https://ntrs.nasa.gov/search.jsp?R=19990027454 2020-06-15T22:22:24+00:00Z

Automated Fluid Feature Extraction from Transient Simulations

IN-34
028572

Progress Report

Robert Haimes
haimes@orville.mit.edu
and
David Lovely

February 8, 1999

NASA Ames Research Center
Agreement: NCC2-985

Department of Aeronautics and Astronautics
Massachusetts Institute of Technology
77 Massachusetts Avenue
Cambridge, MA 02139

1 Introduction

In the past, feature extraction and identification were interesting concepts, but not required to understand the underlying physics of a steady flow field. This is because the results of the more traditional tools like iso-surfaces, cuts and streamlines were more interactive and easily abstracted so they could be represented to the investigator. These tools worked and properly conveyed the collected information at the expense of much interaction. For unsteady flow-fields, the investigator does not have the luxury of spending time scanning only one "snap-shot" of the simulation. Automated assistance is required in pointing out areas of potential interest contained within the flow. This must not require a heavy compute burden (the visualization should not significantly slow down the solution procedure for co-processing environments like **pV3**). And methods must be developed to abstract the feature and display it in a manner that physically makes sense.

The following is a list of the important physical phenomena found in transient (and steady-state) fluid flow:

1.1 Shocks

The display of shocks is simple; a shock is a surface in 3-space. As the solution progresses, in an unsteady simulation, the investigator can view the changing shape of the shock surfaces. Some previous work has been done at MIT (as well as other places) on this problem. This early work [Darmofal91] developed the following algorithm:

First determine the normal direction to the shock. Across a shock, the tangential velocity component does not change; thus, the gradient of the speed at a shock is normal to the shock. The exact location of the shock is then determined by calculating the magnitude of the Mach vector, in the direction of the speed gradient, at all points in the domain. The normal Mach number is defined as the Mach vector dotted into the speed gradient. Thus, a positive normal Mach number indicates streamwise compression and a negative normal Mach number indicates expansion. If this value is 1.0 then a shock has been found (or possibly an isentropic recompression through Mach one). This entire iso-surface can be displayed to show the shock, but must be thresholded to remove the surfaces associated with the recompression and some stray portions of the flow field where the normal Mach number happen to be 1.0. The magnitude of the speed gradient was found to be an effective threshold.

1.2 Vortex Cores

Finding these features is important for flow regimes that are vortex dominated (most of which are unsteady) such as flow over delta wings and flow through turbine cascades. Tracking the core can give insight into controlling unsteady lift and fluctuating loadings due to core/surface interactions.

There has been much work done in the location of these features by many investigators. Again, there has been some success [Kenwright97]. This particular algorithm as fully described in [Sujudi95] has been designed so that no serial operations are required, it is parallel, deterministic (with no 'knobs'), and the output is minimal. The method operates on a cell at a time in the domain and disjoint lines are created where the core of swirling flow is found. Only these line segments need to be displayed, reducing the entire vector field to a tiny amount of data.

This technique, although satisfying, is not without problems. These are:

1. Not producing contiguous lines.
   The method, by its nature, does not produce a contiguous line for the vortex core. This is due to two reasons; (1) for element types that are not tetrahedra the interpolant that describes point location within the cell is not linear. This means that if the core passes through these elements the line can display curvature. By subdividing pyramids, prisms, hexahedra and higher-order elements into tetrahedra for this operation produces a piecewise linear approximation of that curve. And (2) there is no guarantee that the line segments will meet up at shared faces between tetrahedra. This is because the eigenvector associated with the real eigenvalue will not be exactly the same in both neighbors, so when this vector is subtracted form the vector values at the shared nodes each tetrahedra sees a differing velocity field for the face.

2. Locating flow features that are not vortices.
   This method finds patterns of swirling flow (of which a vortex core is the prime example). There are other situations where swirling flow is detected, specifically in the formation of boundary layers. Most implementations of this technique do no process cells that touch solid boundaries to avoid producing line segments in these regions. But this does not always solve the problem. In some cases (where the boundary layer is large in comparison to the mesh spacing) this boundary layer generation is still found.

3. Sensitive to other non-local vector features.

   Critical point theory gives one classification for the flow based on the local flow quantities. 3D points can display a limited number of flow topologies including swirling flow, expansion and compression (with either acceleration or deceleration). The flow outside this local view may be more complex and have aspects of all of these components. The local classification will depend on the strongest type. Also if there are two (strong) axes of swirl, the scheme will indicate a rotation that is a combination of these rotation vectors based on the relative strength of each. This has been reported by [Roth96] where the overall vortex core strength was not much greater that the global curvature of the flow. The result was that the reported core location was displaced from the actual vortex.

1.3 Regions of Recirculation

Recirculation is a difficult feature to locate, but a simple one to visualize. A surface exists that separates the flow (in steady-state) so that no streamlines seeded from one side of this surface penetrate the other side. Some work has been done in locating this feature by computing the stream function. Also it is possible to use vector field topology to find the extent of this region and then draw a series of streamlines connecting the critical points. These lines can be tessellated to create this separation surface.

These methods do not work for transient problems. Like a series of instantaneous streamlines can be misleading in unsteady flow regimes, using techniques based on streamlines will not represent the regions of older fluid. By using the concept of time, recirculations can be identified as regions of fluid that are old in comparison with the *core* flow. therefore instead of looking directly at flow topology we can calculate Residence Time. This is the Eulerian view of unsteady particle tracing (a Lagrangian operation). A simple partial differential equation can be solved on the same mesh along with the flow solver. (NOTE: This is possible when performing co-processing; the CFD solver and Residence Time calculation have similar time limit constraints.) An iso-surface can be generated through the result so that regions of old fluid can be separated from newer fluid elements.

Though the concept of Residence Time is used by process engineers (in particular injection molding) and those individuals concerned about environmental pollutants, it is from the statistical standpoint. We can not find a rigorous definition in the fluid dynamics literature.

1.4 Boundary layers

Boundary layers are features that are very important in most complex fluid flow regimes. The size and shape of the boundary layer are used to determine such values as lift and drag in external aerodynamics. For turbomachinery the size of the boundary layers determine the effective solidity. With regions of recirculation, the boundary layers determine the blockage. In all cases the boundary layer edge can be constructed as a surface (some distance away from solid walls) in 3D flows.

There have been no successes in any known work to robustly determine the surface that represents the extent of the boundary layer from traditional CFD solutions. Fundamentally, this is a very difficult problem. The edge is poorly defined numerically and is more a subtle transition that an abrupt feature.

Accurately knowing the edge of the boundary layer has many numerical benefits for the solver. Turbulence models can be more accurately applied. Grid adaptation can place nodes where they are needed. Split solvers (Euler in core flow, Navier-Stokes in boundary layers) will be more stable and accurate when the position of the edge of the boundary layer is known.

1.5 Wakes

Wakes are usually generated by the merging of boundary layers down stream from a body. Like boundary layers, these features are important for both internal and external flows. Knowing where, and under what circumstances, the wakes impinge on other bodies can have a changing effect on the structural and thermal loads experienced on those surfaces. Again, there has been no real success in finding this feature.

2 Progress Thus Far

The goal of this work is to develop a comprehensive software feature extraction tool-kit that can be used either directly with CFD-like solvers or with the results of these types of simulations (i.e. data files). The output of the feature "extractors" will be produced in such a manner that it could be rendered within most visualization systems. Much effort will be placed in further quantifying these features so that the results can be applied to grid generation (for refinement based on the features), databases, knowledge based and design systems. This requires two distinct phases; (1) the research into algorithms that will accurately and reliably find these features and (2) the design and construction of the software tool-kit.

2.1 Algorithms

2.1.1 Shocks

The procedure explained above has been re-examined. First, much effort was placed in examining algorithms that find discontinuities in scalar fields. These techniques can be thought of as the 3D analogue to the methods used in image processing. This approach failed in finding shocks for the following reasons:

- Sharpness.
  Most CFD solvers that perform differences to compute derivative and flux quantities do not suppress saw-tooth oscillations in the solution. These can become unstable in even in quiescent flow (for numerical reasons) and will blow-up in the presence of discontinuities. For this reason these CFD solvers "smooth" the flow field. This obviously reduces the ability to find sharp discontinuities since they have been removed. Even for solvers that can handle abrupt changes in the flow field, a shock will probably be smeared across 2 to 3 cells.

- Derivative quantities.
  There tends to be noise generated when derivative quantities are computed from local (cell based) operators. Using operators with larger stencils are possible in structured block meshes but difficult in unstructured grids. This noise problem is amplified when second derivatives are required.

Therefore the shock finder that requires looking for the inflection point – where the laplacian of the laplacian of pressure (the second derivative) is zero is doomed in CFD

solutions.

A shock finder has been developed that is a modification of the early work described above. For steady state solutions, the normalized pressure gradient is used instead of the speed gradient – this is less susceptible to other flow features such as boundary layers. It has been found that no thresholding is required. There is also an extension for transient solutions. See the attached document "Shock Detection from Computational Fluid Dynamics Results".

This paper discusses the shock location theory in detail and presents methods for shock classification. Details are presented for computing the shock strength and type (normal, oblique, bow and etc.). This paper also has an analysis of how to determine the shock speed for transient cases.

2.1.2 Vortex Cores

The original algorithm produces a series of disjoint line segments. When displayed, the eye puts together (or closes) a single line, for a single core, (when the strength of the core is large). This is not acceptable for off-line uses (the first problem listed above) in that it is not possible to trace the full extent of the core. This issue is now resolved. Enforcing the cell piercing to match at cell faces insures that the line segments generated will produce a contiguous core. This was first attempted via the following modification to the algorithm:

1. Compute the $\underline{V}$ (the velocity gradient tensor) at each node.
   This requires much more storage – 9 words are needed for each node in the flow field. This has the advantage that the stencil used for the operation is larger than the cell and therefore the result will be generally smoother.

2. Average the node tensors (on the face) to produce a face-based $\underline{V}$.
   This insures that the same tensor is produced for the two cells touching the face.

3. Perform the eigen-mode analysis on the face tensor.
   If the system signifies swirling flow, determine if the swirling axis cuts through the face by the scheme used in the current method. If, so mark the location on the face.

This scheme worked at the expense of memory and a much higher CPU load. Four eigen-mode calculations are required for each tetrahedron instead of just one. In general, this can be reduced to two per tetrahedron, by the additional storage of face results (about

3 words per face). Note: there are about 2 times the number of faces as cells in a tetrahedral mesh.

This was not a good result, in particular for structured blocks, where each individual hexahedron is broken up into 6 tetrahedra (5, the minimum does not promote face matching). This means that for each element in the mesh a minimum of 12 eigen-mode analyses are required.

These performance problems suggested another, related, technique:

1. Compute the $\underline{V}$ at each node.
2. Perform the eigen-mode analysis on the node tensor.
   The tensor can be overwritten with the critical point classification and the swirl axis vector for rotating flow.
3. Average the swirl axis vectors for the nodes that support the tetrahedral face.
   This should only be done if all nodes on the face indicate swirling flow. Some care needs to be taken to insure that the sense of the vectors are the same. Determine if the swirling axis cuts through the face, and if so, mark the location on the face.

For tetrahedral meshes, the reduction of compute load is by a factor of 5 to 6 over the original method (there are roughly 5.5 tetrahedra per node in 'good' unstructured grids). For structured blocks, where the number of nodes is about equal to the number of hexahedra, the number of eigen-mode analyses required is on the order of one per cell.

For coherent collected cores to be produced, all the disjoint lines are used to build threaded lists. The end points now match at tetrahedron faces. Unfortunately, do to the fact that some tetrahedra do not have 2 pierce points, special processing is required:

- 1 intersection
  The point is used as a seed point for streamline integration. This integration only persists until a face is hit (in the original cell – not the decomposed tetrahedron). If there is an intersection on that face the connection is made. If the element was a tetrahedron to begin with, or no match can be found then it is assumed that the core as either started or ended.
- 3 & 4 intersection points
  These connections are deferred. At the end, threads are put together to produce the longest (most number of points) cores. This is a recursive procedure.

All resultant core segments that have less than 4 disjoint segments are culled.

It should be noted that these situations occur because CFD results are, by their very nature, not smooth. Saw-tooth oscillations in the vector field can produce *noisy* results locally.

This new algorithm is still linear and will produce incorrect results when the flow is under 2 or more (relatively equal) rotating influences (the third problem listed above). A paper [Roth98] was presented that, on first viewing, appears to solve this problem. The authors suggest that by looking at higher-order derivatives of the velocity field one can capture curvature. They first recast the technique described above then apply the higher-order correction:

- Parallel alignment
  An intersection point on a face is where the *reduced velocity* is zero. Therefore the velocity vector is parallel to the real eigenvector of $\underline{V}$ (the velocity gradient tensor). $\vec{u}$ (the velocity vector) is an eigenvector of $\underline{V}$ and therefore a solution of $\underline{V}\vec{u} = \lambda\vec{u}$. This suggests that looking for parallel alignment is the same as the current vortex algorithm.

- Second Order
  Computing the velocity second derivatives (**T**) produces a $3x3x3$ tensor. Checking for alignment of the second derivative following a particle produces:
  $\underline{V}\underline{V}\vec{u} + \mathrm{T}\vec{u}\vec{u} = \kappa\vec{u}$

In practice, this does not seem to work. First of all, as noted above, the velocity field is not smooth. Because this technique uses the velocity directly for alignment, it produces both many false positives and misses many intersections for real CFD results. The other problem is the storage requirements. For large data sets, requiring 27 words/node of the second order tensor and 9 words/node for the first order tensor becomes prohibitive.

Further investigation is required.

2.1.3 Regions of Recirculation

The recirculation algorithm, Residence Time, described above requires close integration with the flow solver. The choice is either that solver writer completely incorporates this equation by adding one more equation to the state-vector or some co-processing system (like the visualization suite **pV3**) is used.

Obviously, the best place for this PDE is within the solver. For the second choice, an API for solving this PDE is being developed so that there is access to all of the required data. A Lax-Wendroff scheme has been implemented for time integration. Therefore if some implicit or high-order explicit time integration scheme is used for the solver care must be taken in selecting the time-step so that the solving of the Residence Time equation is stable. There is a call in the API return the maximum stable time-step.

The paper "Using Residence Time for the Extraction of Recirculation Region" (appended to this report) describes in detail both the theory and implementation of the Residence Time concept. The paper describes the current API. Also included with this progress report is the specification for the software that will de delivered at the end of the contract. This document titled "The **F**luid **F**eature **EX**traction Tool-kit" describes a slightly different interface that is consistent with the other extraction techniques.

