_cell_face(i,6), i=1,4), 0
write(iunit, *) 4, (trim_cell_face(i,7), i=1,4), 0

```

```

5 1
7 11 23
5 13 14 15 16 17 0
3 16 18 17 0
5 15 21 20 18 16 0
5 13 17 18 20 19 0
4 13 19 22 14 0
4 14 22 21 15 0
4 19 20 21 22 0

```

All faces for the element are assumed not to have hanging nodes at them.

For the hanging node element we have:

```

write(iunit, *) 5, 1
write(iunit, *) 6, 12, -1
write(iunit, *) 5, (hang_cell_face(i,1), i=1,5), 0
write(iunit, *) 5, (hang_cell_face(i,2), i=1,5), 0
write(iunit, *) 7, (hang_cell_face(i,3), i=1,7), 1, 31
write(iunit, *) 4, (hang_cell_face(i,4), i=1,4), 0
write(iunit, *) 4, (hang_cell_face(i,5), i=1,4), 0
write(iunit, *) 5, (hang_cell_face(i,6), i=1,5), 0

```

```

5 1
6 12 -1
5 20 21 24 25 26 0
5 24 29 28 27 25 0
7 20 26 25 27 28 30 19 1 31
4 20 19 22 21 0
4 21 22 29 24 0
5 22 19 30 28 29 0

```

All faces for the element except face 3 are assumed not to have hanging nodes at them. Face 3 has one hanging node. Node number for the hanging node is 31.



Note: Keeping arbitrary polyhedron elements together is recommended but not required. The arbitrary polyhedron elements can appear before, after, or in-between standard 3D elements (tetrahedron, pyramid, prism, hexahedron) sections.

Closing of grid file:

```
close(iunit)
```

The following is a sample grid file in ASCII format and a description of a simple 4 hex element mesh. Note that it is not required that values, such as the `Nodes` keyword and `18`, appear all on one line. The format shown below is valid, but less compact. It has been shown this way for ease of readability.

```
FieldView_Grids 3 0
```

Header Line

```
Grids
```

```
1
```

Number of grids

```
Boundary Table
```

```
3
```

Number of boundaries

```
0 0 1 inlet
```

Boundary 1 is of type 0 (does not block flow), no face data present and it has the correct clockness. It is called "inlet".

```
0 0 0 outlet
```

Boundary 2 is of type 0 (does not block flow), no face data present and it has the correct clockness. It is called "outlet".

```
1 1 1 wall boundary
```

Boundary 3 is of type 1 (a wall, does block flow), face data is present and it has the correct clockness. It is called "wall boundary".

```
Nodes
```

```
18
```

Number of nodes

```
0.0 0.0 0.0
```

X Y Z Coordinates of node 1

```
0.5 0.0 0.0
```

X Y Z Coordinates of node 2

```
1.0 0.0 0.0
```

X Y Z Coordinates of node 3

```
0.0 0.5 0.0
```

```
0.5 0.5 0.0
```

```
1.0 0.5 0.0
```

```
0.0 1.0 0.0
```

```
0.5 1.0 0.0
```

```
1.0 1.0 0.0
```

```
0.0 0.0 1.0
```

```
0.5 0.0 1.0
```

```
1.0 0.0 1.0
```

```
0.0 0.5 1.0
```

```
0.5 0.5 1.0
```

```
1.0 0.5 1.0
```

```
0.0 1.0 1.0
```

```
0.5 1.0 1.0
```

```
1.0 1.0 1.0
```

```
Boundary Faces
```

3	Number of boundary faces
1 4 1 4 5 2	Definition of boundary face (boundary type 1, contains 4 nodes, node numbers 1, 4, 5, 2)
2 4 14 17 18 15	
3 4 13 16 17 14	
Elements	
2 1 1 2 10 11 4 5 13 14	Element definition (hex element (type = 2), subtype 1, nodes 1, 2, 10, 11, 4, 5, 13, 14)
2 1 2 3 11 12 5 6 14 15	
2 1 4 5 13 14 7 8 16 17	
2 1 5 6 14 15 8 9 17 18	

The file shown above is used to define the dataset shown in **Figure 155**.

Results File in Split ASCII Format

The first section contains an `open` statement for results file:

```
iunit = 16
open (unit=iunit,
+   file='four_hex_ascii_results.uns',
+   status='UNKNOWN', form='FORMATTED',
+   iostat=istat)
if (istat .ne. 0) then
    print *, 'Cannot open file'
    stop 1
endif
```

The first line of the file must contain the word `FieldView_Results` followed by two integers as shown.

```
write(iunit, 1000)
+   'FieldView_Results 3 0'
1000 format(13a)
```

```
FieldView_Results 3 0
```



Note: Comments can be used in the ASCII format. Any line starting with an exclamation point (!) will be considered a comment and ignored. All keywords in the ASCII file are case-insensitive.

The next section specifies the solution time (`TIME`) and 3 constants (`FSMACH`, `ALPHA`, and `RE`) that may be used by the "CFD Calculator". If your results are not transient, you should put a floating point

zero for the time value. Similarly, if you do not wish to use these constants, use a floating point zero for these values as well. This section is preceded by the keyword `Constants`.

```
write(iunit, *) 'Constants'
write(iunit, *) 1., 0., 0., 0.
```

```
Constants
1.0 0.0 0.0 0.0
```

Next, the keyword `Grids`, followed by the number of grids in the file must be displayed.

```
ngrids = 1
write(iunit, *) 'Grids', ngrids
```

```
Grids 1
```

Separating your data into multiple grids is needed in order to use **FieldView** region grouping capabilities. One or more grids may be associated with a region via **FieldView** Region File (see [Chapter 3](#) of the **Reference Manual** for more information on region files). If regions are not to be used, writing multiple grids is still beneficial, as the Grid File will then be suitable for the Grid-Parallel FieldView Server Input options.

The next section contains the number of volume (nodal) variables in the file, followed by the names of the variables. When listing the names of the variables, a vector is indicated by following the first component of the vector with a semicolon and the name of the vector. This will indicate that this variable and the next two listed are the three components of the vector (note that a vector is counted as three variables). The variable names can be up to 80 characters in length and may contain blanks.



Note: The number of variables can be zero, meaning the file contains no information on volume variables. If this is the case, the number of variables, "0", still has to be present in the file.

The section starts with the keyword `Variable Names`.

```
nvars = 4
write(iunit, *) 'Variable Names', nvars
write(iunit, *) 'pressure'
write(iunit, *) 'uvel; velocity'
write(iunit, *) 'vvel'
write(iunit, *) 'wvel'
```

```
Variable Names 4
pressure
uvel; velocity
vvel
```

wvel

The next section contains the number and names of boundary variables in the file. Boundary variables are associated with boundary faces, rather than with grid nodes. **FieldView** will automatically append [BNDRY] to each name so boundary variables can be easily distinguished from ordinary (grid node) variables. The number of boundary variables can be different from the number of ordinary variables. The number of boundary variables can also be zero. The boundary variable names can be up to 80 characters in length and may contain blanks.

The section should start with the keyword `Boundary Variable Names`.

```
nbvars = 4
write(iunit, *) 'Boundary Variable Names', nbvars
write(iunit, *) 'temperature'
write(iunit, *) 'uvel; velocity'
write(iunit, *) 'vvel'
write(iunit, *) 'wvel'
```

```
Boundary Variable Names 4
temperature
uvel; velocity
vvel
wvel
```



Note: If you specified more than one grid, the following sections of this file must be repeated for each grid.

The next section contains some node information for the grid. First the header word `Nodes` is written. Next, the number of nodes in this grid is written. It should be the same as the number of nodes in corresponding grid file.

```
nnodes = 31
write(iunit, *) 'Nodes', nnodes
```

```
Nodes 31
```

Next, the results variables for each node are listed in the same order as the nodes were defined. The variables must be in the same order as in the Variable Names section. One needs to write out all of the results, in node order, for variable 1 (in this case `pressure`), then all of the results, in node order, for variable 2 (in this case `u-velocity`), etc. All of the data for the first variable is output before any of the data for the second variable. The total number of real numbers should match the number of nodes times the number of variables.

The section must begin with the keyword `Variables`. You should skip this section if the number of variables is zero.

```

        if (nvars .gt. 0) then
            write(iunit, *) 'Variables'
            do 500 j = 1, nvars
                do 600 i = 1, nnodes
                    write(iunit, *) vars(i,j)
                600 continue
            500 continue
        endif

```

Variables

```

1.0 0.1 1.2 0.1
1.1 0.2 1.1 0.2

```

...

```

1.18 1.18 1.18 1.18

```

(124 real numbers - 4 variables for 31 nodes)

Next, output the face data (boundary variable data) for this grid. Note that all of the data for the first variable is output before any of the data for the second variable. Remember that the Boundary Table above has a surface results flag indicating which boundary types have surface results. The data should be written in the same order as the faces in the Boundary Faces section, skipping over faces whose boundary type has a surface results flag of zero (false). For each variable, you should write one number per boundary face. The section must begin with the keyword `Boundary Variables`. You should skip this section if the number of boundary variables is zero.

```

        if (nbvars .gt. 0) then
            write(iunit, *) 'Boundary Variables'
            do 700 j = 1, nbvars
c   The data for the bottom face is written first for each variable, because
c   the bottom face was written first in the "Boundary Faces" section.
c   Write data for variable#j for the "bottom" face:
                write(iunit, *) bot_bvars(j)
c   Write data for variable#j for the "top" face:
                write(iunit, *) top_bvars(j)
c   Skip the "wall" faces, because the surface results flag for the wall
c   boundary type was 0 (false) in the Boundary Table section.
c   Arbitrary Polyhedron boundary variables:
                write(iunit, *) (trim_cell_bvars(j,i), i=1,7)
                write(iunit, *) (hang_cell_bvars(j,i), i=1,6)
            700 continue
        endif

```

Boundary Variables

```

1.0
1.0
1.0 1.1 1.2 1.3 1.4 1.5 1.6
1.1 1.11 1.12 1.13 1.14 1.15

```

```

4.5
2.0
1.7 1.8 1.9 1.1 1.11 1.12 1.13
1.16 1.17 1.18 1.19 1.2 1.21
3.0
4.0
1.14 1.15 1.16 1.17 1.18 1.19 1.2
1.22 1.23 1.24 1.25 1.26 1.27
3.0
2.5
1.21 1.22 1.23 1.24 1.25 1.26 1.27
1.28 1.29 1.30 1.31 1.32 1.33

```

Closing of results file:

```
close(iunit)
```

Important Notes:

1. All of the keywords shown above must be entered exactly as they appear in this document. They are, however, case-insensitive.
2. Except where explicitly stated, every section described above must appear in the data file, even if the capabilities provided by the section are not used. For example, if no Boundary Surfaces are to be defined, both the Boundary Table and the Boundary Faces sections must still appear, with the number of types and faces set to zero.

The following is a sample results file in ASCII format and a description of a simple 4 hex element mesh. Note that it is not required that values, such as constants, appear all on one line. The format shown below is valid, but less compact. It has been shown this way for ease of readability.

FieldView_Grids 3 0	Header Line
Constants	
0.0	Floating point number for TIME
0.2	Floating point number for FSMACH
15.0	Floating point number for ALPHA
1.6e06	Floating point number for RE
Grids	
1	Number of grids
Variable Names	
2	Number of variables
pressure	Names of variables
temperature	
Boundary Variable Names	

2	Number of variables
pressure	Names of variables
temperature	
Nodes	
18	Number of nodes
Variables	
0.0 0.1 0.2 0.1 0.2 0.3	Pressure values (in node order)
0.2 0.3 0.4 1.0 1.1 1.2	
1.1 1.2 1.3 1.2 1.3 1.4	
100 200 300 100 200 300	Temperature values (in node order)
200 300 200 200 350 400	
355 405 500 100 200 300	
Boundary Variables	
0.0157 323	Only "wall" has face data and "wall" has only one face. There is therefore only a single Boundary Variable value for this face for each of the two normal variables, pressure and temperature.

The file shown above is used to define the dataset shown in **Figure 155**.

Combined (Grid & Results) ASCII Format

This section will describe the Combined (Grid & Results) ASCII format in detail. See the sample code `write_ascii_uns.f` in the `/uns` subdirectory of the directory where **FieldView** is installed. This sample code is used in the following description.

The first section contains an `open` statement for the grid file:

```
iunit = 16
open (unit=iunit,
+   file='four_hex_ascii_grids.uns',
+   status='UNKNOWN', form='FORMATTED',
+   iostat=istat)
if (istat .ne. 0) then
    print *, 'Cannot open file'
    stop 1
endif
```

The first line of the file must contain the keyword `FIELDVIEW` followed by two integers as shown. No symbols or spaces are allowed before the keyword `FIELDVIEW`.

```
write(iunit, 1000)
+   'FIELDVIEW 3 0'
```

```
1000 format(13a)
```

```
FIELDVIEW 3 0
```



Note: Comments can be used in the ASCII format. Any line starting with an exclamation point (!) will be considered a comment and ignored. All keywords in the ASCII file are case-insensitive.

The next section specifies the solution time (`TIME`) and 3 constants (`FSMACH`, `ALPHA`, and `RE`) that may be used by the "CFD Calculator". If your results are not transient, you should put a floating point zero for the time value. Similarly, if you do not wish to use these constants, use a floating point zero for these values as well. This section is preceded by the keyword `Constants`.

```
write(iunit, *) 'Constants'
write(iunit, *) 1., 0., 0., 0.
```

```
Constants
1.0 0.0 0.0 0.0
```

Next, the keyword `Grids`, followed by the number of grids in the file must be displayed.

```
ngrids = 1
write(iunit, *) 'Grids', ngrids
```

```
Grids 1
```

Separating your data into multiple grids is needed in order to use **FieldView** region grouping capabilities. One or more grids may be associated with a region via **FieldView** Region File (see [Chapter 3](#) of the **Reference Manual** for more information on region files). If regions are not to be used, writing multiple grids is still beneficial, as the Grid File will then be suitable for the Grid-Parallel FieldView Server Input options.

Each boundary face (2D element) must have an integer number, the boundary face type, assigned to it. The integer numbers may range from one to the number of different boundary types (which in the sample source file is equal to 5). This is used to associate the face with a boundary. The next few lines specify the wall and face data results flags and name of each boundary type. A space (' ') is written before each boundary name to separate it from the integer flags. The boundary type names can be up to 80 characters, beginning with a letter. They can contain blanks. Each boundary type name must be different from all other boundary type names. The comparison of names is case-insensitive.

The section contains the table of boundary types, starting with the key phrase `Boundary Table` and the number of different boundary types. Each boundary type name in the table is preceded by 3 integer flags. Note that this differs from the binary/unformatted specification which has 2 flags.

The first flag indicates whether this boundary type is a wall. A flag value of 1 indicates a wall, and a value of 0 indicates a non-wall. Walls are significant for streamline calculation.

The second flag indicates whether the boundary type has face data (surface results). A value of 1 means face data will be present for this boundary type (if any boundary variables are specified in the Boundary Variable Names section below). A value of 0 means no face data will be present.

The third flag indicates whether boundary faces of this type have consistent clockness for the purpose of calculating a surface normal. A value of 1 means that all faces of this type are written following the “right hand rule” for clockness. In other words, if the face vertices are written in the order that implies walking around the perimeter of the face in counter-clockwise direction (see **Figure 154**) then the normal to the face is pointing towards you (not away from you). A value of 0 means that the faces do not have any consistent clockness. The clockness of surface normals is only used for calculating certain boundary surface integrals that involve surface normals. If the surface normals flag is 0, these special integrals will not be available.