2.1.4 Boundary layers and Wakes

As described in last years Progress Report, some progress has been made in this difficult arena. An algorithm is being constructed that will allow the use of iso-surfacing to separate the boundary layers and wakes from core flow. The method stems from the fact that these features display both rotating flow and fluid under shear stress. This is why, sometimes the vortex core technique gives false-positives for locations in boundary layers. Therefore, with a boundary layer finder we should be able to mask out these finds in the boundary layer and only display those lines that trace back from the outer flow.

To numerically define these quantities we again start with the $\underline{V}$ (the velocity gradient tensor) at each node:

- Rate of Rotation.
  This quantity is related to vorticity. A skew-symmetric tensor is produced by subtracting the transpose of $\underline{V}$ from $\underline{V}$. The result has zero on all of the diagonal terms and the off-diagonal terms are symmetric but have opposite signs across the diagonal. These values are coordinate system invariant. For this application, the norm of the upper (or lower) terms is used for the rotation scalar. This is a measure of the rate of solid-body rotation.

- Rate of Shear Stress.
  A symmetric tensor can be produced from $\underline{V}$ by adding it to its transpose. This defines the Rate of Deformation tensor. The matrix represents both the bulk and

shear stresses and is dependent on the coordinate system. To extract a single scalar that is coordinate system invariant and has the bulk terms removed it is necessary to diagonalize this tensor. The result produces a vector which signifies the 'principle axis of deformation'. By employing techniques from Solid Mechanics, the norm of the second principal invariant of the 'stress deviator' can be used as a measure of the shear and employed as the scalar.

This current scheme initially looked promising but these problems have not been resolved:

- A node based scalar function of shear and rotation has not been constructed that can based on theory.
- The value of this function will probably be dimensional, having units of inverse time (the same as shear and rotation).
  This means that the iso-surface value used to define the edge of the layer changes from case to case. This scalar needs to be multiplied by some characteristic time associated with the problem.
- The value used for the iso-surface also needs to be coupled to theory.

The lack of progress using this avenue has prompted examining another approach. For this work we start from Boundary Layer Theory.

The goal is to find a general method to calculate the displacement and momentum thickness of boundary layers near solid surfaces in the model. These quantities are less subjective measures of the boundary layer thickness. The displacement thickness is related to the blockage effect caused by the viscosity near the body, and the momentum thickness is related to the drag on the body; both of which are important to designers. It may also be possible to calculate the blockage effect of viscosity at a section in the flow passage of turbomachinery with these methods.

- Theory
  The method that is currently being investigated is to integrate the vorticity from the solid surface out to the free stream in a direction normal to the surface. The vorticity is being used as an approximation to the change in velocity normal to the wall, and is useful because it goes to zero in the free stream, allowing for a simple marking for the termination of integration. Using the vorticity eliminates the need to determine

the free stream velocity, U, and the boundary layer thickness at each point on the wall. These variables are normally used to define the displacement and momentum thickness, as shown in the following equations 1 and 2, where δ is the boundary layer thickness, δ_1 is the displacement thickness and δ_2 is the momentum thickness.

$$\delta_1 = \int_0^{\delta} (1 - \frac{u}{U}) dn \quad (1)$$

$$\delta_2 = \int_0^{\delta} \frac{u}{U}(1 - \frac{u}{U}) dn \quad (2)$$

The approximation to the displacement thickness is shown in equation 3 where ω is the vorticity, and the integration proceeds from the wall surface to where the vorticity is approximately zero.

$$\delta_1 \approx \int_0^{\omega=0} dy - \frac{\int_0^{\omega=0} u dy}{\int_0^{\omega=0} \omega dy} \quad (3)$$

The procedure that is being followed is outlined below:

1. Determine curves that are normal to the surface and do not intersect one another other. This is being done by actually solving Laplace's equation on the domain of the model, and using the gradient of this solution to define normal vectors.
2. Calculate the vorticity from the velocity components given from the CFD solution.
3. Integrate the vorticity from the surface along the normal curves until the vorticity drops below a certain threshold. This integration yields the free stream velocity at that surface point.
4. Use the free stream velocity value to calculate the displacement and momentum thickness at that point.

- Result

  The first test was to use analytic methods to determine if this method duplicated the results of Blasius for a flat plate in uniform flow at zero angle of attack. Calculating the free stream value by integrating the vorticity yields a free stream value within 1% of the actual value.

  The method was also tested by imposing a Blasius solution on a rectangular mesh and using a numerical implementation of the method to find the displacement thickness along the plate. The numerical results closely followed the Blasius solution of the boundary layer thickness quantities.

If this approach is ultimately successful, it leaves the question of how to deal with wakes. In this case there is no body to integrate from!

2.2 Software Tool-kit

The initial specification of the Application Programming Interface (API) has been completed. The document "**FX** Programmer's Guide: The **F**luid Feature **EX**traction Tool-kit" is included with this report.

The API is split into 2 basic sections:

- Support
  These are the utility and general routines that support the communication of the information that is used to determine the spatial, temporal and partitioning of the CFD data.
- Features
  These routines return the features as 3D structures and associated quantities, such as strength that may be displayed in visualization systems or used for other non-interactive applications.

3 Presentations and Publications

One third of the SIGGRAPH '98 Course #2 ("Exploring Gigabyte Data Sets in Real Time") was on this feature work. This lesson given by Robert Haimes was subtitled "Automatic Flow Feature Detection – Physics-based Extraction From Transient Computational Fluid Dynamics". There were hundreds in attendance. This same presentation has also been given at the Army Research Lab and Sandia National Lab.

Three papers have been written and submitted to the AIAA CFD Conference to be held this summer. These are appended to this report:

- Using Residence Time for the Extraction of Recirculation Regions
  Author: Robert Haimes
- Shock Detection from Computational Fluid Dynamics Results
  Authors: David Lovely and Robert Haimes
- On the Velocity Gradient Tensor and Fluid Feature Extraction
  Authors: Robert Haimes and David Kenwright

The paper "Feature Extraction form Computational Fluid Dynamics" has been submitted to the Communications of the ACM. The authors are David Kenwright and Robert Haimes. This is an overview of the work performed under this contract and at NASA Ames Research Center.

4 Technology Transfer

Individuals at Army Research Lab are currently using the test-bed for some of these algorithms in conjunction with both **Visual3** and **pV3**.

Industry and the software vendors of CFD-style scientific visualization packages have shown great interest in incorporating this work into their systems. Pratt & Whitney is willing to *beta* test the software tool-kit as it is being constructed in next years effort.

Intelligent Light (responsible for **FieldView**) and ICEM-CFD (whose visualization module is called **ICEM-Visual3** will be incorporating the fluid feature extraction tool-kit into their products.

5 Next Years Effort

5.1 Algorithms

5.1.1 Vortex Cores

The alignment of velocity and functions of the derivatives of the velocity field warrant further investigation. If what Roth & Peikert [Roth98] suggest can be made to function with *real* CFD results then the last major difficulty with the current eigen-analysis technique can be overcome.

5.1.2 Boundary layers and Wakes

The Boundary Layer Theory technique needs to be applied to a laminar and turbulent pipe flow solution for two purposes. The first is to validate the method against this exact solution, and the second is to simply extend the software tools that have been created into three dimensions. It is also of interest to determine the effective blockage through a sectional slice of the three dimensional model. Some theoretical work along these lines that show that it is possible to estimate the difference in mass flow rate through a given section with and without viscosity for the same pressure gradient. If it works, this could be used as an indicator of performance in turbomachinery design.

After this the technique will be applied to a transient tapered cylinder model as a final proof of this method.

Wakes then need to be addressed. This may involve coupling the Boundary Layer Theory technique with the shear and rotation technique. By matching results in regions where there are walls, the shear and rotation technique can be used away from the body.

5.2 Software Tool-kit

The code for the API documented in "**FX** Programmer's Guide: The **F**luid Feature **EX**traction Tool-kit" will be written. This will be callable from FORTRAN (both F77 and F90) as well as C and C++ and be accessible from all UNIX platforms and WindowsNT.

At the end of the contract the source for this work will be made freely available.

6 References

Darmofal91 David Darmofal, "Hierarchical Visualization of Three- Dimensional Vortical Flow Calculations". MIT Thesis and CFDL-TR-91-2, March 1991.

Kenwright97 David Kenwright and Robert Haimes, "Vortex Indetification – Applications in Aerodynamics". IEEE Computer Society, Visualization '97, Oct. 1997. Awarded 'Best Case-Study'.

Sujudi95 David Sujudi and Robert Haimes, "Identification of Swirling Flow in 3-D Vector Fields". AIAA Paper 95-1715, San Diego CA, June 1995.

Roth96 M. Roth and R. Peikert, "Flow Visualization for Turbomachinery Design". IEEE Computer Society, Visualization '96, Oct. 1996.

Roth98 M. Roth and R. Peikert, "A Higher-Order Method For Finding Vortex Core Lines". IEEE Computer Society, Visualization '98, Oct. 1998.

PROPOSED COST ESTIMATE
4/1/99 — 3/31/00

	Effort	Total
SALARIES & WAGES		
Principal Research Engineer (Haimes)	10.0%	9,333
Res. Support Personnel	3.3%	1,878
Res. Support Personnel	2.5%	720
Research Assistant (SM Candidate)	50%	6,258
TOTAL SALARIES & WAGES		18,189
EMPLOYEE BENEFITS (excluding UROP & RAs) @	26.7%	3,185
VACATION ACCRUAL (excluding Faculty, UROP & RAs) @	11%	1,313
OTHER COSTS		
Travel (Domestic)		1,840
Computation		1,200
Office Supplies, xerox, toll calls, postage		300
Research Assistant Tuition		12,312
TOTAL OTHER COSTS		15,652
TOTAL DIRECT COSTS		38,339
FACILITIES & ADMINISTRATIVE excluding Tuition & Equipment @	63.5%	16,528
TOTAL		54,867

FX Programmer's Guide

Rev. 0.00 (initial specification)
The **F**luid Feature **EX**traction Tool-kit

Bob Haimes
Massachusetts Institute of Technology

February 8, 1999

License

This software is being provided to you, the LICENSEE, by the Massachusetts Institute of Technology (M.I.T.) under the following license. By obtaining, using and/or copying this software, you agree that you have read, understood, and will comply with these terms and conditions:

Permission to use, copy, modify and distribute, this software and its documentation for any purpose and without fee or royalty is hereby granted, provided that you agree to comply with the following copyright notice and statements, including the disclaimer, and that the same appear on ALL copies of the software and documentation:

Copyright 1999 by the Massachusetts Institute of Technology. All rights reserved.

THIS SOFTWARE IS PROVIDED "AS IS", AND M.I.T. MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, M.I.T. MAKES NO REPRESENTATIONS OR WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE LICENSED SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

The name of the Massachusetts Institute of Technology or M.I.T. may NOT be used in advertising or publicity pertaining to distribution of the software. Title to copyright in this software and any associated documentation shall at all times remain with M.I.T., and USER agrees to preserve same.

Contents

1 Introduction **5**

2 Programming Overview **6**

2.1 Domain Decomposition 6

2.2 Node Numbering 6

2.3 Cell Numbering 7

2.4 Blanking 8

2.5 Surfaces 8

2.6 Programming Notation 9

2.7 Calling Sequences 10

3 Programmer-called subroutines **13**

3.1 FX_Init 13

3.2 FX_Update 14

3.3 FX_Free 14

4 Call-backs **15**

4.1 FXcell 15

4.2 FXcellPtr 15

4.3 FXsurface 17

4.4 FXsurfacePtr 17

4.5 FXgrid 18

4.6 FXgridPtr 18

4.7 FXblank 18

4.8 FXblankPtr 18

4.9 FXvel 19

4.10 FXvelPtr 19

4.11 FXscal 19

4.12 FXstruc 20

5 Shock Routines **21**

5.1 FX_ShockFind 21

5.2 FX_ShockSurface 21

6 Vortex Core Routine **22**

6.1 FX_VortexCore 22

7 Residence Time Routines **23**

7.1 FX_RTParams 23

7.2 FX_RTSolve 23

7.3 FX_RTTimeStep 23

7.4 FX_RTGet 23

7.5 FX_RTSurface 24

7.6 FXmodifyRT 24

8 Boundary Layer/Wake Routine **25**

8.1 FX_BLSurface 25

1 Introduction

FX is the newest in a series of graphics and visualization tools to come out of the Department of Aeronautics and Astronautics at MIT. **FX** (which stands for **F**luid Feature **EX**traction) is designed to work with the results of Computational Fluid Dynamics in either steady-state of in a co-processing transient mode. The end result is the extraction of the feature so that it can be used directly with a visualization (such as **Visual3** or **pV3**) or applied to some "off-line" procedure such as mesh enrichment.

The **FX** tool-kit can be used directly with solvers and has been designed to function in parallel/distributed environments. This has required supporting a fairly complete set of grid discretizations as well as domain decomposition (partitioning).

The Application Programming Interface (API) is split into 2 basic sections:

- Support
  These are the utility and general routines that support the communication of the information that is used to determine the spatial, temporal and partitioning of the CFD data.

- Features
  These routines return the features as 3D structures and associated quantities, such as strength that may be displayed in visualization systems or used for other non-interactive ("off-line") applications.

2 Programming Overview

Before presenting the subroutine argument lists in detail it is helpful to discuss, in general terms, the data structures which the programmer supplies to **FX**. In some cases these data structures can be taken directly from either **Visual3** or **pV3**). See the appropriate Advanced Programmer's Guide.

The programmer gives **FX** a list of unconnected cells and structured blocks. The disjoint cells are of four types; tetrahedra, pyramids, prisms and hexahedra. This element generality covers almost all data structures being used in current computational algorithms. Any special cell type which is different must be split up into some combination of these primitives by the programmer. Linear interpolation is used throughout **FX**, so high order elements must be also be subdivided so that the linear interpolation assumption is valid.

The volume(s) are defined by face matching of the elements (based on equating node numbering). Any exposed face (not shared by 2 cells) must be treated as either a boundary (a *domain surface* in **Visual3/pV3** terminology) or covered with *halo* cells. Therefore for multi-structured block cases, the surfaces that are actually inside the volume must be treated so that **FX** can patch them together.

Note: Poly-tetrahedral strips are not supported.

2.1 Domain Decomposition

The **FX** tool-kit requires the calculation of spatial derivatives. This is performed in a *finite-element* manner. If the data is not completely resident within one computer, additional information is required so that the result is consistent. For all *internal* boundaries (created by the partitioning of the data) a halo of cells is required. This halo is constructed by including all the cells that touch a node on the boundary that exist in the neighboring partition. This produces additional cells and nodes in partition. These are differentiated in the programming interface.