```
write(iunit, *) 'Boundary Table', 3
write(iunit, *) 0, 1, 1, ' ', 'bottom'
write(iunit, *) 0, 1, 1, ' ', 'top'
write(iunit, *) 1, 0, 0, ' ', 'wall'
write(iunit, *) 1, 1, 1, ' ', 'trimmed cell'
write(iunit, *) 1, 1, 1, ' ', 'hanging node cell'
```

Boundary Table 3

```
0 1 1 bottom
0 1 1 top
1 0 0 wall
1 1 1 trimmed cell
1 1 1 hanging node cell
```



Note: If you specified more than one grid, the following sections of this file must be repeated for each grid.

The next section of the file defines the vertices of the grid. It begins with the keyword `Nodes`, followed by the number of nodes in the grid. Subsequent lines contain the `X`, `Y` and `Z` coordinates of each vertex. The ordering of the vertices is important since a node number will be calculated for each node corresponding to the order of the vertex in the list. This node number (which is not entered in this section, and which starts with 1) is used later to define the elements.

```
nnodes = 31
write(iunit, *) 'Nodes', nnodes
c Output the X, Y, Z coordinates of successive nodes.
c Note that this differs from the binary/unformatted specification.
do 200 i = 1, nnodes
```

```

        write(iunit, *) x(i), y(i), z(i)
200 continue

```

```

-1. -1. -1.
-1. -1. 1.
1. -1. -1.
1. -1. 1.
-1. 1. -1.
-1. 1. 1.
1. 1. -1.
1. 1. 1.
-1. 3. -1.
-1. 3. 1.
1. 3. -1.
1. 3. 1.
2. 0. 1.
2. 0. 0.
3. 0. 0.
3. 0. .5
2.5 0. 1.
3. .5 1.
2. 1. 1.
3. 1. 1.
3. 1. 0.
2. 1. 0.
2.5 .5 .5
3. 2. 0.
3. 2. 1.
3. 1.5 1.
2.5 2. 1.
2. 2. 1.
2. 2. 0.
2.0 1.45 1.
2.5 1.5 1.

```

Next, the boundary faces are defined. Each face is preceded by its type number (an index in the the Boundary Table described above) and the number of face vertices. (Note that this differs from the Binary/Unformatted specification.) The number of face vertices may be 3 or 4 for Standard Faces. The number of face vertices may be from 5 to 256 for Arbitrary Polygon Boundary Faces.

You should have at least one face (either Standard Boundary Face or Arbitrary Polygon Boundary Face) for each non-empty boundary type listed in the boundary table above.

Caution: A single boundary type can be broken into several sections if you prefer. Also, boundary face sections do not have to be in order. You may have a section of 10 faces of type 3, followed by a section of 20 faces of type 2, followed by a section of 15 more faces of type 3. Also note that it is

allowed to mix standard (triangular and quadrilateral) faces and arbitrary polygon boundary faces. Hanging nodes are not permitted on boundary faces.

The node ordering for specifying faces should follow the right-handed rule (see [FieldView Compliance for Unstructured Data on page 420](#)). In other words, nodes should be given by walking around the perimeter of the face in a counter-clockwise manner. Hanging nodes are not permitted on boundary faces. In the example presented herein (see code in `/uns` subdirectory of the directory where **FieldView** is installed), there are 10 Standard Faces that belong to `bottom`, `top` and `wall` boundary types, 9 Standard Faces that belong to trimmed node cell and hanging node cell boundary types, and 4 Arbitrary Polygon Boundary Faces that belong to trimmed node cell and hanging node cell boundary types (the cells are shown in the [Arbitrary Polyhedron Cells on page 417](#)). The first boundary face type is `bottom`, with a rectangular face (4 nodes) consisting of nodes 1, 2, 4 and 3. The second boundary face type is `top`, with a rectangular face consisting of nodes 9, 10, 12 and 11. Eight rectangular faces type `wall` are written after that. Nine Standard Faces that belong to trimmed node cell and hanging node cell boundary types are written after that. Four Arbitrary Polygon Boundary Faces that belong to trimmed node cell and hanging node cell boundary types are written after all Standard Boundary Faces are written.

If a face is not a boundary, it is not necessary to define it in this section. The section must start with the keywords `Boundary Faces`, followed by the number of faces that will be defined, as shown below.

```
nbfaces = 10 + 13
write(iunit, *) 'Boundary Faces', nbfaces
```

Boundary Faces 23

```
write(iunit, *) 1, 4, (bot_faces(i,1), i=1,4)
```

```
1 4 1 2 4 3
```

```
write(iunit, *) 2, 4, (top_faces(i,1), i=1,4)
```

```
2 4 9 10 12 11
```

```
do 300 j = 1, 8
  write(iunit, *) 3, 4, (wall_faces(i,j), i=1,4)
```

```
300 continue
```

```
3 4 1 2 6 5
3 4 5 6 10 9
3 4 3 4 8 7
3 4 7 8 12 11
3 4 1 3 7 5
3 4 5 7 11 9
3 4 2 4 8 6
3 4 6 8 12 10
```

c First, write the triangular and quadrilateral (i.e., 3 or 4 node faces)
c that are a part of the trimmed and hanging node cell.

```

write(iunit, *) 4, 3, (trim_cell_face(i,2), i=1,3)
write(iunit, *) 4, 4, (trim_cell_face(i,5), i=1,4)
write(iunit, *) 4, 4, (trim_cell_face(i,6), i=1,4)
write(iunit, *) 4, 4, (trim_cell_face(i,7), i=1,4)
write(iunit, *) 5, 4, (hang_cell_face(i,4), i=1,4)
write(iunit, *) 5, 4, (hang_cell_face(i,5), i=1,4)

```

```

4 3 16 18 17
4 4 13 19 22 14
4 4 14 22 21 15
4 4 19 20 21 22
5 4 20 19 22 21
5 4 21 22 29 24

```

```

write(iunit, *) 4, 5, (trim_cell_face(i,1), i=1,5)
write(iunit, *) 4, 5, (trim_cell_face(i,3), i=1,5)
write(iunit, *) 4, 5, (trim_cell_face(i,4), i=1,5)

```

```

4 5 13 14 15 16 17
4 5 15 21 20 18 16
4 5 13 17 18 20 19

```

```

write(iunit, *) 5, 5, (hang_cell_face(i,1), i=1,5)
write(iunit, *) 5, 5, (hang_cell_face(i,2), i=1,5)
write(iunit, *) 5, 7, (hang_cell_face(i,3), i=1,7)
write(iunit, *) 5, 5, (hang_cell_face(i,6), i=1,5)

```

```

5 5 20 21 24 25 26
5 5 24 29 28 27 25
5 7 20 26 25 27 28 30 19
5 5 22 19 30 28 29

```

3D elements (cells) are defined in the next section. The section must begin with the keyword `Elements`.

```

write(iunit, *) 'Elements'

```

Elements

The elements are described as follows. Each element will get a type and a subtype (currently always set to one). The Standard 3D element types are: 1-tetrahedron, 2-hexahedron, 3-prism, 4-pyramid. Type 5 is assigned to Arbitrary Polyhedron 3D elements.

Standard 3D elements require between 4 and 8 node numbers (depending on the element type) to define the element. The ordering of the nodes is important. See [Standard 3D element types on page 415](#) for node numbering information. The node numbers follow the element type.

The Arbitrary Polyhedron 3D elements are described as follows. First, the element type, 5, is written. Next, the element subtype, 1 is written (at present, subtype is 1 for all elements; it is a reserved field). After that, the number of faces, the number of nodes including the center node, and the center node number are written. A negative integer number for the center node tells **FieldView** there is no center node. Next, each face of the Arbitrary Polyhedron 3D element is described. For each face, the number of face vertices, the vertex numbers, the number of hanging nodes and the hanging node numbers are written.

In the example presented herein (see code in /uns subdirectory of the directory where **FieldView** is installed) there are two hexahedral 3D elements (cells) and two arbitrary polyhedron elements: a trimmed cell element and a hanging node element (the cells are shown in [Arbitrary Polyhedron Cells on page 417](#)).

Two hexahedral cells are written as follows:

```
write(iunit, *) 2, 1
write(iunit, *) (hexes(i,1), i=1,8)
write(iunit, *) 2, 1
write(iunit, *) (hexes(i,2), i=1,8)
```

```
2 1
1 2 3 4 5 6 7 8
2 1
5 6 7 8 9 10 11 12
```

The first element is a hexahedron, consisting of nodes 1 through 8. The second element is also a hexahedron, consisting of nodes 5 through 12.

Next, for the trimmed cell element we have:

```
write(iunit, *) 5, 1
write(iunit, *) 7, 11, 23
write(iunit, *) 5, (trim_cell_face(i,1), i=1,5), 0
write(iunit, *) 3, (trim_cell_face(i,2), i=1,3), 0
write(iunit, *) 5, (trim_cell_face(i,3), i=1,5), 0
write(iunit, *) 5, (trim_cell_face(i,4), i=1,5), 0
write(iunit, *) 4, (trim_cell_face(i,5), i=1,4), 0
write(iunit, *) 4, (trim_cell_face(i,6), i=1,4), 0
write(iunit, *) 4, (trim_cell_face(i,7), i=1,4), 0
```

```
5 1
7 11 23
```

```

5 13 14 15 16 17 0
3 16 18 17 0
5 15 21 20 18 16 0
5 13 17 18 20 19 0
4 13 19 22 14 0
4 14 22 21 15 0
4 19 20 21 22 0

```

All faces for the element are assumed not to have hanging nodes at them.

For the hanging node element we have:

```

write(iunit, *) 5, 1
write(iunit, *) 6, 12, -1
write(iunit, *) 5, (hang_cell_face(i,1), i=1,5), 0
write(iunit, *) 5, (hang_cell_face(i,2), i=1,5), 0
write(iunit, *) 7, (hang_cell_face(i,3), i=1,7), 1, 31
write(iunit, *) 4, (hang_cell_face(i,4), i=1,4), 0
write(iunit, *) 4, (hang_cell_face(i,5), i=1,4), 0
write(iunit, *) 5, (hang_cell_face(i,6), i=1,5), 0

```

```

5 1
6 12 -1
5 20 21 24 25 26 0
5 24 29 28 27 25 0
7 20 26 25 27 28 30 19 1 31
4 20 19 22 21 0
4 21 22 29 24 0
5 22 19 30 28 29 0

```

All faces for the element except face 3 are assumed not to have hanging nodes at them. Face 3 has one hanging node. Node number for the hanging node is 31.



Note: Keeping arbitrary polyhedron elements together is recommended but not required. The arbitrary polyhedron elements can appear before, after, or in-between standard 3D elements (tetrahedron, pyramid, prism, hexahedron) sections.

Next, the results variables for each node are listed in the same order as the nodes were defined. The variables must be in the same order as in the Variable Names section. One needs to write out all of the results, in node order, for variable 1 (in this case `pressure`), then all of the results, in node order, for variable 2 (in this case `u-velocity`), etc. All of the data for the first variable is output before any of the data for the second variable. The total number of real numbers should match the number of nodes times the number of variables.

The section must begin with the keyword `Variables`. You should skip this section if the number of variables is zero.

```

    if (nvars .gt. 0) then
        write(iunit, *) 'Variables'
        do 500 j = 1, nvars
            do 600 i = 1, nnodes
                write(iunit, *) vars(i,j)
600          continue
500        continue
    endif

```

Variables

1.0 0.1 1.2 0.1

1.1 0.2 1.1 0.2

...

1.18 1.18 1.18 1.18

(124 real numbers - 4 variables for 31 nodes)

Next, output the face data (boundary variable data) for this grid. Note that all of the data for the first variable is output before any of the data for the second variable. Remember that the Boundary Table above has a surface results flag indicating which boundary types have surface results. The data should be written in the same order as the faces in the Boundary Faces section, skipping over faces whose boundary type has a surface results flag of zero (false). For each variable, you should write one number per boundary face. The section must begin with the keyword `Boundary Variables`. You should skip this section if the number of boundary variables is zero.

```

    if (nbvars .gt. 0) then
        write(iunit, *) 'Boundary Variables'
        do 700 j = 1, nbvars
c   The data for the bottom face is written first for each variable, because
c   the bottom face was written first in the "Boundary Faces" section.
c   Write data for variable#j for the "bottom" face:
            write(iunit, *) bot_bvars(j)
c   Write data for variable#j for the "top" face:
            write(iunit, *) top_bvars(j)
c   Skip the "wall" faces, because the surface results flag for the wall
c   boundary type was 0 (false) in the Boundary Table section.
c   Arbitrary Polyhedron boundary variables:
            write(iunit, *) (trim_cell_bvars(j,i), i=1,7)
            write(iunit, *) (hang_cell_bvars(j,i), i=1,6)
700        continue
    endif

```

Boundary Variables

1.0

```

1.0
1.0 1.1 1.2 1.3 1.4 1.5 1.6
1.1 1.11 1.12 1.13 1.14 1.15
4.5
2.0
1.7 1.8 1.9 1.1 1.11 1.12 1.13
1.16 1.17 1.18 1.19 1.2 1.21
3.0
4.0
1.14 1.15 1.16 1.17 1.18 1.19 1.2
1.22 1.23 1.24 1.25 1.26 1.27
3.0
2.5
1.21 1.22 1.23 1.24 1.25 1.26 1.27
1.28 1.29 1.30 1.31 1.32 1.33

```

Closing of the file:

```
close(iunit)
```

Important Notes:

1. All of the keywords shown above must be entered exactly as they appear in this document. They are, however, case-insensitive.
2. Except where explicitly stated, every section described above must appear in the data file, even if the capabilities provided by the section are not used. For example, if no Boundary Surfaces are to be defined, both the Boundary Table and the Boundary Faces sections must still appear, with the number of types and faces set to zero.

The following is a sample file and description of a simple 4 hex element mesh. Note that it is not required that values such as "Constants" appear all on one line. The format shown below is valid, but less compact. It has been shown this way for ease of readability.

FIELDVIEW 3 0	Header Line
Constants	
0.0	Floating point number for TIME
0.2	Floating point number for FSMACH
15.0	Floating point number for ALPHA
1.6e06	Floating point number for RE
Grids	
1	Number of grids
Boundary Table	
3	Number of boundaries
0 0 1 inlet	Boundary 1 is of type 0 (does not block flow),

no face data present and it has the correct clockness.
It is called "inlet".

0 0 0 outlet

Boundary 2 is of type 0 (does not block flow),
no face data present and it has the correct clockness.
It is called "outlet".

1 1 1 wall boundary

Boundary 3 is of type 1 (a wall, does block flow),
face data is present and it has the correct clockness.
It is called "wall boundary".

Variable Names
2
pressure
temperature

Number of variables
Names of variables

Boundary Variable Names
2
pressure
temperature

Number of variables
Names of variables

Nodes
18
0.0 0.0 0.0
0.5 0.0 0.0
1.0 0.0 0.0
0.0 0.5 0.0
0.5 0.5 0.0
1.0 0.5 0.0
0.0 1.0 0.0
0.5 1.0 0.0
1.0 1.0 0.0
0.0 0.0 1.0
0.5 0.0 1.0
1.0 0.0 1.0
0.0 0.5 1.0
0.5 0.5 1.0
1.0 0.5 1.0
0.0 1.0 1.0
0.5 1.0 1.0
1.0 1.0 1.0

Number of nodes
X Y Z Coordinates of node 1
X Y Z Coordinates of node 2
X Y Z Coordinates of node 3

Boundary Faces
3
1 4 1 4 5 2

Number of boundary faces
Definition of boundary face (boundary type 1,
contains 4 nodes, node numbers 1, 4, 5, 2)

```

2 4 14 17 18 15
3 4 13 16 17 14

```

Elements

```

2 1 1 2 10 11 4 5 13 14

```

```

2 1 2 3 11 12 5 6 14 15
2 1 4 5 13 14 7 8 16 17
2 1 5 6 14 15 8 9 17 18

```

Variables

```

0.0 0.1 0.2 0.1 0.2 0.3
0.2 0.3 0.4 1.0 1.1 1.2
1.1 1.2 1.3 1.2 1.3 1.4
100 200 300 100 200 300
200 300 200 200 350 400
355 405 500 100 200 300

```

Boundary Variables

```

0.0157 323

```

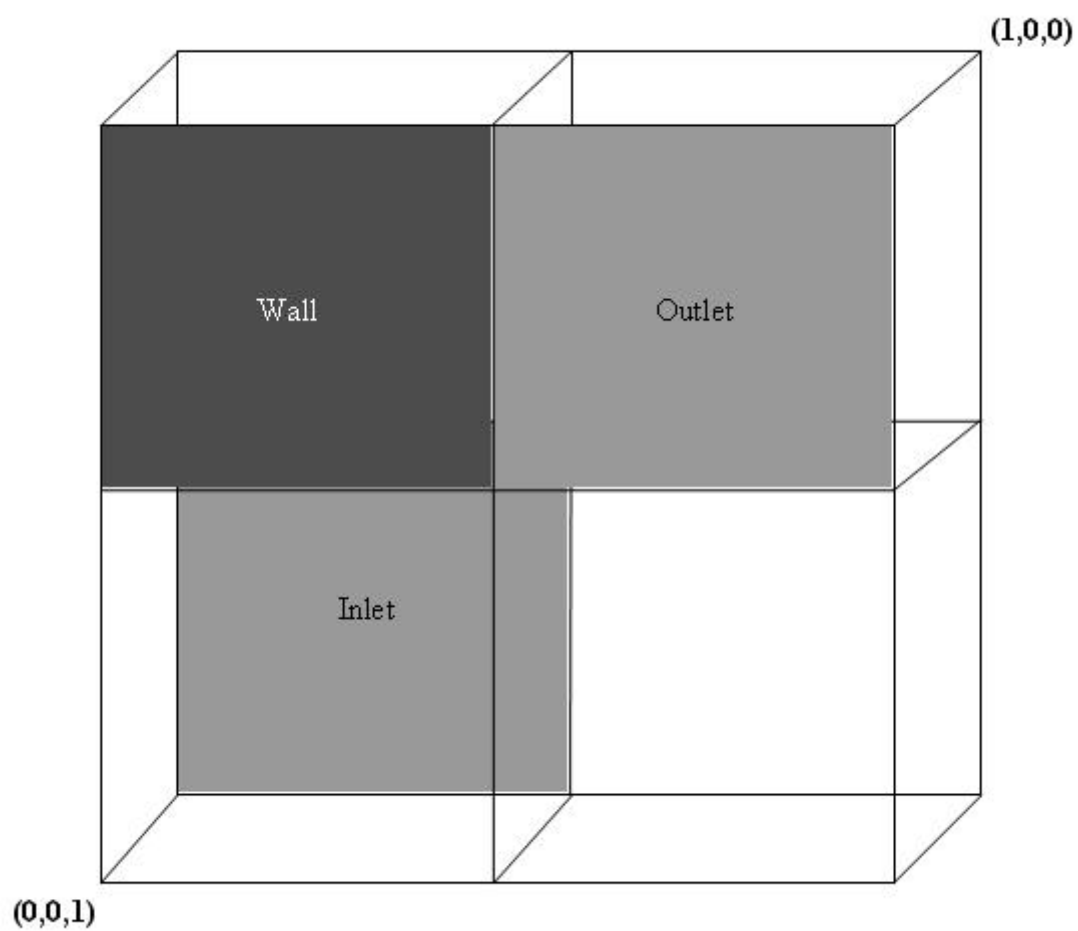
Element definition (hex element (type = 2),
subtype 1, nodes 1, 2, 10, 11, 4, 5, 13, 14)

Pressure values (in node order)

Temperature values (in node order)

Only "wall" has face data and "wall" has only one face. There is therefore only a single Boundary Variable value for this face for each of the two normal variables, pressure and temperature.

The file shown above is used to define the dataset shown in **Figure 155**.

**Figure 155 Example Unstructured Dataset**

Unstructured Data Input panel

To read the **FieldView** unstructured file, choose File -> Data Input -> FV-UNS. The following panel will be invoked:

To read a Grid File in split format or a Grid & Results file in combined format, click Read Grid or Combined Data.

After reading of the Grid file is completed, you may click Read Results Data to read a Results file in split format.

When the data is read in, it may either Replace the data currently in memory, or be Appended to the current data.

To read only the data at the boundaries of the dataset, turn this option on.

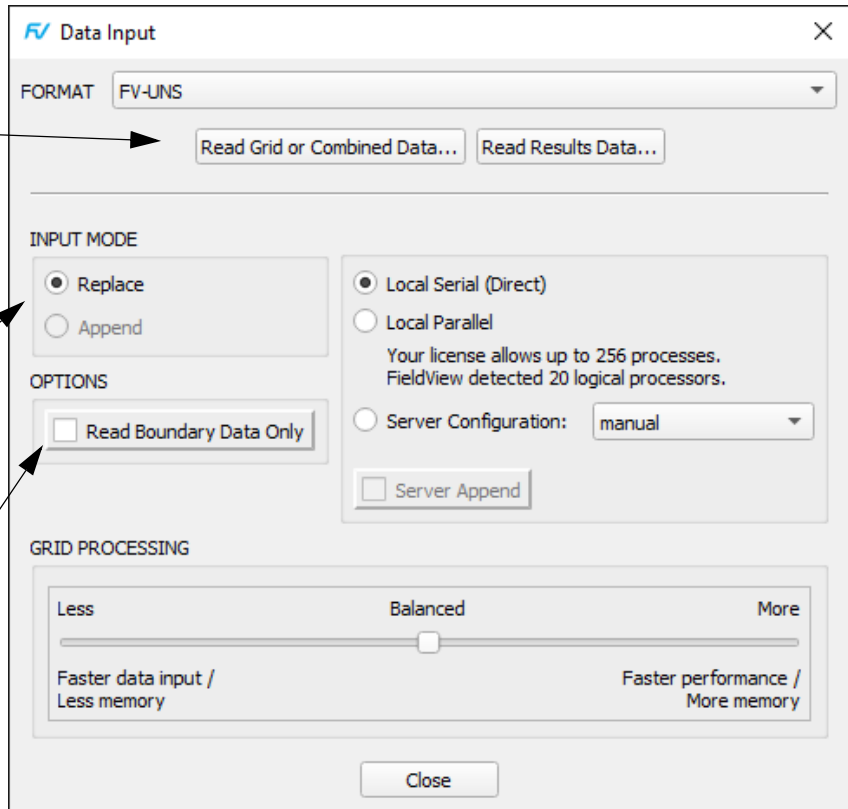


Figure 156 Unstructured Data Input Panel

Transient Data

FieldView supports transient FV-UNS datasets. Transient FV-UNS data requires one file per time-step. This is different from other solvers (FLOW-3D®, for example), where all data resides in a single file. One advantage of this technique is that it is far easier to access a specific time-step during visualization.

A dataset will automatically be recognized as transient if each file has a time step number embedded in its name using a convention described in [Transient Data](#).

If any one of the files of the series is chosen, **FieldView** will find other files with the same file naming convention in the directory and present you with the option of treating the set as transient. If agreed to, the chosen time-step will be loaded into memory, and the remaining filenames stored for reference. Other time-steps can be accessed through the Transient Data Controls panel (see [Transient Data Controls](#) for more details).

Creating FV-UNS files with FORTRAN 77 and C for different OS

General remarks on creating FV-UNS files with FORTRAN 77 and C for different operating systems. CFD data can be read into **FieldView** using files in **FieldView**-Unstructured (FV-UNS) data format. FV-UNS files may be created with FORTRAN 77 and C programming languages. This section describes compiler and operating system related details of the creation process.

ASCII (formatted) data files can be created on any platform and can be read by **FieldView** on any supported platform. The statement is also true for binary (*.bin) files. They can be created and read by **FieldView** on any supported platform. Writing binary **FieldView**-Unstructured files can be accomplished using C. When using FORTRAN 77 however, the data files may either be binary (*.bin) or unformatted (*.unf) depending upon the platform and FORTRAN 77 compiler options. Unformatted **FieldView**-Unstructured files are *not* platform independent. However, FV-UNS unformatted files created as described below may be read by **FieldView** on any supported platform. (Unformatted files created with Microsoft FORTRAN, or Digital Visual Fortran cannot be read by **FieldView**. For these compilers you must write binary files.)

The following table shows what type of files can be created with FORTRAN on what **FieldView** platform:

SGI	unformatted (*.unf) / binary (*.bin)
HP/SUN/IBM	unformatted (*.unf)
PC w/Absoft Fortran (Windows or Linux)	unformatted (*.unf)
PC w/Microsoft or Digital Visual Fortran*	binary (*.bin)
PC w/Lahey Fortran*	unformatted (*.unf)

* Unsupported. For a table of supported compilers, see the Discussion section below.

The sample C code works on all platforms without any modifications needed. The sample FORTRAN 77 code may need modifications. This is detailed below.

For each unique platform, separate instructions are given to use or modify the existing sample FORTRAN 77 code provided in the **FieldView** installation. Since the FORTRAN compilers on the HP, SUN and IBM platforms provide the same functionality, these have been grouped together. When compiler options are required, these are also provided.

ASCII FV-UNS files

ASCII (formatted) files can be created on any platform and read on any platform. However, these files will be larger and slower to read in and will require more dynamic memory. Sample codes for creating ASCII FV-UNS files can be found in the /uns subdirectory of the **FieldView** installation. The files are:

write_split_ascii_uns.f	sample FORTRAN code for creating split ASCII FV-UNS files (grid file & results file)
-------------------------	--

<code>write_ascii_uns.f</code>	sample FORTRAN code for creating a combined ASCII FV-UNS file
--------------------------------	---

Binary FV-UNS files

Sample codes for creating binary FV-UNS files can be found in the `/uns` subdirectory of the **FieldView** installation. The files are:

<code>write_split_binary_uns.c</code>	sample C code for creating split binary FV-UNS files (grid file & results file)
<code>write_binary_uns.c</code>	sample C code for creating a combined binary FV-UNS file

It is possible to write binary files from FORTRAN 77 on only two platforms: SGI and PC (Windows). To do this you must modify the sample FORTRAN code for writing unformatted FV-UNS files described in [Unformatted \(FORTRAN 77\) Format](#).

Source changes to the FORTRAN `OPEN` statement are given in the following table:

Platform	Source Changes Needed	Compiler Switches
SGI	FORM='UNFORMATTED' to FORM='SYSTEM' in the <code>OPEN</code> statement	none
PC w/Microsoft or Digital Visual FORTRAN*	FORM='UNFORMATTED' to FORM='BINARY' in the <code>OPEN</code> statement	none

* Unsupported. For a table of supported compilers, see the Discussion section below.

Note: Binary files cannot be created in FORTRAN 77 on the following systems: HP, SUN and PC (Windows or Linux) with Absoft FORTRAN.

UNFORMATTED FV-UNS files

Sample codes for creating unformatted FV-UNS files can be found in the `/uns` subdirectory of the **FieldView** installation. The files are:

<code>write_split_unformatted_uns.F</code>	sample FORTRAN code for creating split unformatted FV-UNS files (grid file & results file)
<code>write_unformatted_uns.F</code>	sample FORTRAN code for creating a combined unformatted FV-UNS file

Changes that may be necessary to the source code, and compiler switches that may need to be used to create unformatted files, are shown in the following table:

Platform	Source Changes Needed	Compiler Switches
SGI	None	none
HP/SUN/IBM	None	none
PC w/Absoft FORTRAN (Windows or Linux)	None	-N3
PC w/Lahey FORTRAN*	Add ACCESS= 'TRANSPARENT' in the OPEN statement	none

* Unsupported. For a table of supported compilers, see the Discussion section below.

Note: Unformatted files created with Microsoft FORTRAN or Digital Visual Fortran cannot be read by **FieldView**. For these compilers you must write binary files.

Discussion

The information presented above is valid for the following supported compilers:

Platform	C	Fortran 77
SGI	7.2	7.2
SGI (64-bit)	7.3	7.3
IBM	3.1.4.0	4.1.0.0
Sun	5.0	5.0
HP	G.10.32.05	B.10.20.09
PC w/Absoft	-	6.2 (Windows) or 7.0 (Linux)

The information presented in this document is valid for compiler versions given in the above table. For compilers different from those shown in the above table, the sample code and the information may not be correct.

It is possible to write binary files from FORTRAN on only two platforms: SGI and PC (Windows).

SGI

On the SGI, it is easiest to open the file with `FORM='SYSTEM'` (not `BINARY`). You then use unformatted write statements to write to the file.

`FORM='BINARY'` can be used, but then you have to use formatted write statements that use the character '(A)' format.

HP/SUN/IBM

These platforms do not support binary format output from FORTRAN 77, only C.

PC w/ Absoft FORTRAN

Absoft FORTRAN under Windows, or Linux, does not support binary format output from FORTRAN.

To write **FieldView**-compatible unformatted files using this compiler, you need to compile with the `"-N3"` option. The files should be opened and written in the usual way for unformatted files - no source code changes are needed.

Microsoft and Digital Visual FORTRAN (unsupported)

In Microsoft FORTRAN PowerStation 32 Version 1.0 and Digital Visual FORTRAN Versions 5.0 and 6.0, the file is opened with `FORM='BINARY'`. You then use unformatted write statements to write to the file.

Note: In these compilers, files written with `FORM='UNFORMATTED'` have a completely different format from Unix unformatted files. They cannot be read by **FieldView**, even on machines with the same byte order.

Lahey FORTRAN (unsupported)

Lahey FORTRAN does not support binary format output from FORTRAN 77. To write **FieldView** compatible unformatted files using this compiler, use `FORMAT='UNFORMATTED', ACCESS='TRANSPARENT'` in the `OPEN` statement prior to writing the file.

2D FV-UNS FILES

There are two methods that can be use to write 2D FV-UNS files.

Method 1

Since **FieldView** really only handles 3D data well, it is best to introduce some thickness to make it a skinny 3D problem.

Here are the steps you will need to accomplish this to write FV-UNS ASCII files:

1. Assign z values of zero to the original 2D nodes.
2. Double the number of nodes by appending a second copy of the array of nodes to itself. Assign z values of $1e-5$ or greater to the second copy of the nodes.
3. Use the nodes with $z == 0$ and the corresponding nodes with $z \geq 1e-5$ to create elements with a very slight depth to them. Use prisms (wedges) for triangles and hexahedrons for rectangles. You should be able to arrange things so that the node indices you use for one end of 3D elements are simply the indices for a triangle or rectangle. The other end would be the same indices plus the original number nodes - the offset to the second copy of the original nodes.
4. Any vector result that does not supply 3 variables must pad the vars array with zeros where the missing variable would otherwise be. This applies to 3D problems as well. If only U velocity is present in results, both V and W velocity need to be padded in the array we pass to **FieldView** if, and only if, it is specified to be part of a vector function. The variable must also be present in the table of variable names at the top of the file.
5. Double the number of variables by appending a second copy of the array of variables to itself. This is required since we doubled the nodes.

Method 2

Create a file of 3 and/or 4 noded faces. FV-UNS format requires at least one element. If you specify a single dummy element you can then specify as many faces as you want. The downside is that the only useful visualization of these faces would be as **FieldView** boundary surfaces. You could not probe these faces for results values or create coordinate or iso surfaces from this data. If you only require this data to show your model geometry this is a valid approach.

Appendix E Colormap File Format

The eight default colormap button labels and their corresponding colormap file names are listed below. These colormaps are accessed from the Colormap Specification panel.

Button Label	File Name
Spectrum	fv/data/colormaps/c1.col
Gray Scale	fv/data/colormaps/c2.col
Zebra	fv/data/colormaps/c3.col
Striped	fv/data/colormaps/c4.col
Color Striped	fv/data/colormaps/c5.col
Black & White	fv/data/colormaps/c6.col
NASA-1	fv/data/colormaps/c7.col
NASA-2	fv/data/colormaps/c8.col

File Naming Convention

The general form for the colormap filename is shown below:

```
file_name.col
```

The first part of the filename may contain up to 255 alpha characters, numbers, or underscores (Note: spaces are not permitted).



Note: [User Defined Colormaps on page 74](#) of Working With FieldView for placement of .col files so that they are available on the GUI. Also, Colormap file names (.col) are stored in Colormap (.map) Restart Files which can be used as part of a Preference Restart. See [Preference Restart on page 12](#) of the **User's Guide** for information on where these restarts can be placed to load **automatically**.)

The colormap file is an ASCII text file which has the following format:

```

Name
N (number of entries)
red_1      green_1      blue_1
red_2      green_2      blue_2
red_3      green_3      blue_3
...
red_N-2    green_N-2    blue_N-2
red_N-1    green_N-1    blue_N-1
red_N      green_N      blue_N

```



Note: **FieldView** requires at least two entries. If only one color is desired, then enter the same line twice.

The Name Field

The `Name` field represents one line of ASCII text which appears on the first line of each colormap file. Although the application currently ignores this line of text (i.e. it will *not* show up in the **FieldView** interface), it is intended to indicate the name of the colormap. Some example names are Spectrum, Gray Scale, Zebra, or Striped. Any name with up to 256 characters can be specified.

The N Field

The `N` field represents an ASCII integer which appears on the second line of each colormap file. This integer must be greater than or equal to 2 (two). See the notes on Colormap File Compression and Expansion for a detailed explanation of how colormap files are read by the application.

The Red Green Blue Fields

Each line of text after the integer `N` represents an RGB color triplet. Each element of a triplet is an ASCII floating point string in the range of 0.000000 to 1.000000. The number of lines of RGB triplets must match the value of `N` in the colormap file. For example, the following six RGB triplets represent the colors White, Black, Red, Green, Blue, and 50% Gray, respectively:

1.000000	1.000000	1.000000	(White)
0.000000	0.000000	0.000000	(Black)
1.000000	0.000000	0.000000	(Red)
0.000000	1.000000	0.000000	(Green)
0.000000	0.000000	1.000000	(Blue)
0.500000	0.500000	0.500000	(50% Gray)

Note: The rightmost column is *not* part of the file format, but is shown for guidance only.

Limitations:

Out of range RGB values

RGB values must lie between 0.000000 to 1.000000. Values specified outside this range will be silently clamped to either the lower or upper range limits.

Colormap File Compression and Expansion

The number of scalar colors which can be displayed is 100. This is the Internal Scalar Colormap Size.

Compression

If the size of a colormap file is greater than the Internal Scalar Colormap Size, then colormap file is linearly sampled to fit the Internal Scalar Colormap Size.

Expansion

If the size of a colormap file is less than the Internal Scalar Colormap Size, then the colormap file is evenly expanded to fit the Internal Scalar Colormap Size. Note that colors from the colormap file are duplicated to fit the new size and not interpolated.

Note that the first and last colormap entries always appear as the first and last colors in the Internal Scalar Colormap, respectively.

Colormap Expansion Example

If you specify only three colors such as red, white, and blue, and if the Internal Scalar Colormap Size is 100, then the colormap file would be expanded. Internal colors 1 through 33 would be red, colors 34 through 66 would be white, and colors 67 through 100 would be blue.

Colormap Compression Example

If the Internal Scalar Colormap Size is 100, and the colormap file has 1000 entries, then the internal colormap would be set to the values which are linearly sampled from the colormap file.

Note that the first and last entries of the Internal Scalar Colormap are always set to the first and last entries of the colormap file, respectively.

Appendix F FieldView Limits

The following limits apply:

Per Session

Input data / Maximum number of Grids:

- 1,000,000 grids: **HPC FieldView** (Linux only)
- 100,000 grids: Linux and Mac Client & Server
- 50,000 grids: Windows Client & Server

Number of partition files in PFPR layout file: Limited to 1,342 per FieldView “worker” process. Note also that all Layout files in a potential time series must specify the same number of partitions.

Maximum number of Datasets: 3000

Maximum number of Regions:

Bounded by the Maximum number of Grids (see above)

Maximum number of Annotation Objects (Text & Arrows): 100

Maximum Image Size when running batch mode with software rendering:

- with Anti-aliasing off: 8192 X 8192
- with Anti-aliasing on: 4096 X 4096

Maximum number of registered User-defined Functions: 500

Maximum number of bytes in a User-defined Function name: 119

Due to support for international fonts, each character may need more than 1 byte.

Maximum number of zoom levels: 10

This is the zoom initiated with View > ZoomBox, or its associated Toolbar button on the “Side Toolbar”

Maximum number of bytes in a Filename (including full pathname): 255

Per Dataset

Maximum number of bytes for a variable name: 80

Maximum number of Scalar variables: 2048

PLOT3D Q variables don't count.

Maximum number of Vector variables: 400

PLOT3D Q variables don't count.

Maximum number of Boundary Face Scalar variables: 1000

Maximum number of Boundary Face Vector variables: 1000

Maximum number of formulas: 2999

Maximum number of Boundary Types: 20,000

Maximum number of bytes for a Boundary Type name: 80

Maximum number of bytes for a region name: 80

Maximum number of transient time steps: 100,000

When using PFPR, the number of time steps times the number of partition files cannot exceed one billion.

Maximum number of Dynamic Clip Groups at a time: 12

Each Clip Group is composed of up to 16 lines, 4 boxes, or a combination (each box counts as 4 lines).

Per Grid

Maximum number of Nodes ; Maximum number of Elements: 2,147,483,647

This is the maximum value of a (signed 32-bit) integer in Fortran and C. Other restrictions from the hardware and software environment may also apply.

If you do not use the Intel Fortran compiler, the maximum size for FORTRAN UNFORMATTED records is 2GB. This imposes a limit of approximately 666 million nodes or elements per grid (2 billion divided by 3).

By Object

Legends

Maximum number of legend labels: 52

Surfaces

Number of contour lines: 500

Number of Filled Contours: 100

Streamlines

Maximum number of Seeds per Streamline Rake: 2000

Maximum number of Streamline Steps: 10,000 (5,000 in each direction if 'Both')

Maximum ribbon width: 1024

Maximum number of Animation Steps for display types Growing, Spheres & Lines, Spheres, and Dots: 1000

Maximum number of Filaments, Spheres, Arrows, or Dots on a single streamline: 1000

Applies to display types:

- Filament
- Filament & Spheres
- Filament & Arrows
- Lines of Spheres
- Lines of Dots

Annotation

Maximum number of [bytes](#) in a Text object: 512

2D Plots

Maximum number of plots per dataset: 9

Maximum number of paths per plot: 10

Maximum number of tick marks: 500

Arbitrary Polyhedra

Maximum number of faces for an arbitrary polyhedron cell: 256

Maximum number of vertices for a single face of an arbitrary polyhedron cell (also maximum number of vertices for an arbitrary polyhedron boundary face): 256

Appendix G 2D Plot Format

This format is used for both the Point Probe *output* and the 2D Plot export/import options. The file for each plot/probe contains a two line header. The keyword `VARIABLES`, on the first line, is followed by the names of the functions used in the plot or shown by the probe. The keyword `ZONE`, on the second line, delimits a contiguous region of points and function values. If more than one plot is exported, each plot will be separated by a `ZONE` section. The required `J=1` indicates the data format being used. The optional section `T=` is used for the annotation of the legend. Any text following the `T=` keyword and enclosed by double quotes will be displayed in the legend field on the plot.

Following the two line header are the values of `x`, `y`, and `z`, and values for as many functions as are defined on the `VARIABLES` line. The syntax of this format is as follows:

```
VARIABLES = "X", "Y", "Z", "Function 1" [, ... "Function n"]
ZONE J=1 [, T="Legend"]
```

For 2D Plot, the format is:

```
x1 y1 z1 f1 [ f2 ]
x2 y2 z2 f1 [ f2 ]
...
xn yn zn f1 [ f2 ]
```

where `f2` is the threshold function.

For Point Probe, the format is:

```
x1 y1 z1 f1 [ f2 f3 ... f6 ]
x2 y2 z2 f1 [ f2 f3 ... f6 ]
...
xn yn zn f1 [ f2 f3 ... f6 ]
```

where `f1 - f6` are the scalar, threshold, vec1, vec2, vec3 and iso functions.



Note: The 2D Plot GUI ignores ["Function 3", ... "Function n"]. "Function 2" data can be displayed by showing 'right axis' (the same way as showing the threshold function value).

Cylindrical Note: When cylindrical coordinates are specified using an FVREG file (Region definition), the input is assumed to be in RTZ coordinates and the output values and labels are also RTZ (Radius, Theta, Z). See [Chapter 3](#) for more information.

Appendix H Structured Boundary Files

A Structured Boundary Surface file is used to group several computational surfaces by name. This allows you to use the Boundary Surface panel for structured grids, and create groups of named entities as one surface. Making changes to these surfaces or using them for plotting or streamline seeding should be much easier.

The Structured Boundary Surface file must have the same name as the grid file, with a `.fvbnd` extension. For example, if your grid file is named:

```
test.bin
```

The Structured Boundary file must be named:

```
test.bin.fvbnd
```

The file is read automatically when the grid file is read. Afterwards, the boundaries will be accessible on the Boundary Surface panel. A structured boundary file for the `bluntdfin` dataset has been included in the `examples` directory of the **FieldView** installation and is called `bluntdfinx.bin.fvbnd`.



Note: **FieldView** will only look for all upper or all lower case suffix names. Mixed case suffixes will not be seen. That is, `test.bin.Fvbnd` is an invalid Structured Boundary file name.

Transient PLOT3D

If a Structured Boundary file is used for transient PLOT3D files, then there are two valid file naming conventions. The easiest option is to have one global FVBND file for the entire transient sequence. This file would take the root name of the grid or grid/results file without any embedded time step number. The other option is to use one FVBND file for each grid or grid/results file, using the same naming convention. These two options are illustrated in the following example for the case of transient PLOT3D data:

Grid File

```
duct_0010.g.bin
duct_0020.g.bin
duct_0030.g.bin
...
duct_1080.g.bin
```

Separate FVBND Files

```
duct_0010.g.bin.fvbnd
duct_0020.g.bin.fvbnd
duct_0030.g.bin.fvbnd
...
duct_1080.g.bin.fvbnd
```

Global FVBND File

```
duct_.g.bin.fvbnd
```



Important Note: Although the “one file per time step” naming convention is allowed, every FVBND file must be identical.

Face Data and Surface Normal Information

The format for the Structured Boundary file has been enhanced for use with Face Data (see [Chapter 1](#) and later in this appendix for more information on Face Data with PLOT3D formats). This format is specifically for Face Data boundary definitions, but can be used even without face data results. It is the default format used when the Create Wall and Create Exterior tools are used. Previous formats of the Structured Boundary file are still supported. However, if the Create Wall or Create Exterior tools are used when an older FVBND file already exists, a file in the newer format described herein will be created.

The format also allows the specification of boundary surface normal directions. This information, if present, is used by **FieldView** to calculate normal-based boundary surface integrals. This is independent of the presence or absence of face data. Face data variables, whether scalar or vector, are not available for use with the CFD Calculator on the Function Formula Specification panel.

CFX-4

This file allows the Boundary Surface panel to display CFX-4 defined boundaries. CFX-4 Structured Boundary files use the older FVBND 1 3 format described in this appendix. This reader automatically creates structured boundary files in the same directory in which the data files are found. They have the same base file name as the data file (*.dmp) with an additional .fvbnd extension. So, if the data file is called tmp.dmp then the corresponding structured boundary file will be named tmp.dmp.fvbnd. Note that if the user does not have write access to this directory **FieldView** will issue a message:

```
Failed to write structured boundary file
```

and the file will not be generated. In addition, if an older .fvbnd file (with the same name) already exists and the user does not have file permission to overwrite it, the boundary information may not be correct and the file will not be read in if the grid information found in this older .fvbnd file is different from that for the current grid.

NPARC/WIND and WIND US

This file allows the Boundary Surface panel to display NPARC/WIND and WIND Structured Common File defined boundaries. Structured Boundary files use the newer format described below. Structured boundary files are automatically created when using this reader. They are written in the same directory in which the data files are found. They have the same file name as the grid file (.cgd) or the combined file (.cgf) but with an additional extension of .fvbnd. Therefore, if the data file is called

`tmp.cgd` then the corresponding structured boundary file will be named `tmp.cgd.fvbnd`. Note that if the user does not have write access to this directory **FieldView** will issue a message:

```
Failed to write structured boundary file
```

and the file will not be generated. In addition, if an older `.fvbnd` file (with the same name) already exists and the user does not have file permission to overwrite it, the boundary information may be incorrect and will not be read in if the grid information found in this older `.fvbnd` file is different than that for the current grid.

If the user wishes to disable the writing of the structured boundary file, the environment variable `FV_NO_BNDRY_FILE` needs to be set (to any value).



Note: The NPARC/WIND reader creates structured boundaries only at `I=1`, `I=MAX`, `J=1`, `J=MAX`, `K=1`, `K=MAX` of each zone, and only if they are “full face boundaries”. No interior boundaries will be created.

Create Wall Bnd File

This tool will create a Structured Boundary file with a boundary type `wall` for all no slip walls, and boundary types `"wall-grid#"` per grid (if the dataset is single grid the per grid boundary names are omitted). These boundaries will specify the exteriors of each grid where velocity is zero. Complete details on this tool can be found in [Chapter 14](#) of **Working with FieldView**. Note that this feature cannot be used if your data has been read with Parallel FieldView, unless read in Partitioned File (PFPR) format. If tried, FieldView will issue a corresponding message.

Create Exterior Bnd File

This tool will create a file named `gridfilename.fvbnd` with boundary types `exterior-grid#` (if the dataset is single grid the grid number is omitted from the boundary name). These boundaries will specify the exteriors of each grid. Once created, they will then be available with the Boundary Surface panel (see [Chapter 10](#) of **Working with FieldView** for details). Note that this feature cannot be used if your data has been read with Parallel FieldView, unless read in Partitioned File (PFPR) format. If tried, FieldView will issue a corresponding message.

File Format

The format of the file is described below. This simple file format can be used to create your own Structured Boundary file for use with a structured dataset. This is also the format that is used by the Create Wall and Create External Boundary surface tools (see [Chapter 14](#) of **Working with FieldView** for more information about these tools). The automatic file creation by the CFX-4 reader uses the older FVBND 1 3 format, described in this appendix. The FVBND 1 3 file format differs from the FVBND 1 4

file format, described here, by the absence of the `results_flag` and `normal_dir` parameters (see below).

Line 1: Header - This line must exactly as shown below:

```
FVBND 1 4
```

The next few lines list the boundary types. These are the names of the boundaries as they will appear in the Boundary Surface panel. (Note that the names cannot begin with a number.) For example:

```
wall
inlet
symmetry
```

After the boundary types, a keyword must appear exactly as shown below to indicate the start of the boundaries section:

```
BOUNDARIES
```

Next, the actual boundaries are written. Each line defines a boundary 'patch' and has the following information:

```
type grid-number I-min I-max J-min J-max K-min K-max results_flag normal_dir
```

where

<code>type</code>	Boundary type number from the table above (e.g. "1" for type "wall").
<code>results_flag</code>	T (or t) if there exists a boundary results file (Function file) with face data for this patch, and F (or f) if there is no face data.
<code>normal_dir</code>	Integer value which is: positive (surface normal points towards increasing IJK), negative (normal points towards decreasing IJK), or 0 (normal direction is unknown or unspecified).

If `I-min` is set equal to `I-max`, then the surface is assumed to be an `I` surface equal to that value. This is also true for `J` and `K`. If none of these are equal, an error will be displayed when the file is read. A wildcard ("\$") may also be used to indicate that the value should be set to the maximum for that grid.

The direction of the surface normal is computed as follows. If the `IJK` coordinate system is right-handed, then the surface normal points towards increasing `IJK`. In other words, it points in the direction of increasing `I` for an `I` surface, the direction of increasing `J` for a `J` surface, and the direction of increasing `K` for a `K` surface. If the `IJK` coordinate system is left-handed, then the surface normal points towards decreasing `IJK`.

To determine whether the `IJK` coordinate system is right-handed, one needs to compute the cross-product of a vector pointing in the direction of increasing `I` and a vector pointing in the direction of increasing `J`. If the direction of the vector obtained as the cross-product is the same as the direction of the vector pointing in the direction of increasing `K`, then the `IJK` system is right-handed. If the direction of the vector obtained as the cross-product is opposite to the direction of the vector pointing in the direction of increasing `K`, then the `IJK` system is left-handed. The right-hand rule may be used

to determine the direction of the cross-product result. A right-handed IJK system is shown in **Figure 157** below.

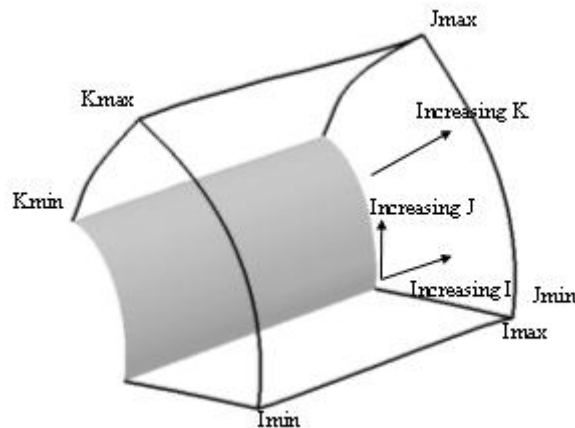


Figure 157 Right-handed IJK system



Note: If you have two results files (solver runs) for the same grid, one with face data and one without, then two different FVBND files will exist. However, both FVBND files need to have the same filename. Therefore, they must be saved under different names and renamed when needed. (Ex.: grid file - `case4.xyz`, face data FVBND file - `case4_face.xyz.fvbnd`, no-face data FVBND file - `case4_noface.xyz.fvbnd`. When you want one or the other to be used when reading in the grid, you must rename that one to `case4.xyz.fvbnd`).

A sample file is shown below:

```
FVBND 1 4
wall
inlet
outlet
BOUNDARIES
1 1 1 20 1 30 1 1 T 1

1 1 1 20 1 1 1 35 F 0
2 3 1 1 2 5 3 10 F 0
3 1 1 1 1 1 1 20 F 0
1 2 1 $ 1 $ 1 1 F 0
```

This defines a part of the boundary called "wall" on grid 1 where $K=1$ for $I=1$ to 20 and $J=1$ to 30.

Face data exists and the normal direction is in the direction of increasing IJK .

Another part of the wall on grid 1, where $J=1$ for $I=1$ to 20 and $K=1$ to 35. No face data, no normal direction.

A part of the inlet boundary on grid 3, where $I=1$, $J=2$ to 5 and $K=3$ to 10. No face data, no normal direction.

A part of the outlet on grid 1, where $I=1$, $J=1$ and $K=1$ to 20

A wall on grid 2, where $K=1$ for all I and J . No face data, no normal direction.

Note that boundary names in structured boundary files must start with an upper or lower case character. Examples of valid names are:

```
left-1
RIGHT_2
bottom 3
l e f t-4
right-2
```

Examples of invalid names are:

```
1 top
_top
```

Face Data for PLOT3D Data

FieldView supports face-based results on boundary surfaces of PLOT3D data. In order to provide face results for a 3D dataset, three additional files will need to be created. The first is a special form of the Structured Boundary file (*.fvbnd) which communicates the boundary surface definitions (described earlier in this appendix), the surface normal directions, and whether there are face results for each of the boundary surfaces. The second is a 2D Function File, which contains the face results for those boundary surfaces that have them and the third is a Function Name file which communicates the names of the face result variables to **FieldView**.

Face Data and Function Files

FieldView supports face-based results on boundary surfaces of PLOT3D data. In order to provide face results for a PLOT3D dataset, one of the additional files needed is a standard 2D Function File which contains the face results for those boundary surfaces that have them. That is, the face data file for a 3D dataset is a 2D, not a 3D, file. The Function file should have the same file name as the results file plus an additional extension: *.fvstrf.

We will use the term "boundary patch" for each definition line in the Structured Boundary file (*.fvbnd). Each such line is a computational surface.



Note: Every boundary patch must have the same number of face data variables as every other patch.

Example File Set:

Grid File	xyz.bin
Structured Boundary File	xyz.bin.fvbnd

Q File (3D)	case4.q.bin
Face Data Function file (2D)	case4.q.bin.fvsrf
Face Data Function Name file	case4.q.bin.fvsrf.nam

Important: The 2D Function file must be in multi-grid format, even if there is only one boundary patch. Also, the format (ASCII, UNFORMATTED, BINARY) must be the same as for the Results file.

Face Data and Function Name Files

FieldView supports face-based results on boundary surfaces of PLOT3D data. In order to provide face results for a 3D dataset, one of the additional files needed is a standard 2D Function File which contains the face results for those boundary surfaces that have them. In order to communicate the names of the face data functions to **FieldView**, a Function Name file needs to be used, like those described in [Appendix C](#) of this **Reference Manual**.

Example File Set:

Grid File	xyz.bin
Structured Boundary File	xyz.bin.fvbnd
Q File (3D)	case4.q.bin
Face Data Function file (2D)	case4.q.bin.fvsrf
Face Data Function Name file	case4.q.bin.fvsrf.nam

Example:

```
Pressure
Temperature
ShearStress_x; ShearStress
ShearStress_y
ShearStress_z
```



Note: To differentiate face data (boundary variables) from identically named volume variables, all face data names will be appended with `[BNDRY]`, except when default names are used (see below). Therefore, in the above example, on the Function Selection panel, the first variable will appear as `Pressure [BNDRY]`. This is true even if there are no identically named volume variables.

Default Names: Like “normal” Function Files, if a Function Name file does not exist, **FieldView** will use default names. The default names for face data functions are: `B1`, `B2`, ..., `BN`. The same error conditions for “normal” (non-face data) Function Name files hold (see [Appendix C](#) of this **Reference Manual**).

Appendix I Plain Text Export Format

This section describes, with examples, the **FieldView Plain Text** Export format. Files may output vertex values (*IJK* - if structured data), spatial data (*XYZ*) and function data (registers) for any surface type (Computational, Coordinate, Iso- and Boundary) or rake (Streamlines). Export files are ASCII and are formatted as columnar data.

NOTE: Current FieldView versions also support CSV (comma separated values) and computationally efficient binary MAT (Mat file format) exports as additional options for object types **Unstructured Boundary** and **Coordinate Surfaces**. For details on those formats, please refer to [Appendix J](#) and [Appendix K](#), following this section. Note also that exports in CSV / Mat-file formats can be used for input to [Point Query...](#) mentioned in Working with FieldView.



Note: The streamline and 2D Plot export files can be read back into **FieldView**.

Cylindrical Note: When cylindrical coordinates are specified using an FVREG file (Region definition), the *XYZ* labels (and values) become *RTZ* (Radius, Theta, Z) labels and values for Iso-Surfaces, Coordinate Surfaces and Boundary Surfaces. Computational Surface exports are unaffected. See [Chapter 3](#) for more information.

In general, each surface Export file contains two sections. The first contains information about the location of the surface (if applicable) along with grid and function information for the surface. The second section contains information to reconstruct the polygons that constitute the surface.

This second section begins with the header word *GEOMETRY*. The second line contains the point count and the number of polygons to follow. This is followed by a listing of the polygons using the node numbers implied from the node information contained in the first section of the file. No special processing of duplicate nodes will be done. No triangulation will be done for warped (non-planar) polygons. A *GEOMETRY* section line such as

```
4 1 8 10 2
```

indicates that the polygon is a hex (4 nodes) and that it can be recreated by joining the nodal *IJK* or *XYZ* information contained in lines 1, 8, 10 and 2 of the first section of the file, in that order or a permutation thereof (i.e. 8, 10, 2, 1 is valid but mixing the order, such as 1, 10, 8, 2 is not).



Note: Nodes removed by thresholding will not be output. Nodes created by thresholding will be output so that the clipped polygon may also be output in the geometry section.

Computational Surfaces

This will dump out the values at the grid points of the comp surface. The file format shall be: a header showing the grid number and surface type (Grid 4 K=5), followed by a line containing a point count, followed by a header line for the columns of data (I J K X Y Z F1 F2 ...). Next, the data values, sorted in IJK order, are output to the file. The data will be: all 3 coordinates (even the constant one), X, Y, Z and the values from the function registers (up to 2 scalars and 1 vector).



Note: The iso-surface register will not be output. An example file follows. This file has been truncated for brevity while still displaying the pertinent aspects of the file format. A "(...cont.)" will signify truncation. The dataset used for the examples is a simple cube with 4 nodes in each IJK direction. Also, for ease of display, the number of spaces between each column of the data values has been decreased from 13 to 5.

Example:

```
Grid 1 I = 2
36
I      J      K      X      Y      Z      S      U      V      W      T
2      1      1      0      0      1      10     4      44     35     -53.14
2      2      1      0      1      1      1      44     35     26     -761.8
(...cont.)
GEOMETRY
9
4 1 8 10 2
4 3 11 13 4
4 5 16 18 6
4 7 19 22 9
4 12 21 25 15
4 14 26 29 17
4 20 31 32 23
4 24 33 34 27
4 28 35 36 30
```

Coordinate Surfaces

By default the data at the nodes of the coordinate surface (the locations stored by **FieldView** to draw the surface) will be output. Again, the file starts with a header showing the type and location of surface (X=0.5), followed by a line containing a point count, followed by a header line for the columns of data (X Y Z F1 F2 ...). Next, the data values are output to the file. The data should be: the 3 coordinates (even the constant one) and the values from the 3 function registers. The data should be sorted in coordinate order. For example, a Z surface will first be sorted in X, with a minor sort in Y, an X surface sorted by Y, and a Y surface sorted by X.

If the user has uniform sampled vectors for the current surface, these X , Y , Z locations and scalar and vector function values will be output instead of the nodal data. The geometry section will be created by outputting a single polygon containing all the points. The point count for this “polygon” will be zero.



Cylindrical Note: When cylindrical coordinates are specified using an FVREG file (Region definition), the XYZ labels (and values) become RTZ (Radius, Theta, Z) labels and values. See [Chapter 3](#) for more information.

Example:

```
X = 1.5
36
X      Y      Z      S      U      V      W      T
1.5    0      0      18     7.5    23     40     -21.597
1.5    1      0      7.5    21.5   38.5    31     -371.286
1.5    1      0      7.5    21.5   38.5    31     -371.286
(...cont.)
GEOMETRY
9
4 2 1 7 11
4 12 8 20 22
4 23 19 31 32
4 4 3 9 15
4 16 10 21 28
4 27 24 33 34
4 6 5 13 17
4 18 14 25 30
4 29 26 35 36
```

Iso-Surfaces

The file will begin with header showing the type of surface ($Pressure = 0.618$), followed by a line containing a point count, followed by the column header and data as described for Coordinate Surface.



Cylindrical Note: When cylindrical coordinates are specified using an FVREG file (Region definition), the XYZ labels (and values) become RTZ (Radius, Theta, Z) labels and values. See [Chapter 3](#) for more information.

Example:

```
Density (Q1) = 1.52256
27
X      Y      Z      S      U      V      W      T
```