It should be noted for unstructured meshes that this will require more elements than those whose faces touch the boundary.

2.2 Node Numbering

The node numbering used within **FX** is local. For distributed memory cases information is required for the halo region(s). This is done by adding the nodes required to produce these cells at the end of the node space. It is the responsibility of the calling application to do any message passing and node number re-mapping so that the halo information is correct.

The node numbering used differentiates between the nodes in the non-block regions (formed by the disjoint cells), the structured blocks, and the halo regions. Figure 1 shows a schematic of the node space. **knode** is the number of nodes for the non-block grid. Each structured block (m) adds $NI_m * NJ_m * NK_m$ nodes to the node space (where NI, NJ and NK are the number of nodes in each direction). The node numbering within the block follows the memory storage, that is, (i,j,k) in FORTRAN and [k][j][i] in C. The **FX** node number = $base + i + (j-1) * NI_m + (k-1) * NI_m * NJ_m$. Where *base* is **knode** for the first block, and **knode** plus the number of nodes in the first block for the second, and etc. Note: all indices start at 1.

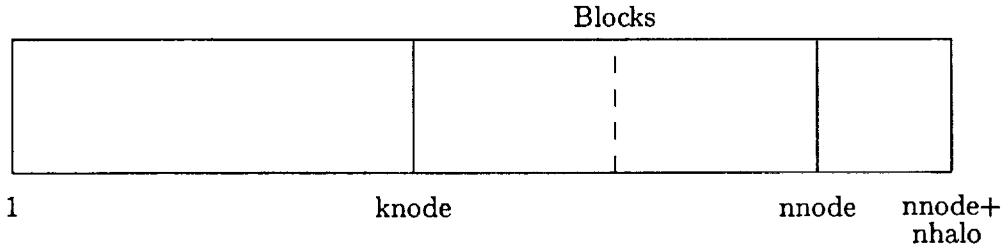


Figure 1: Node Space

nhalo is the number of nodes added to the domain for the halo elements. This is zero for a case with a single partition.

2.3 Cell Numbering

The non-block cell types may contain nodes from the non-block and the structured block volumes but not from the halo nodes. The cell numbering used within **FX** orders the cells by type. Figure 2 shows a schematic of the cell space. The programmer explicitly defines all non-block cells by the call FXcell or provides the pointers by the call-back FXcellPtr. Again the cells within the blocks are defined by the block size. Each structured block (m) adds $(NI_m - 1) * (NJ_m - 1) * (NK_m - 1)$ cells to cell space. The cell numbering within the block follows the memory storage so that a **FX** cell number = $base + i + (j-1) * (NI_m - 1) + (k-1) * (NI_m - 1) * (NJ_m - 1)$. Where *base* is **nTets+nPyra+ nPrism+nHexa** for the first block, and this value plus the number of cells in the first block for the second, and etc.

Note: i goes from 1 to $NI_m - 1$, j goes from 1 to $NJ_m - 1$, and k goes from 1 to $NK_m - 1$.

There are individual structures for each element type. This provides compatibility with both **Visual3** and **pV3** and minimizes the amount of memory required to fully describe complex gridding. The halo cells are handled in a different manner. Each cell is disjoint

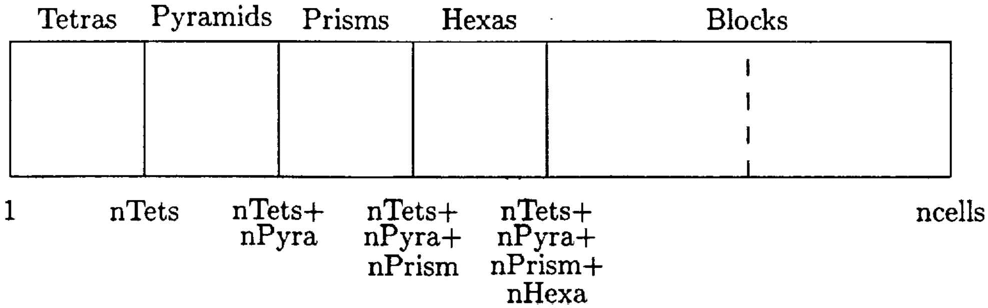


Figure 2: Cell Space

(either a tetrahedron, pyramid, prism or hexahedron) and is stored in the same structure. Node indices that make up the halo cells must contain at least one non-halo node and at least one halo node (index > **nnode**). The exception to this is when blocks are patched or for C meshes where all node indices can be from the non-halos.

Again, the numbering is local for multiple processor applications.

2.4 Blanking

Blanking is an option (see the description of FX_Init) and only used with structured blocks to indicate that some region of the block is 'turned off'. A part of a block is deactivated by flagging the appropriate nodes as invalid. This is done by an IBLANK array. An invalid node is never used.

When blanking is used, all the nodes (**nnode** − **knode**) in the structured block space are given a value; zero corresponds to an invalid mesh point, any non-zero value indicates an existing node point.

2.5 Surfaces

In principle, all exposed facets could be grouped together to form one bounding surface. However, in many applications it is more useful to split the bounding surface into a number of pieces, referred to in **Visual3** and **pV3** documentation as *domain surfaces*. For example, the outer bounding surface of a calculation of airflow past a half-aircraft (using symmetry to reduce the computation) would typically be split into four pieces, the inflow boundary, the outflow, the symmetry plane, the aircraft. The Residence Time functions of **FX** require information on which exposed facets to apply what boundary condition. These must be classified as either inflow, outflow, periodic/equivalence and no-flux (wall).

Internal surfaces are those that get created when the computational domain is subdivided and placed in multiple machines. These artificial surfaces are handled by the halo

elements so it appears to **FX** that they do not exist.

2.6 Programming Notation

FX was designed to be accessible from both FORTRAN and C. FORTRAN is more restrictive in argument passing and naming, therefore it has shaped the programming interface. The routine descriptions in this guide are from the C programmer's point of view. But because FORTRAN is supported with the same API all routine arguments are pass by reference. It is assumed that a routine's argument is not modified unless documented as such.

For IBM and HP ports, all **FX** entry points are the FORTRAN names in lower-case. On all other platforms except the CRAY and WindowsNT, external entries are lower-case with an underscore ('_') appended to the end. CRAY entry points are upper-case with no appended underscores. WindowsNT entry points must be declared as **__stdcall** and are upper-case with no appended underscores. See the file 'FX.h' in the distribution for a method to avoid these problems.

Consistent with the **Visual3** and **pV3** naming conventions, the routines that are part of the **FX** tool-kit are prefixed with 'FX_', those that are supplied by the programmer start with 'FX' and do not have an underscore as the next character. There are a number of pairs of these programmer-supplied call-backs. These pairs exist in order in conserve memory, that is if the programmer already has the data in the proper form then the pointer to that data is passed to **FX**. Otherwise **FX** allocates the appropriate memory and it is the responsibility of the call-back to fill that structure. Only one of the pair will be called during the **FX** session. The naming convention is the routine that returns the pointer has a name with the 'PTR' suffix.

2.7 Calling Sequences

The **FX** tool-kit supports steady-state as well as three types of unsteadiness. In a multiple partition simulation, each sub-domain can have a different transient mode. Each mode causes a different internal calling sequence. In general, the application must first call FX_Init to initialize the **FX** system and then call FX_Update after every time the solution space has been updated. A schematic of a typical CFD solvers coupling with FX can be seen in Figure 3. The name FX_extract is an indication of any series of *underscore* routines documented in Sections 5 to 8.

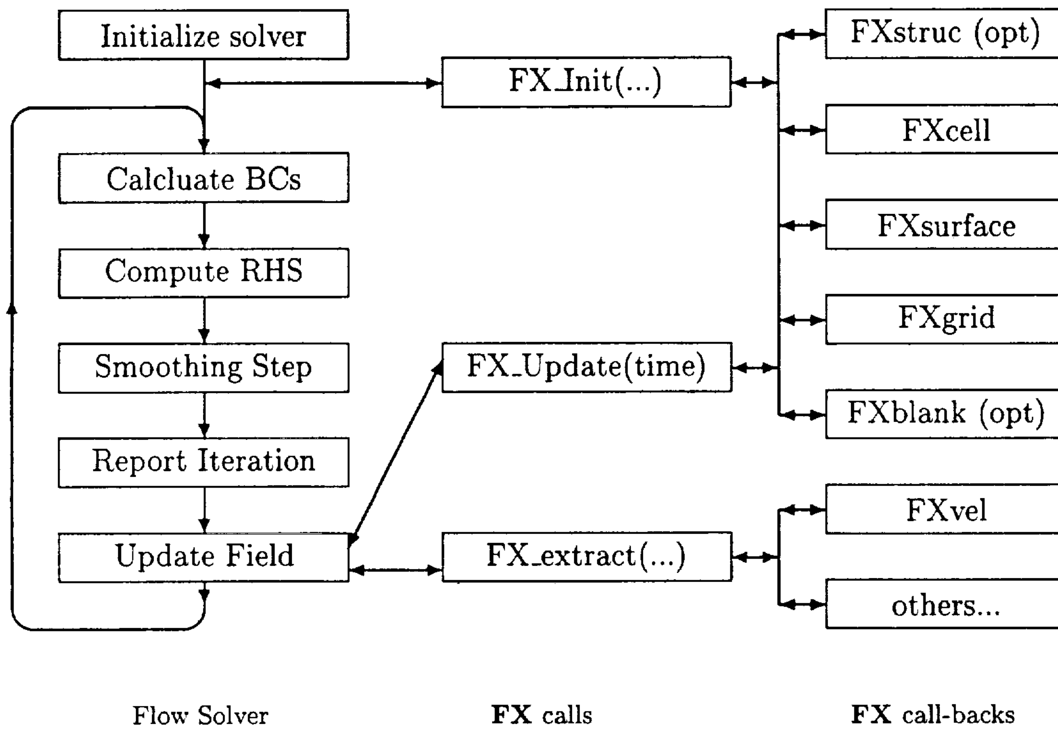


Figure 3: Co-processing Calling sequence

- Steady-State

Call	Calls in Sequence
FX_Init	FXcell or FXcellPtr (optional)
	FXsurface or FXsurfacePtr
	FXgrid or FXgridPtr
	FXBlank or FXblankPtr (optional)
FX_Update	NOT required
FX_extract	FXvel or FXvelPtr
	other call-backs as needed

- Data Unsteady

Call	Calls in Sequence
FX_Init	FXcell or FXcellPtr (optional)
	FXsurface or FXsurfacePtr
	FXgrid or FXgridPtr
	FXBlank or FXblankPtr (optional)
FX_Update	NONE
FX_extract	FXvel or FXvelPtr
	other call-backs as needed

- Grid Unsteady

Call	Calls in Sequence
FX_Init	FXcell or FXcellPtr (optional)
	FXsurface or FXsurfacePtr
FX_Update	FXgrid or FXgridPtr
	FXBlank or FXblankPtr (optional)
FX_extract	FXvel or FXvelPtr
	other call-backs as needed

- Structure Unsteady

Call	Calls in Sequence
FX_Init	NONE
FX_Update	FXstruc
	FXcell or FXcellPtr (optional)
	FXsurface or FXsurfacePtr
	FXgrid or FXgridPtr
	FXBlank or FXblankPtr (optional)
FX_extract	FXvel or FXvelPtr
	other call-backs as needed

3 Programmer-called subroutines

3.1 FX_Init

FX_INIT(IOPT, KNODE, NHALO, NTETS, NPYRA, NPRISM, NHEXA, NBLOCK, BLOCKS, NHCELL, NSURF, NBC, FLAGS)

This subroutine initializes the **FX** tool-kit. Calling this routine defines the type of case and the sizes of various parameters having to do with the volume discretization. This calling sequence also defines how and which call-backs are invoked so that **FX** can get the required data. This routine must only be called once.

int *IOPT	Unsteady control parameter
	IOPT=0 steady grid and data
	IOPT=1 steady grid and unsteady data
	IOPT=2 unsteady grid and data
	IOPT=3 structure unsteady
int *KNODE	Number of non-block nodes
int *NHALO	Number of halo nodes
int *NTETS	Number of tetrahedra
int *NPYRA	Number of pyramids
int *NPRISM	Number of prisms
int *NHEXA	Number of hexahedra
int *NBLOCK	Number of structured blocks
int BLOCKS[][6]	Structured block definitions:
	BLOCKS[m][0] = NI
	BLOCKS[m][1] = NJ
	BLOCKS[m][2] = NK
	BLOCKS[m][3] = cell number that terminates the block
	BLOCKS[m][4] = node number that terminates the block
	BLOCKS[m][5] = Not used by **FX**
int *NHCELL	Number of halo elements
int *NSURF	Number of domain surface facets
int *NBC	Number of domain surface groups (boundary conditions)

int *FLAGS — The call mask:

bit 0 – 1/0 = 0 - call FXcell for the disjoint cell data, 1 - call FXcellPtr for the disjoint information.

bit 1 – 2/0 = 0 - call FXsurface for the Boundary Condition data, 2 - call FXsurfacePtr for the Boundary Conditions.

bit 2 – 4/0 = 0 - call FXgrid for coordinates, 4 - call FXgridPtr for using the pointer.

bit 3 – 8/0 = 0 - call FXvect for the flow vector field, 8 - call FXvectPtr for using a pointer.

bit 4 – 16/0 = 0 - no Blanking, 16 - Blanking.

bit 5 – 32/0 = 0 - FXblank is called for Blanking, 32 - FXBlankPtr iscalled.

Note:
For structure unsteady cases ($IOPT = 3$), the parameters that describe the sizes of the node and cell space should be a good guess at the sizes used during the simulation. For structured block cases, NBLOCK must be the maximum number of blocks for the run. The current sizes are set by a call to FXstruc from within FX_Update.

3.2 FX_Update

FX_UPDATE(TIME)

This subroutine must be called after the solver has updated the solution space. This is when all communication between any partitions is complete including the messages required to transmit the halo data. The call to this routine is not needed if $IOPT = 0$.

float *TIME — The current simulation time.

3.3 FX_Free

FX_FREE(PTR)

This function is equivalent to the C routine 'free'. It deallocates a block of memory. NOTE: Use this utility routine to free up blocks of that have been allocated by **FX** and returned when they are no longer needed. These pointers are labeled as freeable in the routine definition.

void **PTR — The address of the memory block.

4 Call-backs

4.1 FXcell

FXCELL(TETS, PYRA, PRISM, HEXA, HCELL)

This subroutine supplies **FX** with the grid data structure. It is not required for a grid that contains only structured blocks and no halo cells.

int TETS[NTETS][4]	Node indices for tetrahedral cells (filled)
int PYRA[NPYRA][5]	Node indices for pyramid cells (filled)
int PRISM[NPRISM][6]	Node indices for prism cells (filled)
int HEXA[NHEXA][8]	Node indices for hexahedral cells (filled)
int HCELL[NHCELL][9]	Halo cell descriptions (filled)

HCELL[m][0-7] = Node indices for the cell

HCELL[m][8] = 1 - tetrahedron, 2 - pyramid, 3 - prism, 4 - hexahedron

The correct order for numbering nodes for the four disjoint cell types is shown in Fig. 4.

4.2 FXcellPtr

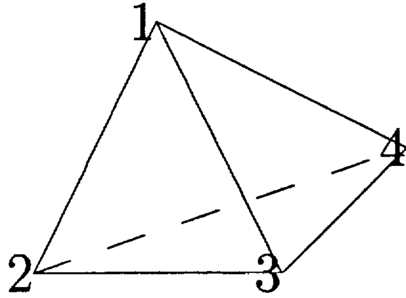
FXCELLPTR(PTETS, PPYRA, PPRISM, PHEXA, HCELL)

This subroutine supplies **FX** with the pointers to grid data structure. It is not required for a grid that contains only structured blocks and no halo cells.

int **PTETS	Pointer to node indices for tetrahedral cells (returned)
int **PPYRA	Pointer to node indices for pyramid cells (returned)
int **PPRISM	Pointer to node indices for prism cells (returned)
int **PHEXA	Pointer to node indices for hexahedral cells (returned)
int HCELL[NHCELL][9]	Halo cell descriptions (filled)

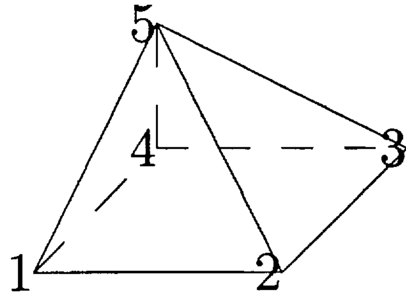
HCELL[m][0-7] = Node indices for the cell

HCELL[m][8] = 1 - tetrahedron, 2 - pyramid, 3 - prism, 4 - hexahedron



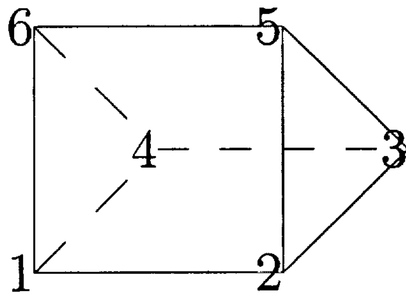
face	nodes
1	1 2 3
2	2 3 4
3	3 4 1
4	4 1 2

Tetrahedron



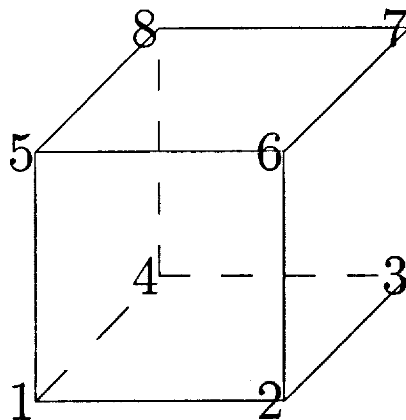
face	nodes
1	1 2 3 4
2	2 3 5
3	3 4 5
4	4 5 1
5	5 1 2

Pyramid



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

Prism



face	nodes
1	1 2 3 4
2	2 3 7 6
3	3 4 8 7
4	4 8 5 1
5	5 6 7 8
6	6 5 1 2

Hexahedron

Figure 4: Disjoint cell types and node/face numbering

4.3 FXsurface

FXSURFACE(NSURF, SCEL)

This subroutine supplies **FX** with the surface data structure. This specifies that these are exposed facets and indicates the type of boundary condition to apply.

int NSURF[NBC][2]	NSURF[m][0] is the pointer to the end of domain boundary group n, i.e. it contains the index to the last entry in SCEL for that group. NSURF[m][1] is the boundary type: **1** inflow **2** outflow **3** wall **4** wall (slip) **5** symmetry **6** nothing – extrapolate
int SCEL[KSURF][4]	node numbers for surface faces. For quadrilateral faces SCEL must be ordered clockwise or counter-clockwise; for triangular faces, SCEL[m][3] must be set to zero. (filled)

Note:

The correct order for numbering faces for the four disjoint cell types is shown in Fig. 4. For structured blocks; face #1 is for exposed cells with cell index $k = 1$, face #2 is for $i = NI_m - 1$, face #3 is for cells with $j = NJ_m - 1$, face #4 is for $i = 1$, face #5 is associated with $k = NK_m - 1$, and face #6 is for $j = 1$.

4.4 FXsurfacePtr

FXSURFACEPTR(NSURF, PSCEL)

This subroutine supplies **FX** with the surface data pointer. This specifies that these are exposed facets and indicates the type of boundary condition to apply.

int NSURF[NBC][2]	NSURF[m][0] is the pointer to the end of domain boundary group n, i.e. it contains the index to the last entry in SCEL for that group. NSURF[m][1] is the boundary type.
int **PSCEL	pointer to the structure containing node numbers for surface faces (returned)

4.5 FXgrid

FXGRID(XYZ,HXYZ)

This subroutine supplies **FX** with the grid coordinates for all of the nodes.

float XYZ[NNODE][3]	(x, y, z)-coordinates of grid nodes (filled)
float HXYZ[NHALO][3]	(x, y, z)-coordinates of halo grid nodes (filled)

4.6 FXgridPtr

FXGRIDPTR(PXYZ,HXYZ)

This subroutine supplies **FX** with the pointer to the grid coordinates for all of the nodes.

float **PXYZ	the pointer to the structure containing (x, y, z)-coordinates of grid nodes (returned)
float HXYZ[NHALO][3]	(x, y, z)-coordinates of halo grid nodes (filled)

4.7 FXblank

FXBLANK(IBLANK)

This subroutine supplies **FX** with blanking data. Required for bit 4 on and bit 5 off in FLAGS (of FX_Init).

int IBLANK[NNODE-KNODE]	Blanking data (filled):
	$= 0$ off, invalid node
	$\neq 0$ on

4.8 FXblankPtr

FXBLANKPTR(PIBLANK)

This subroutine supplies **FX** with a pointer to the blanking data. Required for bit 4 on and bit 5 on in FLAGS (of FX_Init).

int **PIBLANK	pointer to blanking data (returned)

4.9 FXvel

FXVEL(V,HV)

This subroutine supplies **FX** with the velocity field.

float V[NNODE][3]	Velocity function values (Vx, Vy, Vz) (filled)
float HV[NHALO][3]	Halo velocity function values (Vx, Vy, Vz) (filled)

4.10 FXvelPtr

FXVELPTR(PV,HV)

This subroutine supplies **FX** with the pointer to the velocity field.

float **PV	Pointer to the Velocity structure (returned)
float HV[NHALO][3]	Halo velocity function values (Vx, Vy, Vz) (filled)

4.11 FXscal

FXSCAL(TYPE,S,HS)

This subroutine supplies **FX** with the specified scalar field.

int TYPE	Scalar field indicator
float S[NNODE]	Scalar functional values based on TYPE (filled):
	TYPE = 1 - density
	TYPE = 2 - pressure
	TYPE = 3 - Mach number
	TYPE = 4 - Total viscosity (laminar and turbulent)
	TYPE = 5 - Enthalpy
float HS[NHALO]	Halo scalar functional values based on TYPE

4.12 FXstruc

FXSTRUC(KNODE, NHALO, NTETS, NPYRA, NPRISM, NHEXA, NBLOCK, BLOCKS, NHCELL, NSURF, NBC)

This subroutine is required for structure unsteady cases ($IOPT = 3$) only. This routine supplies the sizes of the current state of the problem.

int *KNODE	Number of non-block nodes / static flag
int *NHALO	Number of halo nodes
int *NTETS	Number of tetrahedra
int *NPYRA	Number of pyramids
int *NPRISM	Number of prisms
int *NHEXA	Number of hexahedra
int *NBLOCK	Number of structured blocks
int BLOCKS[][6]	Structured block definitions
int *NHCELL	Number of halo elements
int *NSURF	Number of domain surface facets
int *NBC	Number of domain surface groups (boundary conditions)

Note:
If KNODE is -1 that is a special flag to indicate that the structure has NOT changed for this iteration. With this flag set, no other parameters should be modified, in that **FX** reverts to the grid unsteady calling sequence.

5 Shock Routines

5.1 FX_ShockFind

FX_SHOCKFIND(TEST)

This subroutine returns the result of the shock test function.

float TEST[NNODE]	Any value greater than 1.0 is an indication that the node is in a shock region.

5.2 FX_ShockSurface

FX_SHOCKSURFACE(TEST, NSPTS, PSXYZ, NSTRIS, PSTRIS, PSCELL)

This subroutine takes the shock test function, generates and returns the surface(s) at the value 1.0. The surface(s) can be constructed from the triangle indices (bias 1) into the shock nodes pointed to by PSXYZ.

float TEST[NNODE]	This must be the data returned by FX_ShockFind.
int *NSPTS	The number of points that support the shock surface (returned)
float **PSXYZ	Pointer to the block of memory (freeable) that contains the coordinates (returned) The memory block is of the form SXYZ[NSPTS][3].
int *NSTRIS	The number of triangles that make up the surface (returned)
int **PSTRIS	Pointer to the block of memory (freeable) that contains the triangle indices (returned) The memory block is of the form STRIS[NSTRIS][3].
int **PSCELL	Pointer to the block of memory (freeable) that contains the cell indices for the triangle (returned) The memory block is of the form SCELL[NSTRIS].

6 Vortex Core Routine

6.1 FX_VortexCore

FX_VORTEXCORE(NVCSEG, PVCSEG, PVCXYZ, PVCCELL, PVCSTREN)
This routine returns the vortices found in the domain. They are processed as a number of segments each with a particular length.

int *NVCSEG	The number of vortex core segments (returned)
int **PVCSEG	Pointer to the block of memory (freeable) that contains the core end point indices (returned) The memory block is of the form VCSEG[NVCSEG].
float **PVCXYZ	Pointer to the block of memory (freeable) that contains the vortex core points for all segments (returned) The memory block is of the form VCXYZ[VCSEG[NVCSEG-1]][3].
int **PVCCELL	Pointer to the block of memory (freeable) that contains the cell indices for the vortex core points (returned) The first value in a segment refers to the cell that contains the first 2 points. Therefore the last cell value in a segment does not contain any valid data. The memory block is of the form VCCELL[VCSEG[NVCSEG-1]].
float **PVCSTREN	Pointer to the block of memory (freeable) that contains the vortex core strength (returned) The memory block is of the form VCSTREN[VCSEG[NVCSEG-1]].

7 Residence Time Routines

7.1 FX_RTParams

FX_RTPARAMS(RTTYPE, SM2, SM4, KAPPA)

This routine must be called before any other residence time functions. It is best to put this call after FX_Init when computing residence time. All parameters are input.

int *RTTYPE	0 to 3 for inviscid incompressible, viscous compressible, constant viscosity and density and inviscid compressible, respectively.
float *SM2	second-difference smoothing coefficient (σ_2).
float *SM4	fourth-difference smoothing coefficient (σ_4).
float *KAPPA	$\kappa = \frac{\mu}{\rho}$, required for RTYPE = 2 only.

7.2 FX_RTSolve

FX_RTSOLVE()

This routine must be called after FX_Update to integrate the residence time equation for the time-step.

No Arguments

7.3 FX_RTTimeStep

FX_RTTIMESTEP(MAXDT)

This routine can be called to get the current maximum delta-time that may be used to insure stability. The residence time equation has less of a time step constraint than either the Euler of Navier-Stokes equations, so this is not required for co-processing with explicit solvers. This call may be required when using residence time integration with steady-state solutions.

float *MAXDT	The maximum delta-time that is acceptable.

7.4 FX_RTGet

FX_RTGET(RT)

This subroutine returns the result of the shock test function.

float RT[NNODE]	The residence time for each node in the domain.

7.5 FX_RTSurface

FX_RTSURFACE(RT, RTV, NRTPT, PRTXYZ, NRTTRI, PRTTRI, PRTCELL)

This subroutine takes the residence time values, generates and returns the surface(s) at the value RTV. The surface(s) can be constructed from the triangle indices (bias 1) into the residence time nodes pointed to by PRTXYZ.

float RT[NNODE]	This must be the data returned by FX_RTGet.
float *RTV	This is the residence time value used to generate the surface.
int *NRTPT	The number of points that support the residence time surface (returned)
float **PRTXYZ	Pointer to the block of memory (freeable) that contains the coordinates (returned) The memory block is of the form RTXYZ[NRTPT][3].
int *NRTTRI	The number of triangles that make up the surface (returned)
int **PRTTRI	Pointer to the block of memory (freeable) that contains the triangle indices (returned) The memory block is of the form RTTRI[NRTTRI][3].
int **PRTCELL	Pointer to the block of memory (freeable) that contains the cell indices for the triangle (returned) The memory block is of the form RTCELL[NRTTRI].

7.6 FXmodifyRT

FXMODIFYRT(RT, DRT)

This optional call-back is invoked from FX_RTSolve and exposes both the internal array of residence time values and the deltas to be applied. This is called just before the values are updated. FXmodifyRT allows the modification of either RT or DRT directly. This is required for special boundary conditions, such as moving interfaces or others not supported.

float RT[NNODE]	Node based residence time values.
float DRT[NNODE]	Node based updates of the residence time values.

8 Boundary Layer/Wake Routine

8.1 FX_BLSurface

FX_BLSURFACE(NBLPTS, PBLXYZ, PBLD, NBLTRIS, PBLTRIS, PBLCELL)

This subroutine returns the boundary layer and wake surfaces found with the domain. The surface(s) can be reconstructed from the triangle indices (bias 1) into the BL nodes pointed to by PBLXYZ.

int *NBLPTS	The number of points that support the boundary layers (returned)
float **PBLXYZ	Pointer to the block of memory (freeable) that contains the coordinates (returned) The memory block is of the form BLXYZ[NBLPTS][3].
float **PBLD	Pointer to the block of memory (freeable) that contains the thickness – a negative value is the indication of a wake (returned) The memory block is of the form BLD[NBLPTS].
int *NBLTRIS	The number of triangles that make up the boundary layer(s) (returned)
int **PBLTRIS	Pointer to the block of memory (freeable) that contains the triangle indices (returned) The memory block is of the form BLTRIS[NBLTRIS][3].
int **PBLCELL	Pointer to the block of memory (freeable) that contains the cell indices for the triangle (returned) The memory block is of the form BLCELL[NBLTRIS].