```

0      0.03  0      1.52  42.6  33.6  24.6  -693.683
1      0.97  0      1.53  39.2  35.4  26.3  -680.365
(...cont.)
0      1.01  1      1.53  43.8  34.8  25.8  -751.955
0      1      1.0    1.53  41.9  35.5  26.5  -724.669
GEOMETRY
7
3 2 5 17
6 3 1 15 19 21 14
3 20 18 27
6 6 9 13 22 25 16
3 24 23 26
3 8 4 12
3 10 7 11

```

Boundary Surfaces

The boundary faces and values at the grid point should be output. The first line is a header (which should read: Boundary Surface). Next is a line containing a count of boundary types, followed by the boundary types (one per line). The boundaries listed are types that are used by the exported surface. Following this is the data. Data is identical to an Iso-surface and Coordinate Surface export (except that it is not sorted) for an unstructured boundary surface. For Structured Boundary Surfaces, the output will be like a series of Computational Surfaces, except that the points will not be sorted.



Cylindrical Note: When cylindrical coordinates are specified using an FVREG file (Region definition), the `XYZ` labels (and values) become `RTZ` (Radius, Theta, Z) labels and values. See [Chapter 3](#) for more information.

Face Data Note: Boundary surface export files will not contain face data for those registers loaded with face data. Only those registers with non-face data (“volume” data) will be exported. The export file contains nodal values, and the nodes can be shared by more than one face, so in the example below, the Threshold register (normally indicated by a “T” column) was loaded with face data, and so was not exported. The numbers and spacing of the following example have been compressed for better readability.

Example:

```

Boundary Surface
3
body      (7616 faces)
wing      (2400 faces)
tail      (1056 faces)
Grid 1 J = 1
7168
I   J   K   X       Y       Z       S       U       V       W

```

1	1	1	0	0	0	0.994	0.218	0.201	0
1	1	2	0	0	0	0.994	0.218	0.201	0.0141
2	1	2	0.03	0.022	0.00208	0.9927	0.2649	0.1527	0.0090
2	1	1	0.031	0.0230	0	0.99275	0.2644	0.15300	5.90721e-018
2	1	1	0.0314	0.023	0	0.9927	0.2644	0.15300	5.90721e-018
2	1	2	0.0314	0.0229	0.00208	0.9927	0.2642	0.15275	0.00903411
3	1	2	0.0979	0.0524	0.00445	0.99042	0.28535	0.1098	0.0042317
...									
32	1	57	1.39	-0.058	0	1.0286	0.1852	-0.00045	1.14027e-017
33	1	57	1.41	-0.059	0	1.0289	0.1847	-0.0060	1.13987e-017
33	1	56	1.419	-0.057	0.0147	1.0276	0.1840	0.0022	0.0146064

GEOMETRY

1792

4 1 2 3 4

4 5 6 7 8

4 9 10 11 12

4 13 14 15 16

4 17 18 19 20

4 21 22 23 24

4 25 26 27 28

4 29 30 31 32

4 33 34 35 36

(etc. for each surface that comprises the boundary surface)

Streamlines

The current rake of streamlines data will be output using the **FieldView** particle path format. This information will include the **X**, **Y** and **Z** coordinates, **Duration**, and the current scalar. Each streamline's points are sorted in time. Note: The **Duration** in the streamline export file is the residence time.

Example:

FVPARTICLES 2 1

Tag Names

0

VARIABLE NAMES

2

Duration

Density (Q1)

201

1.45503	0.189872	0.215411	0	0.867428
1.46802	0.189749	0.216805	0.0433329	0.867534
1.48291	0.189568	0.21886	0.0929378	0.867516
1.49566	0.189462	0.220879	0.135466	0.867403
1.50693	0.189436	0.222858	0.173108	0.86736

1.5153	0.18947	0.224442	0.201096	0.867307
1.52228	0.189541	0.225838	0.224464	0.867247
1.52715	0.189616	0.226851	0.240782	0.8672
1.53129	0.189698	0.227738	0.254701	0.867147
1.53471	0.189781	0.228488	0.266208	0.867094
1.53723	0.189851	0.229049	0.27467	0.867047
1.53943	0.189919	0.229547	0.282103	0.867001
1.54133	0.189984	0.229981	0.28851	0.866954
1.54308	0.190048	0.230386	0.294425	0.866903
1.54485	0.190118	0.230797	0.300389	0.866848
1.54663	0.190192	0.231215	0.306404	0.866786
1.54845	0.190273	0.231649	0.312584	0.866715
1.55032	0.19036	0.232096	0.318909	0.866639
1.55223	0.190454	0.232558	0.325379	0.866555
1.55427	0.19056	0.233059	0.332331	0.866633
1.55641	0.190679	0.233589	0.339604	0.866708
1.55865	0.19081	0.234147	0.347192	0.866776
1.56112	0.190963	0.234773	0.35561	0.86684
1.56374	0.191135	0.235446	0.364542	0.866952
1.56651	0.191326	0.236166	0.373992	0.866952

Appendix J MAT-File Export

This export uses the MAT-File version 6 format. It is binary, uncompressed, and has a limit of 2^{31} bytes per variable. It contains up to 8 variables, each one being stored as a separate single precision array of dimension $N \times 1$, N being the number of nodes on the exported surface. The arrays are named X Y and Z for the coordinates, S for the current scalar, U V W for the components of the current vector and T for the current threshold scalar. If any of these functions is undefined, the array will still be present, but all of the data values will be NaN (not a number). Also, the variable name in the TOC (see next paragraph) will be "None".

An additional cell array named TOC (Table of Contents) will be exported, containing the full length variable names as they appear in **FieldView**. For unsteady datasets, the current time step and physical time (if available) will also be exported.

Note that as opposed to the text export, the MAT-File export does not include any information to reconstruct the polygons of the surface. Also, each node of the surface mesh will only be listed once.

Note also that exports in CSV / Mat-file formats can be used for input to [Point Query...](#) mentioned in Working with FieldView.

Appendix K CSV Export

The file will begin with two lines of header:

- The first line starts with a "#" character, a common convention in CSV files for comments. If the current dataset is unsteady, it is followed by the keyword "STEP=" and the current time step number. If physical time is defined for this dataset, it will also be exported after the keyword "TIME="
- The second line lists the content of the file. Coordinates will always be exported, followed by the current scalar, the three components of the current vector and the current threshold scalar. If any of these functions is undefined, the column will still be present, but the name will be "None" and all of the data values will be NaN (not a number).

The rest of the file lists the coordinates and variables mentioned in the first line for all vertices on the current surface. See the simplified example below.

```
#STEP=10,TIME=0.1
"X","Y","Z","pressure","U","V","W","temperature"
0,0,0.5,0.5,1,2,3,350.01
0,0.5,0.5,0.6,1,2,3,350.03
0.5,0.5,0.5,0.7,1,2,3,350.02
0.5,0,0.5,0.6,1,2,3,350.04
1,0.5,0.5,0.8,1,2,3,350.07
1,0,0.5,0.7,1,2,3,350.05
0,1,0.5,0.7,1,2,3,350.06
0.5,1,0.5,0.8,1,2,3,350.08
1,1,0.5,0.9,1,2,3,350.09
```

Note that as opposed to the text export, the CSV export does not include any information to reconstruct the polygons of the surface. Also, each node of the surface mesh will only be listed once.

Note also that exports in CSV / Mat-file formats can be used for input to [Point Query...](#) mentioned in [Working with FieldView](#).

Appendix L Particle Path Formats

This section describes, with examples, the **FieldView** Particle Path formats. The ASCII file is a user generated format that is also used by the Streamline Export and the Vortex Cores / Surface Flows Export. This format is for steady state problems. The BINARY format is used by **FieldView** when transient streaklines are calculated. The BINARY format is for transient data only. The BINARY format is *not* merely a binary version of the ASCII format.

The **FieldView** Particle Path Format describes a set of paths, with each path being made up of a series of points. For each point along a path, an *x*, *y* and *z* location value is specified. In addition, a number of variables may be defined at that location.

ASCII

The ASCII Particle Path format is used by the Streamline Export and the Vortex Cores / Surface Flows Export tools and can also be used to create particle path files for import to **FieldView**. The ASCII Particle Path format consists of a header section describing global information about the paths followed by 'blocks' of data, one for each path. The following small example illustrates the general format. A line-by-line description follows. Comments in parentheses are for illustration only and cannot be included in the file.

Important Note: This format is for steady state streamlines only. For transient streaklines, the BINARY format must be used.

```
FVPARTICLES 2 1
Tag Names
2
Inlet
Outlet
Variable Names
3
Time
Pressure
Temperature
5
1

0.0 0.1 1.5 0.1 1013.0 304.321

0.1 0.3 1.5 0.2 1016.3 313.847

0.2 0.5 1.5 0.3 1019.7 322.369

0.4 0.9 1.5 0.4 1022.2 343.332
```

(Path is 5 points long)

(Path has Tag=1, i.e. Inlet. Only include if using Tag Names)

point 1 of Rake 1)

(X, Y, Z, Time, Pressure, Temperature for

point 2 of Rake 1)

(X, Y, Z, Time, Pressure, Temperature for

point 3 of Rake 1)

(X, Y, Z, Time, Pressure, Temperature for

point 4 of Rake 1)

```

0.4 0.9 1.5 0.5 1026.8 347.833
6
2
0.0 0.1 1.7 0.1 1013.0 304.321
0.1 0.3 1.7 0.2 1017.2 315.361
0.2 0.5 1.7 0.3 1021.4 324.873
0.4 0.9 1.7 0.4 1024.7 345.568
0.4 0.9 1.7 0.5 1027.1 350.843
0.6 1.1 1.7 0.6 1033.8 353.733

```

```

(X, Y, Z, Time, Pressure, Temperature for
point 5 of Rake 1)
(Path is 6 points long)
(Path has Tag=2, i.e. Outlet. Only include if using
Tag Names)
(X, Y, Z, Time, Pressure, Temperature for
point 1 of Rake 2)
(X, Y, Z, Time, Pressure, Temperature for
point 2 of Rake 2)
(X, Y, Z, Time, Pressure, Temperature for
point 3 of Rake 2)
(X, Y, Z, Time, Pressure, Temperature for
point 4 of Rake 2)
(X, Y, Z, Time, Pressure, Temperature for
point 5 of Rake 2)
(X, Y, Z, Time, Pressure, Temperature for
point 6 of Rake 2)

```

The above example and following description is for Version 2 1 of the ASCII format. This version allows for, but does not require the use of Tag Names.

The first line must be the keyword and version numbers exactly as shown below:

```
FVPARTICLES 2 1
```

Next, the keywords shown below must appear, to signal the start of the tags section:

```
Tag Names
```

The next line must contain the number of tags in the file (if there are no tags, this will be the number zero):

```
3
```

This is followed by the names of the tags (if any), one per line:

```
Inlet1
Inlet2
Injector
```

Next, the keywords shown below must appear, to signal the start of the variables section:

```
Variable Names
```

The next line must contain the number of variables in the file (if there are no variables, this will be the number zero). The maximum number of particle path variables is 100.

3

This is followed by the names of the variables (if any), one per line:

```
Time
Pressure
Diameter
```

This concludes the header information for the ASCII particle path file. The rest of the particle path file contains a block of information for each path. Each path consists of 1 or more points with an associated tag name. This block consists of the following information. There needs to be 1 block for each path.

The first line of a path block contains the number of point (XYZ) entries for the path:

```
10
```

The next line of a path block contains the tag number for the first path. This associates the path to the list of tag names that appears in the header section of the file.

```
1
```

Lastly, the point information for the path is given. This consists of the X, Y and Z locations as well as the variable values at each location for the path.

0.0 0.1 1.5 0.0 1000.0 0.005	(X, Y, Z, Time, Pressure and Diameter for point 1)
0.0 0.2 1.5 0.1 1100.0 0.004	(X, Y, Z, Time, Pressure and Diameter for point 2)
...	
0.0 1.1 1.5 0.2 1300.0 0.001	(X, Y, Z, Time, Pressure and Diameter for point 10)

If there is more than one path, the next line would contain the number of point entries and tag number for path 2, followed by the X, Y, Z and variable values for that path. There is no limit on the number of paths that may be included in a single **FieldView** Particle Path file. The maximum path length (in points) is 4000.



Note: If `Time` is one of the variables, the time value must increase at each point. If one or more values for time is the same, only the first particle position at that time value is used. If any one of the time values is less than the preceding one, all particles are rejected and an error pop-up is shown.

How do I format the ASCII Particle Path file if I do not want to use Tag Names?

In this case, the format will show that there are zero (0) Tag Names. In addition, there is no tag number after the point entry field at the start of each block. See the example below.

FVPARTICLES 2 1

Tag Names

0

Variable Names

3

Time

Pressure

Temperature

5

0.0 0.1 1.5 0.1 1013.0 304.321

0.1 0.3 1.5 0.2 1016.3 313.847

0.2 0.5 1.5 0.3 1019.7 322.369

0.4 0.9 1.5 0.4 1022.2 343.332

0.4 0.9 1.5 0.5 1026.8 347.833

6

0.0 0.1 1.7 0.1 1013.0 304.321

0.1 0.3 1.7 0.2 1017.2 315.361

0.2 0.5 1.7 0.3 1021.4 324.873

0.4 0.9 1.7 0.4 1024.7 345.568

0.4 0.9 1.7 0.5 1027.1 350.843

0.6 1.1 1.7 0.6 1033.8 353.733

(Path is 5 points long)

(X, Y, Z, Time, Pressure, Temperature for
point 1 of Rake 1)(X, Y, Z, Time, Pressure, Temperature for
point 2 of Rake 1)(X, Y, Z, Time, Pressure, Temperature for
point 3 of Rake 1)(X, Y, Z, Time, Pressure, Temperature for
point 4 of Rake 1)(X, Y, Z, Time, Pressure, Temperature for
point 5 of Rake 1)

(Path is 6 points long)

(X, Y, Z, Time, Pressure, Temperature for
point 1 of Rake 2)(X, Y, Z, Time, Pressure, Temperature for
point 2 of Rake 2)(X, Y, Z, Time, Pressure, Temperature for
point 3 of Rake 2)(X, Y, Z, Time, Pressure, Temperature for
point 4 of Rake 2)(X, Y, Z, Time, Pressure, Temperature for
point 5 of Rake 2)(X, Y, Z, Time, Pressure, Temperature for
point 6 of Rake 2)***Can I use Particle Paths to display stationary "point data" in FieldView?***

Yes. **FieldView's** Particle Path import format can be used to read in point data such as experimental data which can then be displayed as stationary particles. These particles can then be colored with the current scalar, for example, to show differences between the experimental data and the solver data which has been read into **FieldView**. The only useful DISPLAY TYPE is Spheres, Dots or Polyspheres. For these DISPLAY TYPES, you must define one path for each data point. This path has only a single XYZ location. It does not need a 'start' and an 'end' point. You will need as many paths as you have particles. The following example shows what a file might look like for importing experimental data with pressure information at given locations:

FVPARTICLES 2 1

Tag Names

2