Using Residence Time for the Extraction of Recirculation Regions

Robert Haimes*
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology, Cambridge, MA 02139
haimes@orville.mit.edu

This paper introduces the concept of residence-time, from the Eulerian view point, in a rigorous manner. The equations for various flow regimes are derived and a numerical solver is introduced based on Lax-Wendroff integration. An implementation is discussed that allows the coupling of this solver to any explicit CFD code. Examples of this concept are shown for extracting recirculation regions by segregating old fluid from fluid that has not been in the simulation for much time. The comparison of iso-surfaces generated using this procedure and separation surfaces are examined.

Introduction

In the past, feature extraction and identification were interesting concepts, but not required to understand the underlying physics of a steady flow field. This is because the results of the more traditional tools like iso-surfaces, cuts and streamlines were more interactive and easily abstracted so they could be represented to the investigator. These tools worked and properly conveyed the collected information at the expense of much interaction. For unsteady flow-fields, the investigator does not have the luxury of spending time scanning only one "snap-shot" of the simulation. Automated assistance is required in pointing out areas of potential interest contained within the flow.

Automated feature detection and identification procedures are being developed for the examination of 3D transient simulations. This software tool-kit will allow for the post-processing or co-processing visualization of Computational Fluid Dynamics results where the features are displayed in a manner that physically makes sense. Also, these techniques will allow "off-line" procedures like grid adaptation and design optimization to use the physics found in the flow-field to perform the desired task.

This paper discusses a technique that locates regions of recirculation in both steady-state and transient solutions.

Flow separation represents interesting, and sometimes important, features in many flow regimes. In combustion, swirling flow is used to enhance mixing but can also cause isolated regions of flow that do not quickly leave the system. These regions may be high-temperature and therefore can have a negative impact on the lifetime of the unit. Similarly, in turbomachinery, separated flows are associated with extremely hot regions where high-speed flow exiting the combustor has stagnated. These hot spots are undesirable since the allowable operating stress of the turbine blades is closely related to temperature. In flow over a wing, the adverse pressure gradient on the wing upper surface can lead to separated flow casuing a drastic loss of lift (or stalling) and a significant increase in pressure drag.

The importance of separated flow motivates the development of a tool which can automatically locate these regions. Ideally for steady-state flows, this tool can operate directly on the vector field imported from the flow solution. For transient simulations each time slice of unsteady data should be all that is required, without other types of data or information from other time levels. This reduces the amount of memory required.

Individuals investigating the results of CFD simulations have historically seeded streamlines (in steady-state flows) to determine where there are regions of recirculation. With a numerically accurate streamline integration scheme, recirculation is found when a streamline is "trapped" in the flow-field. In this case the streamline, going up-stream or down-stream from a point within the region, does not leave the region. The boundary of this region is a surface – the separation surface. There are a number of problems in automating this interactive procedure:

- Locating the region. Automatically seeding streamline to find these regions is difficult, indeed. Any streamline started outside the region will not enter. Therefore, one can not use the in-flow of the domain as a starting point. One would need to seed large numbers of streamlines so that every cell in the mesh is touched to insure that the region(s) are found.

- Cost. Clearly, the cost of such a procedure would be prohibitive. There is also the issue of stopping

*Principal Research Engineer

Copyright © 1999 by Robert Haimes. Published by the American Institute of Aeronautics and Astronautics, Inc. with permission.

the integration once a point inside the region is found.

- Constructing the surface. It is not obvious how one takes the outer "husk" of the trapped streamlines and then construct the separation surface.

The discussion up to this point assumes steady-state flow. It is unclear how a recirculation region should be defined in unsteady flows since a region that appears to be recirculating at an instantaneous time slice might actually be moving with the flow as time progresses. It is well known that examining streamlines to understand transient flows can be misleading. Streaklines need to be used.

Helman and Hesselink[1,2] have developed a visualization scheme for generating separation surfaces using only the solution's vector field. The scheme starts by finding the critical points on the surface of the object. Streamlines are integrated along the principal directions of certain classes of critical points and then linked to the critical points to produce a 2-D skeleton of the flow topology near the object. Streamlines are integrated out to the external flow starting from points along certain curves in the skeleton. These streamlines are then tessellated to generate the separation surfaces. With this approach, difficulties might be encountered in integrating streamlines from critical points, and also in finding separated regions that are not attached to an object.

Sujudi[3] attempted an eigen-analysis technique looking for topology where the flow is diverging locally from a plane. This was done after the success of finding vortices by looking for swirling flow, also see Sujudi.[4] This technique was determined to be unreliable. Critical point theory only provides a single classification based on the strongest local topology. In most separated flow regimes there are areas where the swirl component is much greater than the diverging topology. This can overwhelm the ability to locate the separation surface. But, Kenwright[5] has had some success in using 2D critical points for finding surface separation and attachment lines. Both the vortex core finder[4,6] and the finder of surface separation and attachment line work well in transient regimes.

This paper discusses a method that attempts to stay within the streamline/streakline definition of a separated region but applies a transformation from Lagrangian (moving with the massless particle) to an Eulerian point of view (fixed within a grid and watching the fluid flow past). The concept is *residence-time*. Essentially, one computes the amount of time the fluid has been in (or in residence within) the domain. Residence-time zero is defined as the time when the simulation starts. Most of the fluid within a separation region stays within that region for a considerable amount of time. Thus, a common feature of separation region is that the residence-time of the fluid within it is much larger than that of the surrounding fluid. An iso-surface can then be used to distinguish this region. The value of the iso-surface can be selected knowing the characteristic time for the system. Therefore, residence-time can easily produce the separation surface for either steady-state or transient simulations.

This can be thought of as a streakline process where, periodically at the in-flow, particles are seeded and marked with their start-time. At some later time the particles with the current time subtracted from the start-time are segregated. The advantage of residence-time is that you get complete grid coverage and the surface generation is trivial.

The visualization test-bed used in this paper is **pV3**,[7,8] a distributed system developed at MIT. **pV3** is ideal for this work, in that, it is designed for co-processing. Co-processing allows the investigator to visualize the data as it is being computed by the solver. Distributed computing decomposes the computational domain into 2 or more sub-domains, which can be processed across a network of workstation(s) and other types of compute servers. The algorithm used in computing residence-time has been developed to co-exist with the parallel capability of **pV3**.

Theory

The residence-time of a volume of inviscid fluid is defined by

$$\frac{\mathbf{D}\mathcal{T}}{\mathbf{Dt}} = 1 \qquad (1)$$

where $\mathcal{T}$ denotes residence-time. This is similar to conservation of mass, but with a source term. It should be noted that this author has not seen any references to this definition in the open literature.

Since $\frac{\mathbf{D}}{\mathbf{Dt}} \equiv \frac{\partial}{\partial t} + \vec{u} \cdot \nabla$, then Equation (1) becomes

$$\frac{\partial \mathcal{T}}{\partial t} + \vec{u} \cdot \nabla \mathcal{T} = 1 \qquad (2)$$

where $\vec{u}$ is the velocity vector.

Since the time when the residence-time calculation starts is defined as time zero, then initial the condition is

$$\mathcal{T}(x, y, z) = 0. \qquad (3)$$

At in-flow boundaries, new fluid is entering. By definition, this fluid has zero residence-time. Therefore, the boundary condition is

$$\mathcal{T}(x, y, z) = 0 \qquad at\ in - flow. \qquad (4)$$

To obtain the conservative form of Equation (2) for incompressible flows, this can be rewritten as

$$\frac{\partial \mathcal{T}}{\partial t} + \vec{u} \cdot \nabla \mathcal{T} + \mathcal{T} \nabla \cdot \vec{u} = 1 + \mathcal{T} \nabla \cdot \vec{u}$$
$$\frac{\partial \mathcal{T}}{\partial t} + \nabla \cdot (\mathcal{T} \vec{u}) = 1 + \mathcal{T} \nabla \cdot \vec{u}.$$

And since $\nabla \cdot \vec{u} = 0$ for incompressible flows, then

$$\frac{\partial \mathcal{T}}{\partial t} + \nabla \cdot (\mathcal{T}\vec{u}) = 1. \quad (5)$$

Equation (5) reflects the residence-time in an analogous manner to streaklines. Here the result is only a function of the velocity field. Additional realism may be applied to the formulation. The conservative form for compressible flows can be obtained by rewriting Equation (2) as

$$\rho\frac{\partial \mathcal{T}}{\partial t} + \rho\vec{u} \cdot \nabla\mathcal{T} = \rho$$

where ρ denotes density. And since the conservation of mass equation for compressible flows is

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho\vec{u}) = 0$$

then

$$\rho\frac{\partial \mathcal{T}}{\partial t} + \rho\vec{u} \cdot \nabla\mathcal{T} + \mathcal{T}\left[\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho\vec{u})\right] = \rho$$
$$\frac{\partial(\rho\mathcal{T})}{\partial t} + \nabla \cdot (\rho\mathcal{T}\vec{u}) = \rho. \quad (6)$$

The effect of viscosity on $\mathcal{T}$ is the same as its effect on velocity in that the same mechanism is at work. This is justified from the statistical mechanics viewpoint. At the molecular level, viscous action mixes individual molecules (each with it's own residence-time), so the average local residence-time is effected by viscosity.[9] The viscous term for Equation (6) mimics the viscous term in the conservation of momentum equation of the Navier-Stokes equations. Thus, the residence-time equation for a viscous compressible flow is

$$\frac{\partial(\rho\mathcal{T})}{\partial t} + \nabla \cdot (\rho\mathcal{T}\vec{u}) = \rho + \nabla \cdot (\mu\nabla\mathcal{T})$$

or

$$\frac{\partial(\rho\mathcal{T})}{\partial t} + \nabla \cdot (\rho\mathcal{T}\vec{u} - \mu\nabla\mathcal{T}) = \rho \quad (7)$$

where μ is the total viscosity, which accounts for both laminar and turbulent components.

For the special case where the flow has both a constant viscosity and density, Equation (7) reduces to

$$\frac{\partial \mathcal{T}}{\partial t} + \nabla \cdot (\mathcal{T}\vec{u}) = 1 + \kappa\nabla^2\mathcal{T}$$
$$\frac{\partial \mathcal{T}}{\partial t} + \nabla \cdot (\mathcal{T}\vec{u} - \kappa\nabla\mathcal{T}) = 1 \quad (8)$$

where $\kappa = \frac{\mu}{\rho}$ which is a constant.

All the conservative forms of the residence-time equations [Equations (5), (6), (7), and (8)] can be expressed as

$$\frac{\partial U}{\partial t} + \frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} + \frac{\partial H}{\partial z} = Q \quad (9)$$

For incompressible inviscid flow

$$U = \mathcal{T}, \quad F = \mathcal{T}u, \quad G = \mathcal{T}v, \quad H = \mathcal{T}w, \quad Q = 1,$$

where u, v, and w are the components of $\vec{u}$. For compressible inviscid flow

$$U = \rho\mathcal{T}, \quad F = \rho\mathcal{T}u, \quad G = \rho\mathcal{T}v, \quad H = \rho\mathcal{T}w, \quad Q = \rho,$$

for compressible viscous flow

$$U = \rho\mathcal{T}, \quad F = \rho\mathcal{T}u - \mu\frac{\partial \mathcal{T}}{\partial x}, \quad G = \rho\mathcal{T}v - \mu\frac{\partial \mathcal{T}}{\partial y}, \quad H = \rho\mathcal{T}w - \mu\frac{\partial \mathcal{T}}{\partial z}, \quad Q = \rho,$$

and for a flow with constant viscosity and density

$$U = \mathcal{T}, \quad F = \mathcal{T}u - \kappa\frac{\partial \mathcal{T}}{\partial x}, \quad G = \mathcal{T}v - \kappa\frac{\partial \mathcal{T}}{\partial y}, \quad H = \mathcal{T}w - \kappa\frac{\partial \mathcal{T}}{\partial z}, \quad Q = 1.$$

Residence-time Integration

The clear disadvantage of using residence-time to extract regions of recirculation is that a Partial Differential Equation (PDE) needs to be solved. Equation (9) has a form similar to the conservative formulation of the Euler equations. This similarity enables the use of integration schemes developed for either the Euler or Navier-Stokes equations. It should also be noted that the coupling is looser than turbulence models. $\mathcal{T}$ is a function of $\vec{u}$ and optionally ρ and μ but does not feedback to the Euler or Navier-Stokes equations.

Ideally the solver writer would include another entry ($\mathcal{T}$) to the state vector. The time-step requirement for residence-time is less restrictive than either the Euler or Navier-Stokes equations because of the lack of acoustic waves so time-marching would not be effected.

In the case where modifying the solver can not be considered (and in pure post-processing applications) a residence-time solver is required. The scheme discussed in the rest of this paper is explicit, operates on a cell-by-cell manner, and can therefore take advantage of **pV3**'s parallel capability. Coupling to any implicit solver would require more care do to the time-step restrictions.

In producing a tool for general use it is important to consider the design goals. In the case of selecting an integration scheme for solving the residence-time PDE the following must be considered:

- Spatial Accuracy. The spatial accuracy must be at least as good as the solver.
- Temporal Accuracy. The temporal accuracy should be consistent with the solver and be at least second order accurate so it can be used in unsteady simulations.

- Numerical Dissipation. The dissipation produced by the scheme must be a minimum so that the sharp gradients in residence-time generated at the separation surfaces are not smeared.
- Node Based. The function of iso-surfacing is done where the value is at nodes that support the cells of the mesh. The scheme selected must be able to accurately place $\mathcal{T}$ at the nodes and with minimal averaging.
- Support General Discretizations. **pV3** supports structured blocks, disjoint tetrahedra, pyramids, prisms and hexahedra as either a homogenous or heterogeneous collection. The scheme selected should be able to deal with general meshing.

Lax-Wendroff

The explicit time-marching algorithm of the Lax-Wendroff type is a good choice to solve Equation (9). The Lax-Wendroff scheme is both second order in space and time. Little numerical dissipation is generated by the method and only small amounts of numerical smoothing are required under favorable conditions. The result is node based and the scheme can be cast for the general discretizations supported by **pV3**.

The integrator developed is similar to that used by Saxer[10] to solve the Euler equations for turbomachinery stator/rotor flow. The basic integration scheme was introduced by Ni,[11] recast by Hall,[12] and then extended by Ni and Bogoian[13] to 3-D. Saxer[10] then adapted the formulation to handle hexahedral unstructured grids. For this tool the complete suite of cell types are supported. Structured blocks are treated as a collection of disjoint hexahedra.

In solving the residence-time equation, it is assumed that the flow variables $\vec{u}$, and optionally ρ, and μ are known at all the nodes. The algorithm then computes the flux across each cell face by averaging the fluxes F, G, and H at the appropriate nodes. The flux residual is computed by adding the fluxes through the faces of the element, and then adding the source term for the cell. This residual is then distributed back to the nodes according to the Lax-Wendroff algorithm to evaluate the change to the residence-time.

In a Lax-Wendroff scheme there are 3 components; the first and second order terms as well as the source term. The major part of the first order term is cell based. For cell A:

$$\Delta U_A = -\frac{\Delta t}{V_A} \sum_{faces}^{CellA} (\bar{F}S_x + \bar{G}S_y + \bar{H}S_z) + \Delta t Q.$$

The bar over F, G, and H denotes an average over the nodes associated with the cell face. S_x, S_y, and S_z are the projected areas on the yz, xz, and xy planes for the specific face. V_A is the volume of the cell and Δt is the time-step.

The second order term is node based and requires the dual of the mesh. The contribution is based upon what proportion of the intersection of the cell of interest and the pseudo-mesh cell centered on node i. Specifically the second order and source terms are:

$$\Delta U_{iA} = -\sum_{faces}^{dual} \left(\Delta F_A \tilde{S}_x + \Delta G_A \tilde{S}_y + \Delta H_A \tilde{S}_z\right) + \frac{V_A}{2}\Delta Q_A.$$

Where $\tilde{S}_x$, $\tilde{S}_y$ and $\tilde{S}_z$ are the projected areas on the yz, xz, and xy planes for the faces of the pseudo-mesh cell contained in cell A. For inviscid incompressible flow:

$$\begin{aligned}
\Delta F &= \bar{u}\Delta\mathcal{T} + \bar{\mathcal{T}}\Delta u \\
\Delta G &= \bar{v}\Delta\mathcal{T} + \bar{\mathcal{T}}\Delta v \\
\Delta H &= \bar{w}\Delta\mathcal{T} + \bar{\mathcal{T}}\Delta w \\
\Delta Q &= 0
\end{aligned}$$

for inviscid compressible flow

$$\begin{aligned}
\Delta F &= \bar{u}\Delta(\rho\mathcal{T}) + \overline{(\rho\mathcal{T})}\Delta u \\
\Delta G &= \bar{v}\Delta(\rho\mathcal{T}) + \overline{(\rho\mathcal{T})}\Delta v \\
\Delta H &= \bar{w}\Delta(\rho\mathcal{T}) + \overline{(\rho\mathcal{T})}\Delta w \\
\Delta Q &= \Delta\rho
\end{aligned}$$

for inviscid compressible flow

$$\begin{aligned}
\Delta F &= \bar{u}\Delta(\rho\mathcal{T}) + \overline{(\rho\mathcal{T})}\Delta u - \bar{\mu}\Delta\frac{d\mathcal{T}}{dx} - \overline{\frac{d\mathcal{T}}{dx}}\Delta\mu \\
\Delta G &= \bar{v}\Delta(\rho\mathcal{T}) + \overline{(\rho\mathcal{T})}\Delta v - \bar{\mu}\Delta\frac{d\mathcal{T}}{dy} - \overline{\frac{d\mathcal{T}}{dy}}\Delta\mu \\
\Delta H &= \bar{w}\Delta(\rho\mathcal{T}) + \overline{(\rho\mathcal{T})}\Delta w - \bar{\mu}\Delta\frac{d\mathcal{T}}{dz} - \overline{\frac{d\mathcal{T}}{dz}}\Delta\mu \\
\Delta Q &= \Delta\rho
\end{aligned}$$

and, for flow with constant density and viscosity

$$\begin{aligned}
\Delta F &= \bar{u}\Delta(\rho\mathcal{T}) + \overline{(\rho\mathcal{T})}\Delta u - \kappa\Delta\frac{d\mathcal{T}}{dx} \\
\Delta G &= \bar{v}\Delta(\rho\mathcal{T}) + \overline{(\rho\mathcal{T})}\Delta v - \kappa\Delta\frac{d\mathcal{T}}{dy} \\
\Delta H &= \bar{w}\Delta(\rho\mathcal{T}) + \overline{(\rho\mathcal{T})}\Delta w - \kappa\Delta\frac{d\mathcal{T}}{dz} \\
\Delta Q &= 0.
\end{aligned}$$

Except for quantities defined above, the subscript A denotes quantities evaluated at cell A, while the subscript i stands for average quantities at node i. Quantities marked with the overbar are averages over the cell. Other quantities designated with the Δ operator are the change in that quantity with respect to time.

Finally, the complete contribution for node i in cell A is computed by:

$$\delta U_{iA} = \frac{1}{8}\frac{\Delta t}{V_i}\left\{\frac{V_A}{\Delta t}\Delta U_A + \Delta U_{iA}\right\}. \tag{10}$$

The contributions to node i (from all cells that touch the node) are computed. The sum of these contributions define the change at node i:

$$\delta U_i = \sum_{J=1}^{cells} \delta U_{iJ}.$$

Boundary conditions

Three types of boundary conditions must be considered: inlet, outlet, and wall. At the inlet, or the in-flow boundary, new fluid enters the computational domain. By definition, the residence-time of this fluid is zero. Therefore, the condition at the inflow boundary for any node i at the inflow is:

$$U_i = 0.$$

At outlet boundaries, the simplest boundary condition has been used, which is to do nothing – extrapolate. This assumption is reasonable as long as the gradients of the flow quantities in the direction normal to the boundary are small. However, if the user determines a more elaborate boundary treatment would be more appropriate, a call-back can be supplied which overrides the standard method. The values of U and δU_i of all the nodes will be passed to this subroutine, and the necessary modifications/adjustments (be it for the outlet boundary nodes or any other regions of the flow) can made.

At a wall boundary with non-zero velocity, the scheme must ensure that there is no flux through the wall. A flux through the wall would either create or destroy residence-time producing a non-physical result.

The wall condition is accomplished by making a correction to the contribution of nodes that touch the wall. The correction is performed after the Lax-Wendroff changes δU have been applied. The contribution of cell A (where A touches the wall) to δU_{iA} is corrected as follows:

$$\delta U_{iA} + \frac{1}{8}\frac{\Delta t}{V_i}\left(\bar{F}S_x + \bar{G}S_y + \bar{H}S_z\right)_{face=wall}.$$

Numerical smoothing

As stated above, Lax-Wendroff was selected, in that, under some conditions little numerical dissipation is required. A fourth-difference smoothing operator is required to damp out saw-tooth oscillations emitted by the scheme in the presence of strong gradients in $\vec{u}$ (such as at shocks). But, a fourth-difference smoother alone is insufficient. In fact, it will worsen the stability of the solution. This has to do with the dispersive characteristics of Lax-Wendroff under large gradients. The integration scheme *kicks-up* oscillations on both sides of a discontinuity. This causes negative values of residence-time on one side of the discontinuity and values of $\mathcal{T}$ larger than the current simulation on the other.

To make matters worse, strong discontinuities are found at the bounds of any recirculation region. Inside the region, the values of $\mathcal{T}$ would continue to get larger and larger where outside the region the values remain constant (for steady-state flow).

Therefore, a mix of second-difference and fourth-difference smoothing is used. This mix is switched so that under high gradients only the second order smoother is applied and under quiescent conditions only the fourth-difference smoothing is used. The switch used is simply

$$\mathcal{S}_i = \frac{\sum_k^{edge-nodes}\left(U_k - U_i\right)}{\epsilon + \sum_k^{edge-nodes}\sigma_1\left|U_k - U_i\right| + \sigma_0\left|U_k\right|}$$

where $\mathcal{S}_i$ is the switch value for node i. This value must be bounded on the high side at 1.0. σ_0 and σ_1 are coefficients that are currently set to 0.05 and 1.0 respectively.

The fourth-difference smoothing operator, identical to the one used by Saxer,[10] has been constructed for general discretizations. The smoothing term is added to the right-hand-side of Equation (10) and has the form

$$-\sigma_4\nabla\cdot\left(\ell\nabla\left(\ell^2\nabla^2 U\right)\right)$$

where σ_4 is the fourth-difference smoothing coefficient and ℓ is a length comparable to the local grid size. In discrete form, the smoothing operator becomes

$$-\sigma_4\frac{1}{V_i}\sum_J^{cells} V_J\left(\bar{D}_J^2 - D_i^2\right)$$

where V_J is the volume of the cell. D_i^2 is a pseudo-Laplacian based on the all of the edge nodes surrounding node i. It is defined by Holmes and Connell[14] as

$$D_i^2 = \sum_k^{edge-nodes}\Psi_k\left(U_k - U_i\right)$$

and, $\bar{D}_J^2$ is the discrete representation of a cell-averaged pseudo-Laplacian

$$\bar{D}_J^2 = \frac{1}{\#nodes}\sum_k^{nodes} D_k^2$$

where Ψ_k is a grid-dependent weight which determines the degree of dependence on the neighboring nodes. For details on how these weights are obtained, refer to Saxer.[10] In general, σ_4 is a coefficient given a value of between 0.0001 and 0.005.

The smoothing operator finally becomes

$$\begin{aligned}\delta U_i|_{smo} = \quad & \sigma_2\mathcal{S}_i\tfrac{1}{V_i}\textstyle\sum_J^{cells} V_J\left(\bar{U}_J - U_i\right)\\ & -\sigma_4(1-\mathcal{S}_i)\tfrac{1}{V_i}\textstyle\sum_J^{cells} V_J\left(\bar{D}_J^2 - D_i^2\right)\end{aligned}$$

where σ_2 is the second-difference smoothing coefficient given a value of between 0.001 and 0.01.

The Programming Interface

The residence-time and the **pV3** programming interfaces are similar in form and in coupling with the solver. Both require a call to an initialization routine that defines the spatial problem and sets constants. Also, a call is required at the end of the iterative loop (when all of the flow variables have been updated – new values for $\vec{u}$, μ and ρ have been computed). These are the only calls added to the solver and they have the prefix naming convention "RT_". This minimizes the changes to the solver and allows for easy removal during debugging.

The solver data is communicated through call-backs. These user-supplied routines are invoked by the "RT_" calls and their responsibility is to have the appropriate data arrays filled. The call-back routines are differentiated in that they do not contain the '_' after the "RT".

Tool-kit Calls

- `RT_Init(type, sm2, sm4, kappa, nNode, nCell, nInNode, nWallCell)`
  This routine must be called before any other residence-time functions. It specifies the size and type of problem. When mixing with **pV3**, the call should be placed before the `pV_Init` call.
  - `int type` is 0 to 3 for inviscid incompressible, viscous compressible, constant viscosity and density and inviscid compressible, respectively.
  - `float sm2` – second-difference smoothing coefficient (σ_2).
  - `float sm4` – fourth-difference smoothing coefficient (σ_4).
  - `float kappa` – κ, required for `type = 2` only.
  - `int nNode` – number of nodes in the volume.
  - `int nCell` – number of cells in the volume.
  - `int nInNode` – number of in-flow nodes.
  - `int nWallCell` – number of wall faces.
- `RT_Update(time)`
  This routine must be placed before the call to `pV_Update` when mixing with **pV3**.
  - `float time` – the current simulation time.
- `RT_getResTime(tau)`
  This call retrieves the current value of $\mathcal{T}$ for all nodes. This can be placed in the scalar field **pV3** call-back, `pVScal`, to fill the residence-time scalar for visualization.
  - `float tau[]` – the current values of residence-time.

Tool-kit Call-backs

- `RTgetGrid(xyz, iCell, tCell)`
  This routine is always required. It defines the volume discretization.
  - `float xyz[nNode][3]` – the node coordinates.
  - `int iCell[nCell][8]` – the cell definitions.
  - `int tCell[nCell]` – the cell type:
    1 = tetrahedron (4 nodes indices in `iCell`)
    2 = pyramid (5 nodes indices in `iCell`)
    3 = prism (6 nodes indices in `iCell`)
    4 = hexahedron (8 nodes indices in `iCell`)
- `RTinNodes(inNode)`
  This routine is always required. It defines the inflow nodes.
  - `int inNode[nInNode]` – the node indices for the inlet.
- `RTwallCells(wallCell)`
  `RTwallCells` defines all wall facets and the associated 3D cell.
  - `int wallCell[nWallCell][2]` – the first entry defines the cell index and the second defines the face index.
- `RTlocalVel(vel)`
  This routine is always required. It defines $\vec{u}$ for the entire volume.
  - `float vel[nNode][3]` – Node based velocities.
- `RTgetRho(rho)`
  This routine is required for inviscid and vsicous compressible cases (`type = 1 and 3`). This returns ρ for the entire volume.
  - `float rho[nNode]` – Node based densities.
- `RTgetMu(mu)`
  This routine is required for the vsicous compressible case (`type = 1`) only. This returns μ for the entire volume.
  - `float mu[nNode]` – Node based viscosity.
- `RTmodify(dU, U)`
  This routine, if supplied, allows the modification of either δU_i and/or U_i directly. This is required for special boundary conditions, such as moving interfaces, periodics or internal boundaries due to domain decomposition.
  - `float dU[nNode]` – Node based δUs.
  - `float U[nNode]` – Node based Us.

Validation and Examples

Axi-symmetric flow

The residence-time algorithm is first tested on flow through a converging-diverging duct. This case is selected because of the unusual nature of the solver. This axi-symmetric (with swirl) system was developed by Darmofal[15] and computes the streamfunction as part of the state-vector. This allows the simple calculation of the zero streamfunction or separation surface. All residence time calculations are done in full 3D by spinning the 2D geometry and producing 16 azimuthal sections. The cells are all hexahedra except those at the centerline that are represented as prisms. The following examples were computed with a Reynolds number of 200 and a swirl ratio of 1.75. The duct walls are treated with a slip condition. The solver was run in a transient mode but the input conditions produce a stable flow. Residence-time is calculated for both inviscid incompressible and constant viscosity and density cases.

Figure 1 displays the duct and a rake of streamines generated from the upsteam region. It is clear that something interesting is occuring at the converging part of the pipe.

When seeding streamlines from within that region, it can be seen that there is a part of the flow field that traps the paths. The steamline module in **pV3** stops the integration after a fixed number of integration points to avoid endless compute. Two streamlines are seeded and depicted in Figure 2.

The separation bubble is fully displayed just downstream of the converging section as shown in Figure 3. This is an iso-surface of the streamfunction with value 0.

The characteristic time for this problem is about 24. This is found by looking at an iso-surface of residence-time and finding the time where the major portion of the flow exits the domain. A value greater than that needs to be selected to differentiate old fluid from the core flow. Figure 4 displays an iso-surface of residence-time at the value of 42 when running using the inviscid incompressible formulation.

The tail seen behind the bubble is due to the fact that the flow behind this object is slow. An axi-symmetric comparsion of contours can be seen in Figure5.