```

Rake 1
Rake 2
Variable Names
1
Pressure
1
1
0.0 0.1 1.5 1013.0          (X, Y, Z, Pressure for point 1 of Rake 1)
1
1
0.0 0.2 1.5 1116.5          (X, Y, Z, Pressure for point 2 of Rake 1)
...
1
1
0.0 1.1 1.5 1300.0          (X, Y, Z, Pressure for point N of Rake 1)
1
2
0.0 0.1 1.5 1013.0          (X, Y, Z, Pressure for point 1 of Rake 2)
1
2
0.0 0.2 1.5 1116.5          (X, Y, Z, Pressure for point 2 of Rake 2)
...
1
2
0.0 1.1 1.5 1300.0          (X, Y, Z, Pressure for point N of Rake 2)

```

BINARY

The BINARY Particle Path format is used when **FieldView** creates a transient particle path file during streakline creation. Binary Particle Path files can be created/read by **FieldView** on any platform.

Important Note: This format is for transient streaklines only. For steady state particle paths, the ASCII format must be used.

Sample C code fragments have been included below to help with the writing of this format. Included with the Binary **FieldView**-Unstructured sample code called `write_binary_uns.c` (which can be found in the subdirectory `/uns` of the **FieldView** installation) is a useful subroutine for writing out strings called `fwrite_str80`. In the sample codes fragments below, it is assumed that this subrou-tine is used.

BINARY PARTICLE SET Format Description

The file header for the binary PARTICLE SET format is similar to the older binary STREAKLINE format (described in the section after this one). The only differences are:

- The major version is 3 instead of 1 (version 2 is reserved for ASCII).
- No time step number or solution time is stored in the file header. The time step number will be derived from the file name. There should be an integer time step number (or iteration number) in front of the first '.' in the file name. The number of digits can vary from time step to time step. Leading zeroes are allowed.

Example #1:

```
my_particles01.fvp
my_particles11.fvp
my_particles33.fvp
my_particles101.fvp
...
```

Example #2:

```
my_particles01.Nov2013.fvp
my_particles11.Nov2013.fvp
my_particles33.Nov2013.fvp
my_particles101.Nov2013.fvp
...
```

Within the file body, data is written out as follows:

```
x y z for particle 1
x y z for particle 2
..

scalar#1 for particle 1
scalar#1 for particle 2
...

scalar#2 for particle 1
scalar#2 for particle 2
...
```

The following sample C code assumes that various variables have already been defined. For example,

```
#define MAJOR_VERSION 3
#define MINOR_VERSION 2
#define FV_MAGIC      0x00010203
```

Line	Description	Sample C
integer	FV_MAGIC = 0x00010203	ibuf[0] = FV_MAGIC;
		fwrite(ibuf, sizeof(int), (size_t)1, fp);
string	"FVPARTICLES"	fwrite_str80("FVPARTICLES", fp);
integer	major version number (= 3)	ibuf[0] = MAJOR_VERSION;

integer	minor version number (= 2)	<code>fwrite(ibuf,sizeof(int), (size_t)1, fp);</code> <code>ibuf[0] = MINOR_VERSION;</code>
integer	reserved (= 0)	<code>fwrite(ibuf,sizeof(int), (size_t)1, fp);</code> <code>ibuf[0] = 0;</code>
integer	number of variables	<code>fwrite(ibuf,sizeof(int), (size_t)1, fp);</code> <code>ibuf[0] = num_vars;</code> <code>fwrite(ibuf,sizeof(int), (size_t)1, fp);</code>
	(then, for each variable)	
string	variable name	<code>for (i = 0; i < num_vars; i++)</code> <code>{fwrite_str80 (var_names[i], fp);}</code>
integer	number of particles	<code>ibuf[0] = num_part;</code> <code>fwrite(ibuf,sizeof(int), (size_t)1, fp);</code>
	(then, for each particle)	
float	x	<code>for (j = 0; j < num_part; j++)</code> <code>{fwrite(x_values[j],sizeof(float), (size_t)1,</code> <code>fp);</code>
float	y	<code>fwrite(y_values[j], sizeof(float), (size_t)1,</code> <code>fp);</code>
float	z	<code>fwrite(z_values[j], sizeof(float), (size_t)1,</code> <code>fp);}</code>
	(then, for each particle for each variable)	
float	value	<code>for (i = 0; i < num_vars; i++)</code> <code>for (j = 0; j < num_part; j++)</code> <code>{fwrite(var_values[i][j], sizeof(float),</code> <code>(size_t)1, fp);}</code>
where		
integer:	4-byte integer value	
float:	4-byte floating point value	
string:	80-byte string value (unused bytes are filled with NULL characters)	

The following points should be noted:

Mapping is used when the byte order matches the native byte order.

Floating-point data is stored using single-precision.

There are no reserved scalar names. The scalar name "Time" for example is not a reserved name, and does not get used to enable any special functionality within **FieldView**.

Consistent with our ASCII and binary STREAKLINE formats, there is no support for vector variables (only scalars).

As scalars are written to the FVP files, a scalar written for "seed number" would be very useful to the **FieldView** user, as this would provide information about the origin of a given particle. It would also be useful to provide a constant, different for each emitting surface, which would allow users to identify particles originating from a given inlet, for instance.

BINARY STREAKLINE Format Description

The file header for the binary STREAKLINE format is similar to the binary PARTICLE SET format. The only differences are:

- The major version is 1 instead of 3 (version 2 is reserved for ASCII).
- The time step and the solution time for every time step are stored in the file.

The following sample C code assumes that various variables have already been defined. For example,

```
#define MAJOR_VERSION 1.1
#define MINOR_VERSION 1
#define FV_MAGIC      0x00010203
```

Line	Description	Sample C
integer	FV_MAGIC = 0x00010203	ibuf[0] = FV_MAGIC; fwrite(ibuf,sizeof(int), (size_t)1, fp);
string	"FVPARTICLES"	fwrite_str80("FVPARTICLES", fp);
integer	major version number (= 1)	ibuf[0] = MAJOR_VERSION; fwrite(ibuf,sizeof(int), (size_t)1, fp)
integer	minor version number (= 1)	ibuf[0] = MINOR_VERSION; fwrite(ibuf,sizeof(int), (size_t)1, fp)
integer	number of variables other than solution time (then, for each variable other than solution time)	ibuf[0] = num_vars; fwrite(ibuf,sizeof(int), (size_t)1, fp)
string	variable name (then, for each time step)	for (i = 0; i < num_vars; i++) {fwrite_str80 (var_names[i], fp)}
integer	time step number	fwrite(&time_step,sizeof(int), (size_t)1, fp)
float	solution time	fwrite(&solution_time,sizeof(int), (size_t)1, fp)
integer	number of particles (then, for each particle)	fwrite(&num_particles,sizeof(int), (size_t)1, fp)
integer	path ID	fwrite(&pathID[i],sizeof(int), (size_t)1, fp)
float	x	fwrite(x_values[i], sizeof(float), (size_t)1, fp)
float	y	fwrite(y_values[i], sizeof(float), (size_t)1, fp)
float	z	fwrite(z_values[i], sizeof(float), (size_t)1, fp)


fp)

(then, for each variable)

float value

`fwrite(var_values[i][j], sizeof(float),`
`(size_t)1, fp)`

- where
- integer: 4-byte integer value
 - float: 4-byte floating point value
 - string: 80-byte string value (unused bytes are filled with NULL characters)



Note: The `FV_MAGIC` constant should always be written out as a 4-byte integer, and not as a series of 4 bytes. On read in, it will be used to determine whether or not byte-swapping is needed.

Note: Solution time must be continuously ascending.

Note: Time step number must start at 1.

Note: The Path ID identifies which path a particle belongs to. Path IDs must start at 1.

Note: The solution time of the first point in a path becomes the particle path data variable "Emission Time" when imported into **FieldView**.

Appendix M FieldView Math Fonts

The Annotation panel allows you to add annotation to your visualization. Two of the font formats available are Math Upper Case and Math Lower Case. The font mappings of these two fonts with the standard US keyboard is presented in the following table. This will ease your use of these fonts for special characters that may be needed for your annotations.

The alpha-numeric symbols are illustrated in the following tables.

Lower case keyboard characters:

KB	`	1	2	3	4	5	6	7	8	9	0	-	=
MU	°	1	2	3	4	5	6	7	8	9	0	-	=
ML	°	1	2	3	4	5	6	7	8	9	0	-	=

KB = keyboard , MU = Math Upper Case Font , ML = Math Lower Case Font

KB	q	w	e	r	t	y	u	i	o	p	[]	\
MU	\int	\approx	\cup	$($	$[$	\parallel	$]$	\rightarrow	$\sqrt{}$	\int	$[$	$]$	\backslash
ML	\int	\approx	\cup	$($	$[$	\parallel	$]$	\rightarrow	$\sqrt{}$	\int	$[$	$]$	\backslash

KB = keyboard , MU = Math Upper Case Font , ML = Math Lower Case Font

KB	a	s	d	f	g	h	j	k	l	;	'		
MU	$\$$	$)$	\subset	\supset	\cap	\in	\uparrow	\leftarrow	\downarrow	Σ	\cong		
ML	$\$$	$)$	\subset	\supset	\cap	\in	\uparrow	\leftarrow	\downarrow	Σ	\cong		

KB = keyboard , MU = Math Upper Case Font , ML = Math Lower Case Font

KB	Z	X	c	v	b	n	m	,	.	/			
MU	\perp	\div	$\sqrt{}$	\exists	$\sqrt{}$	∇	∂	,	.	/			
ML	\perp	\div	$\sqrt{}$	\exists	$\sqrt{}$	∇	∂	,	.	/			

KB = keyboard, MU = Math Upper Case Font, ML = Math Lower Case Font

KB	~	!	@	#	\$	%	↑	&	*	()	_	+
MU	\sim	\pm	\equiv	\times	.	%	∞	\leq	*	()	∞	+
ML	\sim	\pm	\equiv	\times	.	%	∞	\leq	*	()	∞	+

KB = keyboard, MU = Math Upper Case Font, ML = Math Lower Case Font

KB	Q	W	E	R	T	Y	U	I	O	P	{	}	
MU	Q	W	E	R	T	Y	U	I	O	P	{	}	\angle
ML	q	w	e	r	t	y	u	i	o	p	{	}	\angle

KB = keyboard, MU = Math Upper Case Font, ML = Math Lower Case Font

KB	A	S	D	F	G	H	J	K	L	:	“		
MU	A	S	D	F	G	H	J	K	L	Π	\mp		
ML	a	s	d	f	g	h	j	k	l	Π	\mp		

KB = keyboard, MU = Math Upper Case Font, ML = Math Lower Case Font

KB	Z	X	C	V	B	N	M	<	>	?			
MU	Z	X	C	V	B	N	M	<	>	\neq			
ML	z	x	c	v	b	n	m	>	<	\neq			

KB = keyboard, MU = Math Upper Case Font, ML = Math Lower Case Font

Appendix N NPARC/WIND Constants and Formulas

This appendix describes the handling of constants by the NPARC/WIND reader in **FieldView** as well as the formulas available through a Formula Restart (see [Chapter 5](#) for more information about restart files) file.

The NPARC/WIND reader will convert the following constants associated with each grid in an NPARC/WIND file from the SI (MKS) system of units to the English (FSS) system of units:

Constant	From MKS unit	To FSS unit	Description
Pinf	N/M ²	lb _f /ft ²	freestream static pressure
Tinf	K (Kelvin)	°R (Rankine)	freestream static temperature
R	J/(kg-K)	ft*lb _f /(slug-°R)	gas constant

The added annotation '**inf**' denotes freestream conditions. These constants, in addition to **Gamma** (γ , ratio of specific heats, $c_p/c_v = 1.4$), **FSMach** (Mach number) and **Re** (Reynold's number) are available as buttons in the Function Formula Specification panel.

In addition, an NPARC/WIND Formula Restart file `wind.frm` can be found in `fvx_and_restarts` subdirectory of the **FieldView** installation. This Restart file can be started interactively from the File menu Open Restart fly-out, or on the command line by using the "**-f**" argument and the path to the `wind.frm` file.

See [Chapter 1](#) of the **User's Guide**, for a complete description of the command line arguments.

The NPARC/WIND Formula Restart file `wind.frm` defines several additional flow quantities. These are functions of the above constants and the five NPARC/WIND flow quantities that already exist for each grid in the NPARC/WIND Results file:

NPARC/WIND Flow Quantities	
"rho, density"	(ρ)
"rho*u, x-momentum"	(ρu)
"rho*v, y-momentum"	(ρv)
"rho*w, z-momentum"	(ρw)
"rho*e0, stagnation energy"	(ρe_0)

The formulas for the additional flow quantities defined in the `wind.frm` file are defined by using these 5 flow quantities and quantities defined with them.

wind.frm Variables					
Variable	Designation	in	Type	Symbol	Units
FieldView					
Velocity [WIND]			vector	V	ft/s
Pressure [WIND]			scalar	p _{calc}	lb _f /ft ²
Pressure (lbf/in2) [WIND]			"	p _{PSI}	lb _f /in ² (PSI)
Temperature [WIND]			"	T	°R
C _p [WIND]			"	C _p	(dimensionless)
Stagnation Pressure [WIND]			"	p ₀	lb _f /ft ²
Enthalpy [WIND]			"	h	ft ² /s ²
Entropy [WIND]			"	s	ft*lb _f (slug-°R)

These flow quantities are defined in the following way

$$u = \rho \mathbf{u} / \rho \quad (1)$$

$$v = \rho \mathbf{v} / \rho \quad (2)$$

$$w = \rho \mathbf{w} / \rho \quad (3)$$

$$e_0 = \rho \mathbf{e}_0 / \rho \quad (4)$$

$$V^2/2 = 0.5(u^2 + v^2 + w^2) \quad (5)$$

$$\mathbf{V} = (u) \nabla X + (v) \nabla Y + (w) \nabla Z \quad (6)$$

$$|\mathbf{V}| = \text{sqrt}(u^2 + v^2 + w^2) \quad (7)$$

$$p_{\text{calc}} = (\gamma - 1) \rho (e_0 - V^2/2) \quad (8)$$

$$p_{\text{PSI}} = p_{\text{calc}} / 144.0 \quad (9)$$

$$T = p_{\text{calc}} / (\rho R) \quad (10)$$

$$C_p = [2 / (\gamma M^2)] (p_{\text{calc}} / p_{\text{inf}} - 1) \quad (11)$$

$$a = \text{sqrt}(\gamma R T) \quad (12)$$

$$M_t = |\mathbf{V}| / a \quad (13)$$

$$p_0 = p_{\text{calc}} [1 + M_t^2 (\gamma - 1) / 2]^{\gamma / (\gamma - 1)} \quad (14)$$

$$h = \gamma (e_0 - 0.5 |\mathbf{V}|^2) \quad (15)$$

$$s = [R / (\gamma - 1)] \ln [(p_{\text{calc}} / p_{\text{inf}}) / (p_{\text{inf}} / (R T_{\text{inf}}))^{\gamma}] \quad (16)$$

The equation (9) for p_{PSI} follows from the simple conversion formula $1 \text{ ft}^2 = 144 \text{ in}^2$. Equations (8), (10) and (14-16) use the same formulas as the PLOT3D functions, but are dimensional and in the FSS system of units. It can be shown that equation (11) follows from the definition of the pressure coefficient

$$C_p \equiv (p - p_{inf}) / (0.5 \rho_{inf} V_{inf}^2) \quad (17)$$

by expressing the dynamic pressure $(0.5 \rho_{inf} V_{inf}^2)$ in terms of the freestream pressure and the freestream Mach number ($M \equiv V/c$ where c is the speed of sound).

Index

Numerics

2D Plot Controls Panel	
Cylindrical coord. (Regions)	112
2D Plot Format	515
cylindrical	515

A

Actions (keyframe)	317
AcuSolve	1
AcuSolve Direct Reader	1
Animation	316
Flipbooks	309, 310, 311
Holding View	305
Streamlines	305
View Interpolation	305
Animation Data	28
anut	44
Append	5, 11, 406
Appending Datasets, same Server Process	12
Attribute Actions (keyframe)	319
Auxiliary Seed Plane	
Cylindrical coord. (Regions)	112
AVUS	1

B

Background Color	339
BANFF	1
Blade Row	118
blades per row (regions)	118, 122, 141
Boundary Data Only	9
Boundary Surface Panel	516, 517

C

C language	394, 404
CFD Calculator	344, 351, 439, 460, 485, 491
CFD++	1
CFD-ACE	1
CFX-4	517
dmp files	517
fvbnd files	516, 517, 518
CFX-5	1

Export to FV-UNS	79
CFX-TASCflow	2, 95
CGNS	2, 387
chunk	143
COBALT	2
COBALT60	2
Colormap File Format	509
Colormap Specification Panel	509
command-line switches	
-f	262
-gamma	403
-gasconstant	403
-s	262
Complete Restart	260, 262, 266, 267, 279
Computational Surface Panel	405, 406
Constants, function	103
Contours	
Filled	278, 513
Control Panels	
Colormap Specification Panel	509
Dataset Controls	11
Scaling	331
Sweeping	11
Flipbook Controls Panel	
Graphics Layout Size (NTSC, PAL, D1)	300
Integration Controls Panel	
Multiple Surfaces/Functions	307
Script	289
Sweep Integration	113
Transform Controls	
Dataset Switching	11
Demotion	285
Detach	308, 331
Object	
Region	111
Transient Data	
Looping	301
Transient Data Controls Panel	314, 404
Coordinate Surface Panel	
cylindrical coord. (Regions)	112
Subsetting	111
CTH	41
Current Frame (keyframe)	321
Curve Length	350

Cutting Plane	NONE286
Cylindrical coord. (Regions) 112	ROTATE287
Cylindrical coordinates (regions)	
Coordinate surfaces 112	
Point Probe 113	
D	E
Data	Equal Length Vectors343
FV-UNS 37, 100, 101, 103, 286, 290, 527	Experimental Data535
PLOT3D 55, 100, 394, 397, 402, 403, 404,405, 545	Export524
Merge Series 404, 405, 409, 410	cylindrical coord. (Regions) 113, 289, 524,526, 527
Q variables 102, 397, 399, 403, 404	Streamline55, 524, 528, 532
Transient ..6, 37, 314, 394, 397, 404, 410	Export Format524
DataGuide403	
Regions 111	F
Dataset	Face Data
Appending	FV-UNS 37, 100, 101, 103, 286, 290, 527
Same Server 12	PLOT3D 57, 100, 101, 103, 286, 394, 400, .411, 517, 521, 522, 523, 527
Different Systems 12	PLOT3D Face Data file (fvsrf) ...400, 522,523
Reading multiple 11	Surface Normals290
Dataset Comparison 11	Feature Detection Formulas349
Dataset Controls 11	Feature Extraction
Duplication	Separation / Reattachment532
Mirroring 275	Surface Flows532
Rotation 275	Vortex Cores532
Negative Scaling331	FENSAP2
Reflection331	FIDAP2, 27, 84
Scaling331	FDNEUT27
Sweeping308	FieldView Limits512
Dataset Reflection331	FIELDVIEW-UNS
Dataset Switching 11	face data 37, 100, 101, 103, 286, 290, 527
Datasets	Filled Contours278, 513
Multiple 11, 100, 262	FIRE2
dbx251	Fixed Vectors (regions) 119
debugger251	Flipbook Controls Panel311
Demotion285	Graphics Layout Size (NTSC, PAL, D1)300
Detach308, 331	Flipbooks309, 310, 311
Differences between Datasets 107	Flooded Contours278, 513
DIRECT Readers 20	Flow Science2
dmp files517	FLOW-3D
dump(variable)247	Files
DUPLICATION286	FLSINP 28, 29
MIRROR286	

FLOW-3D®	2, 28	Surface Normals	342
FLOW-3D® Animation Data	2	Unit Vectors	341, 347, 348
FLOW-3D® Restart Data	2	Volume Integrals	346
flsgrf.dat	28	fv	
FLUENT	2	-f	262
Fluent	36	-gamma	403
FIDAP	27	-gasconstant	403
2D DATA	27	-s	262
FDNEUT	27	FV_2D_to_3D	9
Rampant-Fluent/UNS	36	FV_ACUSOLVE_GRAD	23
FLUENT Direct Reader	2	FV_ARB_POLY	17
FLUENT Universal	2	FV_HOME	355
FLUENT/UNS (and RAMPANT)	2	FV_NO_FVX_RESTART	258
Force	348	FV_PLUGINS	22
FORTTRAN language	354, 356, 394, 404	FV_TET_CONV	17
Files		fvp	532, 536
binary	37, 403, 404	FVREG file	113, 117, 118
formatted	403, 404	Blade Row section	118
unformatted	403, 404	Dataset section	118, 119, 138
Frame Number (keyframe)	321	Fixed Vectors section	119
FrontFLOW	2	FV-UNS	
Function Files	57, 394, 399, 400, 403, 404, 407, 408, 411, 517, 521, 522, 523	face data	37, 100, 101, 103, 286, 290, 527
Function Formula Specification	103, 105, 106, 341, 543	FVX	
Constants	103	Templates	257
Non-rotating vectors	344	G	
Operations	105	-gamma	403
Rotating Quantities	351	gamma	352, 403, 543
Function Name File	399, 404	-gasconstant	403
Use Defaults	403, 407, 412	gasconstant	403
Function Selection	5, 100, 103	GASP	2
Function Specification Panel	100, 103	GLACIER	3, 95
Registers	100, 102, 103, 106, 107	Graphics Layout Size (NTSC, PAL, D1)	300, 333
Scalar	100, 102, 103, 391	GRID PROCESSING	4
Threshold	100, 102, 103	H	
Vector	6, 100, 102, 103, 341, 343	HAVOC	3, 41
Functions		Holding View (animations)	305
Curve Length	350	I	
Equal Length Vectors	343	IBLANK	28, 400
Force	348	Import	
Line Integrals	346	2D Plot	113
Normal Vectors	347, 348	Improved Restarts	268
PLOT3D	55, 100, 103, 351, 391, 393, 394, 402, 545		
Second derivatives	350		

Increment (Inc)	36, 350	Multiple datasets	11, 100, 262
Integration		N	
Multiple Surfaces/Functions	307	Negative Scaling	331
Integration Controls Panel		newlink plot3d_or_overflow-2_options	169
Multiple Surfaces/Functions	307	Non-rotating vectors	344
Script	289	Normal Vectors	347, 348
Sweep Integration		NPARC/WIND	3
cylindrical coord. (Regions)	113	O	
Interpolated Actions (keyframe)	318	OpenFOAM	3
Iso-Surface Panel		Operations, function	105
Cutting Plane		Out of Range Handling	109
Cylindrical coord. (Regions)	112	OVERFLOW-2	3, 48
Subsetting	111	OVERFLOW-2 Auto-detect	8
K		OVERFLOW-2, brk files	51
Keyframe Actions	317	OVERFLOW-2, Transient File Naming	51
Attribute	319	P	
Interpolated	318	panel 244, 279, 341, 342, 344, 346, 350,	351
Simple	317	Particle Path Panel	534
Keyframe Animation	316	Particle Paths	
Graphics Layout Size (NTSC, PAL, D1)		Experimental Data	535
.....	300	FIELDVIEW	532, 534
Perspective Warning	328, 329	formats	
Keyframe Animation Panel	320	ASCII 532	
Current Frame	321	BINARY 536	
Frame Number	321	period (regions)	118, 122, 141
Restarts	326	Perspective	328, 329
Keyframe Restarts	326	Perspective Warning	328, 329
Keyframe Time Line	324	PFPR, Layout File	74
Keyframe Time Line Track Selection	324	PFPR, PLOT3D example	74, 75
Keyframe Track Selection Panel	322	PHI	54
Keyframe Value Specification Panel	324	PHOENICS	54
L		Phoenix	54
Limits, FieldView	512, 513	PHI file	54
Line Integrals	346	PHOENICS - BFC Data	3
LINKED_SURFACE_SWEEP	291	PHOENICS - non-BFC Data	3
Local Parallel	22	PLOT3D 55, 351, 391, 393, 394, 402, 403,	
Loop	314	404, 405, 545
Looping	301	Data	
LS-DYNA	3	Function File 57, 394, 399, 400,	
M		403, 404, 407, 408, 411, 517,	
Merge Series	404, 405, 409, 410	521, 522, 523	
Merged Transient	12	Function Name File 57, 394, 399,	
Mirroring	275		

400, 404, 411, 517, 521, 522, 523	Blade Row section 118
Use Defaults 403, 407, 412	Dataset section 118, 119, 138
Merge Series 404, 405, 409, 410	Fixed Vectors section 119
XYZ files 395, 403, 405, 406, 407	Subsetting 111
PLOT3D Auto-Detect 8	Regions, Machine Axis 118
PLOT3D, Transient 55	Register 100, 103, 106
Point Increment 407	Rendering
Point Probe Format 515	Presentation Rendering 299
cylindrical 515	Shine 280
Point Probe Panel	Specular Shading 280
cylindrical coord. (Regions) 113	Replace 5, 11, 406
Output 515	Restart Data 28
Regions 111	Restart Files
Point Query 524	Complete ... 260, 262, 266, 267, 279
POLYFLOW 3	Keyframe 326
PostScript 335, 337	Preference 274
PowerFlow 3	Restarts
Preference Restart 274	Formula
Presentation Rendering 299	NPARC/WIND 543
Shine 280	Rotating Quantities 351
Specular Shading 280	Rotation, Duplication
Presentation Rendering Highlight Size .. 299	Dataset 275
Printing	Regions 116
DPI 336, 337	S
PS 335, 337	Sampled Datasets 285
Resolution 337	Scaling 331
Python-Enabled FVX 253	Script Language 260, 274, 280, 301
R	Animating Streamlines 305
Rampant 36	Export ... 55, 288, 301, 524, 528, 532
RavenCFD 3	cylindrical coord. (Regions) 113, 289, 524, 526, 527
Refresh 299	Holding View 305
Region Hierarchy 112	Integrate 289, 307
Region Transform 111	Print PS 337
Regions 113, 118	SIZE 300
Blade Row 118	View Interpolation 305
cylindrical coordinates ... 37, 55, 112, 113, . 118, 289, 515, 524, 526, 527	SCRYU 3
DataGuide 111	Second derivatives 350
Detach 308, 331	Separation / Reattachment 532
Duplication	set_auto_redraw() 246
Rotation 116	set_preserve_globals() 247
FVREG file 113, 117, 118	Shine 299
	Simple Actions (keyframe) 317
	-size 300
	SIZE (script command) 300

Skip	314	Thresholding	346, 348, 524
Space Bar Viewer toggle	330	TIME SET MERGEDTIMES	303
Specular Shading	280	Toolkit	356
STAR-CCM+	3	Tools	
stop()	248	Dataset Controls	
Streakline Export		Duplication	
format	536	Rotation	275
Streamlines & Streaklines 103, 301, 305,		Mirroring	275
406,	516	Export	55, 524, 528, 532
Auxiliary Seed Plane		Graphics Layout Size	300, 333
Cylindrical coord. (Regions) 112		Keyframe Animation Panel	320
Export	55, 524, 528, 532	Current Frame	321
Looping	301	Frame Number	321
Presentation Rendering	299	Restarts	326
Subsetting	111	Transform Controls	
Structured boundary file (fvbnd) ..44, 57,		Dataset Switching	11
394, .400, 411, 516, 517, 518, 521, 522,		Demotion	285
523		Detach	308, 331
Subsetting		Object	
Regions	111	Region	111
Subvolumes	110	Transient Data	
Surface Detach	308, 331	Looping	301
Surface Flows	532	Transient Data Controls Panel .314, 404	
Surface Normals	290, 342	Loop	314
Surface Plot		Skip	314
Export	524	Transient Particle Paths	
Surface-based face data		format	536
FV-UNS 37, 100, 101, 103, 286, 290,		Transient Sweep	314
527		Translators	
PLOT3D 57, 100, 101, 103, 286, 394,		CFX-TASCflow	95
400, .411, 517, 521, 522, 523,		PHOENICS	54
527			
Surface Normals	290	U	
Sweep		Uniform Sampling (vectors)	
Transient	314	cylindrical coord. (Regions)	112
Sweep Integration		Unit Vectors	341, 347, 348
cylindrical coord. (Regions)	113	Use Defaults	403, 407, 412
Sweep, Dataset	308	User-Defined Data Reader	356
		User-Defined Functions	356
T		USM3D	3
table 54, 95, 105, 112, 303, 354, 357, 403,			
.425, 426, 434, 440, 442, 452, 454, 455,		V	
466, ..468, 469, 478, 491, 504, 505, 506		VECTIS	3
tasc2pl3d	95	Vector Quantities	341
Tetrex	3	Components	342
ThermoAnalytics	3	Equal Length Vectors	343

Surface Normals	342
Unit Vectors	341
Vectors	
uniform sampling	
cylindrical coordinates	112
Velocities Section (regions)	119, 139
View	
Presentation Rendering	299
Highlight Size	299
View Interpolation (animations)	305
Viewer Toolbar	
Space Bar toggle	330
Visualization Panels	
2D Plot Controls Panel	
Cylindrical coord. (Regions)	112
Boundary Surface Panel	516, 517
Computational Surface Panel	405, 406
Coordinate Surface Panel	111
cylindrical coord. (Regions)	112
Iso-Surface Panel	
Subsetting	111
Particle Path Surface Panel	532, 534
Particle Paths	
Experimental Data	535
Point Probe Panel	
cylindrical coord. (Regions)	113
Output	515
Regions	111
Streamlines	103, 301, 406, 516
Auxiliary Seed Plane	
Cylindrical coord. (Regions)	112
Cylindrical coord. (Regions)	112
Looping	301
Streamlines & Streaklines	
Subsetting	111
Volume Integrals	346
Vortex Cores	532
W	
wheel speed (regions)	118, 122, 141
WIND	44, 103, 517, 518, 543
fvbnd files	44, 516, 517, 518
Variables	44, 103, 517, 518, 543
wind.frm	543, 544