Figure 6 shows the residence-time iso-surface for the constant viscosity and density case. It should be noted that the tail trailing the bubble is larger indicating the the greater mixing due to the viscous mixing.

Conclusions

The residence-time equations for different types of flow conditions (inviscid incompressible, viscous incompressible, inviscid compressible, and constant density/viscosity) have been formulated. An explicit time-marching algorithm of Lax-Wendroff type is used to solve the equations either coupled to a solver or in a post-processing mode. This explicit algorithm performs the computation on a cell-by-cell/node-by-node manner, and thus can be used within the context of **pV3**'s distributed processing. In order to handle the variety of possible boundary-condition treatments, provisions are made to allow the programmer to supply a subroutine where some integration variables can be modified before the residence-time values are updated.

This procedure is inefficient for steady-state post-processing and can not be used for co-processing if the solver formulation is implicit. An accelerated solving procedure is required that can assist in these setting. This procedure must be immune to the fact that there will likely be flow reversal with in the domain of interest.

Acknowledgments

David Sujudi performed much of the initial work on this technique when he was a student at MIT. Mike Giles of Oxford suggested its path. Dave Darmofal was helpful in fixing the numerical smoothing and keeping me honest.

This work was partially sponsored by NAS at NASA Ames Research Center with David Kao as the Technical Monitor and by Army Research Labs with Stephen Davis as the Technical Monitor. Additional support came from the IBM SUR Project and the IBM UUP Project.

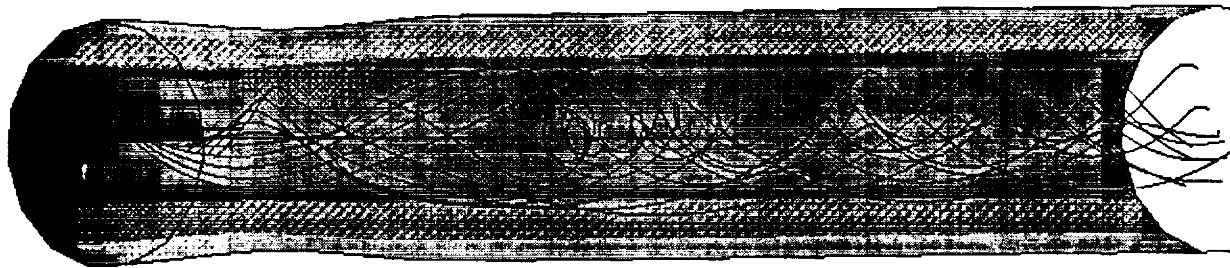


Fig. 1 The geometry with a rake of 10 streamlines seeded from an up-stream position

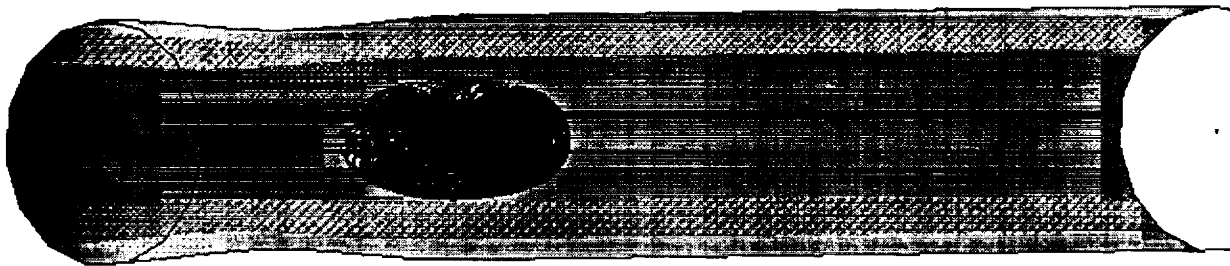


Fig. 2 2 streamlines seeded from within the recirculation region

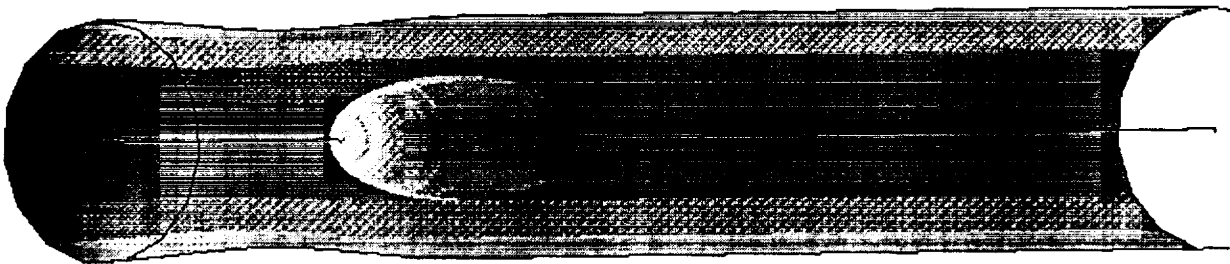


Fig. 3 Streamfunction zero from the solver

References

[1]J. Helman and L. Hesselink. Analysis and Representation of Complex Structures in Separated Flows. SPIE Proceedings, Vol 1459, 1991.

[2]J. Helman and L. Hesselink. Visualizing Vector Field Topology in Fluid Flows. IEEE Computer Graphics and Applications, May 1991.

[3]David Sujudi. Distributed Visualization and Feature Identification for 3D Steady and Transient Flow Fields. MIT Thesis, 1996.

[4]D. Sujudi and R. Haimes. Identification of Swirling Flow in 3-D Vector Fields. AIAA Paper 95-1715, 1995.

[5]D. Kenwright. Automatic Detection of Open and Closed Seperation and Attachment Lines. Proceedings of IEEE Visualization '98, 1998.

[6]D. Kenwright and R. Haimes Vortex Indetification – Applications in Aerodynamics. Proceedings of IEEE Visualization '97, 1997.

[7]R. Haimes. **pV3**: A Distributed System for Large-Scale Unsteady Visualization. AIAA Paper 94-0321, 1994.

[8]R. Haimes and D. Edwards. Visualization in a Parallel Processing Environment. AIAA Paper 97-0348, 1997.

[9]M. B. Giles, personal communication.

[10]A. Saxer. A Numerical Analysis of 3-D Inviscid Stator/Rotor Interactions Using Non-Reflecting Boundary Conditions. MIT Thesis, 1992.

[11]R.-H. Ni. A Multiple Grid Scheme for Solving the Euler Equations. AIAA Journal, Vol 20, pp. 1565-1571, 1981.

[12]M. G. Hall. Cell-Vertex Multigrid Schemes for Solution of the Euler Equations. Technical Report 2029, Royal Aircraft Establishment, 1985.

[13]R.-H. Ni and J. Bogoian. Prediction of 3-D Multi-Stage Turbine Flow Field Using a Multiple-Grid Euler Solver. AIAA Paper 89-0203, 1989.

[14]D. G. Holmes and S. Connell. Solution of the 2-D Navier-Stokes Equations on Unstructured Adaptive Grids. AIAA Paper 89-1932, 1989.

[15]D. Darmofal. A Study of the Mechanisms of Axisymmetric Vortex Breakdown. MIT Thesis, 1993.

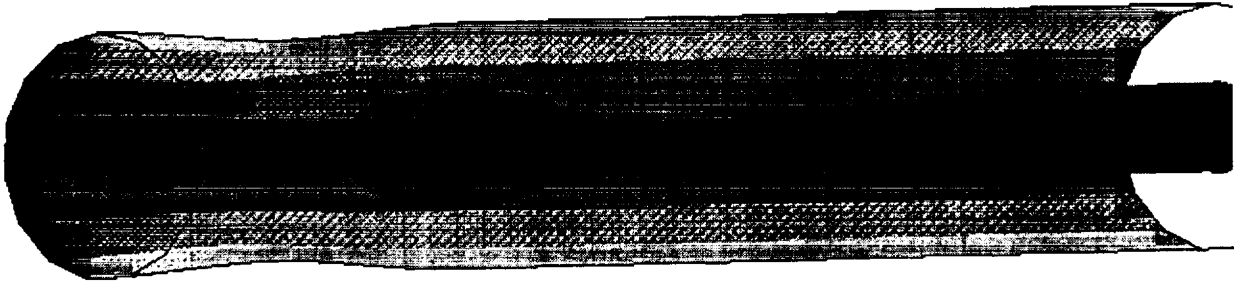


Fig. 4 Iso-surface of residence-time using the invicid formulation

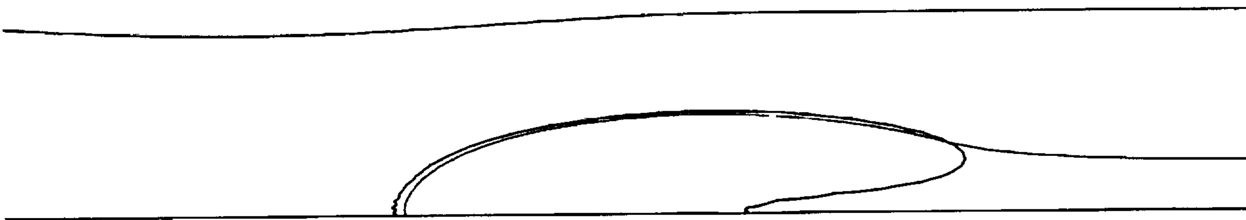


Fig. 5 A comparison between streamfunction 0 and residence-time contours in an axi-symmetric cut

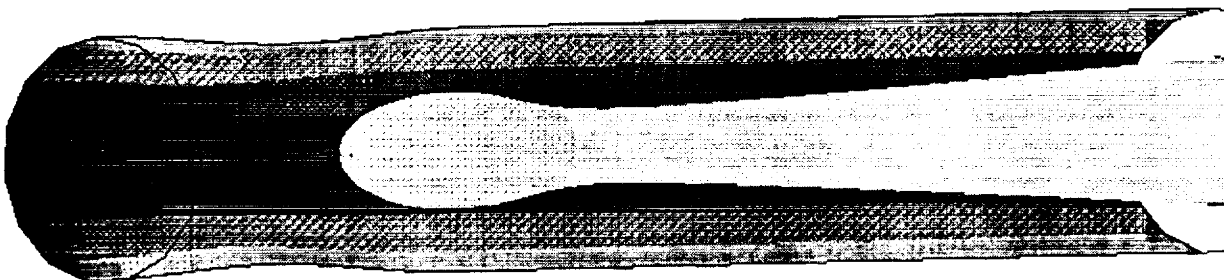


Fig. 6 Iso-surface of residence-time for the constant viscosity and density case

On the Velocity Gradient Tensor and Fluid Feature Extraction

Robert Haimes*
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology
haimes@orville.mit.edu
and
David Kenwright†
MRJ Technology Solutions
NASA Ames Research Center
davidk@nas.nasa.gov

In the analysis of the velocity gradient tensor the local flow characteristics can be classified. By focusing on critical points one can build a global view of the flow topology. For this 3x3 tensor an eigen-analysis produces 3 eigenvalues. Mapping these to the complex plane produces the classification signature. Vector field topology can be used as the foundation of automated fluid feature extraction.

This paper builds the foundation for using the velocity gradient tensor and discusses how it has been (or could be) successfully used in finding fluid flow features.

Introduction

Fluid flow features such as vortices, separation, and shocks are items of interest that can be found in the results obtained from Computational Fluid Dynamics (CFD) simulations. Most commercial visualization systems provide users with a suite of general-purpose tools (e.g., streamlines, iso-surfaces, and cutting planes) with which to analyze their data sets. In order to find important flow features, users must interactively search their data using one or more of these exploratory tools. Scientists and engineers that use these tools on a regular basis have reported the following drawbacks:

- Exploration Time.
  Interactive exploration of large-scale CFD data sets is laborious and consumes hours or days of the scientists/engineers time.

- Field Coverage.
  Interactive techniques produce visualizations based on a limited number of sample points in the grid or solution fields. Important features may be missed if the user does not exhaustively search the data set.

- Non-specific.
  Interactive techniques usually reveal the flow behavior in the neighborhood of a flow feature rather than displaying the feature itself.

- Visual Clutter.
  After generating only a small number of visualization objects (e.g., streamlines, cutting planes, or iso-surfaces) the display becomes cluttered and makes visual interpretation difficult.

Flow visualization research is now concentrating on feature extraction algorithms that automate the data analysis and extract the salient features with little or no human intervention. One of stumbling blocks has been the lack of precise mathematical definitions for flow structures such as vortex cores, separation lines, recirculation bubbles, and shock surfaces.

Using local representations of the velocity vector and velocity gradient fields, feature extraction techniques have been developed that locate many of the aforementioned features. These techniques, based on concepts from critical point theory, utilize the property that linear vector fields have a finite number of flow topologies. Feature extraction techniques have the following advantages over exploratory visualization tools:

- Fully Automated.
  The analysis can be done off-line in a batch computation.

- Local Analysis.
  The computations for each cell are independent of any other cell and may be performed in parallel.

- Deterministic Algorithms.
  There are no "parameters" that the users can adjust.

*Principal Research Engineer
†Senior Research Scientist

Copyright © 1999 by Robert Haimes & David Kenwright. Published by the American Institute of Aeronautics and Astronautics, Inc. with permission.

- Data Reduction.
  The output geometry is several orders of magnitude smaller than the input data set.

- Quantitative Information.
  Precise locations for all the flow features can be extracted.

This paper discusses how the velocity gradient tensor has been used in the visualization of CFD results. In particular, it describes the relationship between the tensor and the topology of the flow field and describes several automated feature extraction techniques based on this relationship.

Calculating the Velocity Gradient

The analyses presented in this paper are three-dimensional. In this context velocity is a vector with 3 components in Cartesian space

$$\vec{u} = \begin{bmatrix} \frac{dx}{dt} \\ \frac{dy}{dt} \\ \frac{dz}{dt} \end{bmatrix} = \begin{bmatrix} u \\ v \\ w \end{bmatrix}$$

Velocity can be obtained for every node in the discretized volume from the results of CFD simulations. This vector field is commonly used in the visualization phase of the analysis to generate streamlines or streaklines in an attempt to assist the investigator in understanding the topology of this vector field. The gradient of the velocity field, $\underline{V}$, is rarely used for visualization purposes, although it is central part of vector field topology.[4]

The velocity gradient contains the information on how the velocity is changing in space. This 3×3 tensor is defined as

$$\underline{V} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & \frac{\partial u}{\partial z} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial z} \\ \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} & \frac{\partial w}{\partial z} \end{bmatrix} \tag{1}$$

$\underline{V}$ is not usually a quantity that is directly output by a CFD solver but can easily be derived from the vector field and the supporting mesh. For example, a first-order approximation to $\vec{u}$ gives rise to the following linear interpolation function:

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} + \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} & \frac{\partial u}{\partial z} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} & \frac{\partial v}{\partial z} \\ \frac{\partial w}{\partial x} & \frac{\partial w}{\partial y} & \frac{\partial w}{\partial z} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2}$$

A tetrahedron has exactly the correct amount of information to compute $\underline{V}$ and the coefficient vector $(c_1, c_2, c_3)^T$. This means that a unique (and constant over the entire element) $\underline{V}$ can be constructed. Elements with more nodes (e.g. pyramids, prisms, hexahedra, and etc.) have a changing $\underline{V}$ based on the cell's interpolant. For all grid types, elements can be broken up into tetrahedra, $\underline{V}$ calculated and then distributed back to the nodes in a finite element manner.

It should be noted that $\underline{V}$ is not dimensionless. It has the units of $time^{-1}$. Therefore most all data derived directly from the velocity gradient tensor has the form of a rate.

Velocity Gradient Decomposition

A common technique used to better understand the local flow is to decompose $\underline{V}$ to its symmetric and anti-symmetric parts:

$$\underline{S} = \frac{1}{2}\left(\underline{V} + \underline{V}^T\right) \tag{3}$$

$$\underline{\Omega} = \frac{1}{2}\left(\underline{V} - \underline{V}^T\right) = \frac{1}{2}\begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \tag{4}$$

Where $\underline{S}$ is a measure of the strain which contains both bulk and shear components. $\underline{\Omega}$ contains the rotational part of the flow. As can be seen in Equation (4) there are only 3 components of this tensor. Vorticity is more commonly viewed as a vector:

$$\vec{\Omega} = \begin{bmatrix} \omega_x \\ \omega_y \\ \omega_z \end{bmatrix} = \begin{bmatrix} \frac{\partial w}{\partial y} - \frac{\partial v}{\partial z} \\ \frac{\partial u}{\partial z} - \frac{\partial w}{\partial x} \\ \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \end{bmatrix} \tag{5}$$

The trace of $\underline{S}$, which equals the trace of $\underline{V}$, reflects the divergence of the field:

$$\nabla \cdot \vec{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \tag{6}$$

This decomposition has been used by a number CFD visualization researchers, notably:

- Darmofal and Haimes[1]. During the integration of streamlines both $\nabla \cdot \vec{u}$ and $\vec{\Omega}$ are tracked. The streamlines optionally are plotted as ribbons, where one edge is the actual streamline and the other rotates to reflect the curl. The streamline could be plotted as a tube centered on the streamline where the thickness of the tube relates to the local divergence. This is similar to the *Stream Polygon*.[2] Also, an option would draw a spiral pattern on the tube to display the direction (the tube itself), the divergence (the tube cross-section size) and the vorticity (the spiral pattern).

- deLeeuw and vanWijk[3]. Here an interactive probe is constructed that points in the local direction of the flow and has the ability to concurrently display curvature, $\nabla \cdot \vec{u}$, $\vec{\Omega}$, acceleration and shear in a physically meaningful manner.

These techniques are interactive and attempt to locally display components of $\underline{V}$.

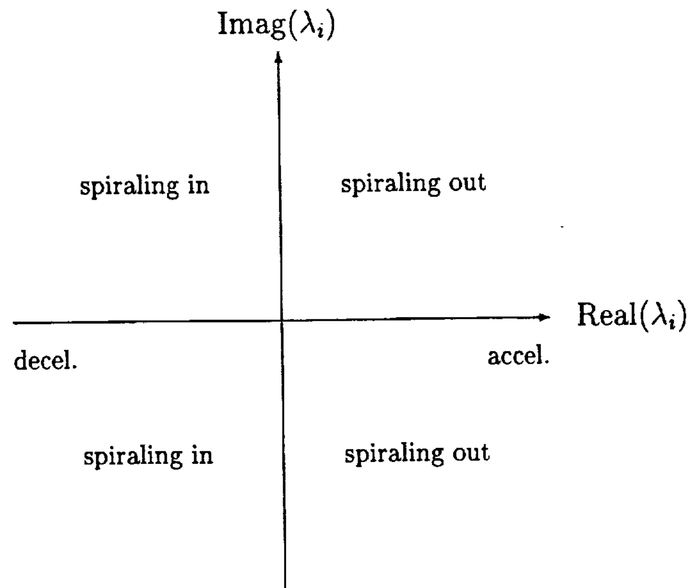


Fig. 1 Flow classifications in eigenvalue plane, $\lambda_{1,2,3} = \text{eig}\,(\underline{\nabla})$.

Critical Point Theory

Critical points of a vector field are those points where the magnitude of the velocity vanishes. Stated in another way, these points are defined where the streamline slope is indeterminate and the velocity is zero relative to an observer.[4] From this definition it is clear that the process of locating critical points is not Galilean invariant.

From a topological point of view critical points in the field mark changes. An eigen-analysis is used on the velocity gradient tensor at critical points to classify the local topology. For $\underline{\nabla}$, the eigenvalues ($\lambda_{1,2,3}$), are the fundamental quantities which determine the qualitative features of the flow pattern. In general, various possible flow patterns and the corresponding eigenvalue-based flow classifications are summarized in Figure 1. It should be noted that when 3 eigenvalues exist, there are two general types: 3 real eigenvalues or 1 real (λ_r) and a complex conjugate pair (λ_c).

Any real eigenvalue sits on the real axis of Figure 1. The sign of the eigenvalue indicates whether the flow is accelerating (positive) or decelerating (negative). The magnitude of the value reflects the strength. The complete suite of critical point classification for 3 real eigenvalues from $eig(\underline{\nabla})$ can be seen in Figure 2. In this Figure, each type is displayed by a *cartoon* of the flow pattern approaching the critical point and a plot of the λs in the complex eigenvalue plane.

In Figure 2 the portrait is displayed as flow on a 2D plane and then arrows indicating a flow direction orthogonal to that surface. The choice is somewhat arbitrary. The plane is defined by the 2 of eigenvectors (the ones plotted on the plane).

When there is a complex conjugate pair of eigenvalues, λ_r acts as described above. The magnitude of the imaginary part of λ_c indicates the strength of the spiraling flow. If the value is small (near the real axis) the flow (in the plane) is hardly swirling. If the magnitude is great then the flow is rotating rapidly abound the point. The sign of the real part of λ_c indicates whether the flow is converging (negative) or diverging (positive) from the point, with the magnitude of the value again reflecting the strength of the attraction or repulsion. The special case where the real part of λ_c is zero just produces concentric periodic paths.

The complete suite of critical point classification for λ_r and λ_c can be seen in Figure 3. In this Figure, each type is again displayed by a *cartoon* of the flow pattern approaching the critical point on the complex eigenvalue plane. The portrait is displayed as flow on a 2D plane and then arrows indicating a flow direction orthogonal to that plane (this direction is defined by the eigenvector associated with λ_r). In these cases, the plane always depicts the spiraling flow. The plane is defined by the 2 eigenvectors associated with λ_c.

Critical Points and general eigen-analysis of $\underline{\nabla}$ has been used by a number CFD visualization researchers, these include:

- Globus, Levit and Lasinski[5]. This paper introduced the vector field topology module to FAST.[6] In this work, critical points were located and marked with *glyphs*. The shape of these icons reflected the critical point classification. One could use this module to interactively mark vortex cores. This was drawn by selecting *glyphs* that indicate spiraling flow and integrating a streamline in the direction of the eigenvector associated with λ_r. This was an important step in the right direction for feature extraction. Unfortunately, this module of FAST was not frequently used because:
  - A knowledge of Critical Point Theory. Without the knowledge and understanding of the concepts, looking at these icons told the investigator little.

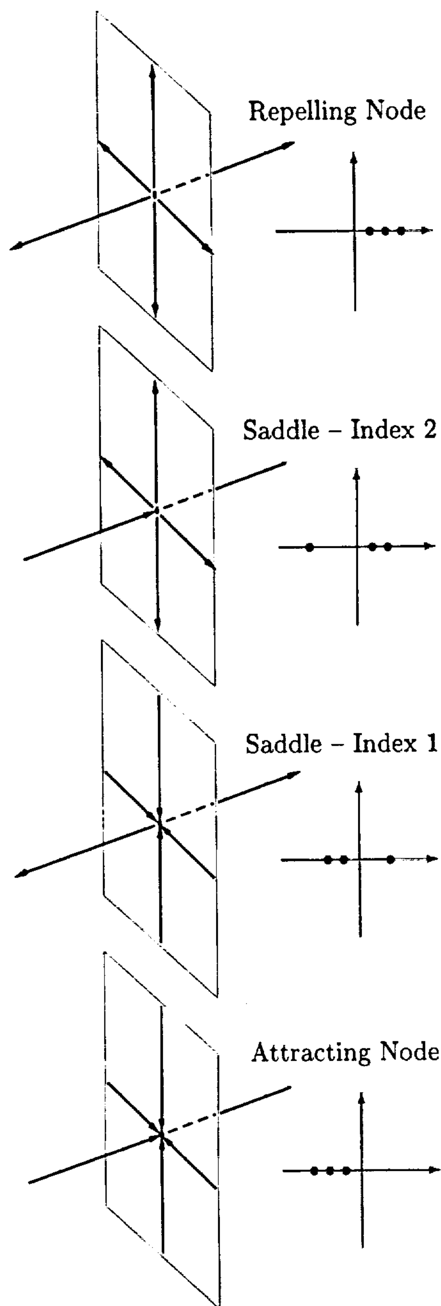


Fig. 2 Critical point portrait and eigenvalue plane for non-spiral flow

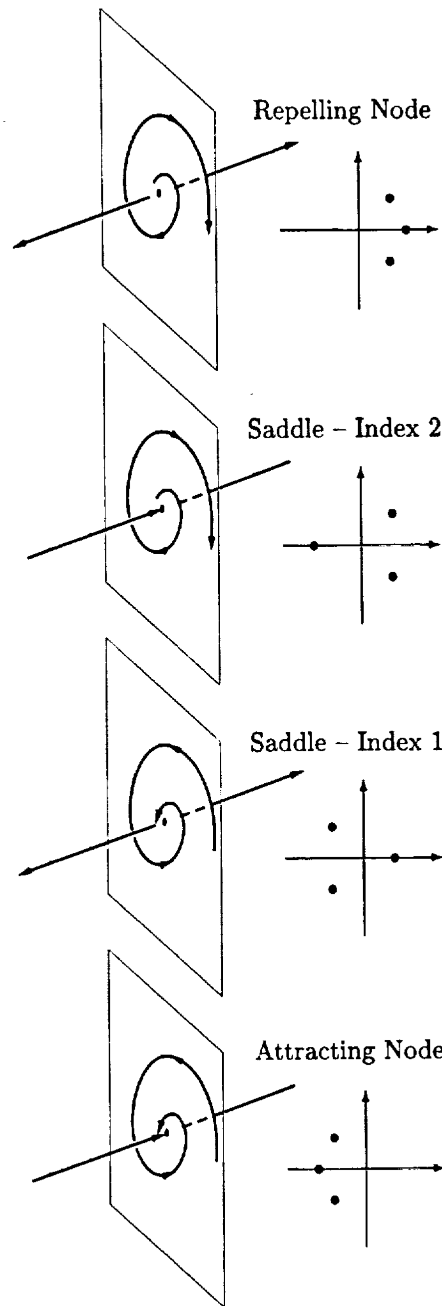


Fig. 3 Critical point portrait and eigenvalue plane for spiral flow

- Results were difficult to interpret. Even with the knowledge, one needs a good spatial imagination to *connect the dots.*
- Complex Flows suffered from clutter. With an interesting (and therefore complex) flow topology the number of Critical Points may approach 100. It is not clear what one can do with this many (sometimes overlapping) *glyphs.*

- Delmarcelle and Hesselink[7]. The concept of a *Hyperstreamline* is introduced. This concept uses tensor field lines as the direction of the *Hyperstreamline* and the cross-section is defined from the eigen-analysis. This technique also suffers from difficult to interpret graphics.

Fluid Feature Extraction

Visualization is the final phase of the suite for traditional CFD analysis. In this step, the investigator attempts to understand the results by visually probing the data. These tools (cuts, iso-surfaces, streamlines and others) only hint at the real answers. Usually, much interaction is required to get a full understanding of the flow field, in particular where the flow topology is complex.

Feature extraction is the next important step in visualization. The results can be used interactively to point directly to areas of interest. As analysis suites

become more integrated, feature extraction can be used directly to enhance the fidelity of the solution by grid adaptation in the appropriate regions. During parametric studies, the classification of features found in the simulation can be quickly scanned to find transitions in topology.

The discussion below will review the important fluid flow features and how the velocity gradient tensor either contains the ability to extract the feature or is otherwise affected.

Shocks

Normal shocks express themselves as abrupt changes in the magnitude of the velocity field. This can be seen in the eigenvalues of $\underline{V}$ as at least one strong negative real λ. This is not sufficient for constructing a shock finder because one can not easily differentiate a strong compression wave from a shock. See Lovely and Haimes[8] for a complete description on shock extraction.

Vortex Cores

Vortices can be automatically detected by using $\underline{V}$ throughout the mesh looking for situations of swirling flow. All mesh elements are broken into tetrahedra (if not already this type of element). The unique $\underline{V}$ is constructed and then classified. If swirling, the direction orthogonal to the spiral plane (the eigenvector associated with λ_r) is used as the axis of swirl. This direction is subtracted from the nodal velocities. These *reduced velocities* are used to see if any faces display a zero. If so, that location on the face is marked. With two (or more) marks on the tetrahedron's face, it is determined that the core center-line has pierced the cell. These lines are collected and drawn to display the core segments.

This particular algorithm is fully described in Sujudi and Haimes[9] and contains no constants and requires no user intervention. The core finder has been applied to a fairly complete suite of applications, see Kenwright and Haimes.[10]

The strength of the vortex is best described by $|\vec{\Omega}|$.

This technique, although satisfying, is not without problems. These are:

- Not producing contiguous lines.
  The method, by its nature, does not produce a contiguous line for the vortex core. This is due to two reasons; (1) for element types that are not tetrahedra the interpolant that describes point location within the cell is not linear. This means that if the core passes through these elements the line can display curvature. By subdividing pyramids, prisms, hexahedra and higher-order elements into tetrahedra for this operation produces a piecewise linear approximation of that curve. And (2) there is no guarantee that the line segments will meet up at shared faces between tetrahedra. This is because the eigenvector associated with the real eigenvalue will not be exactly the same in both neighbors, so when this vector is subtracted from the vector values at the shared nodes each tetrahedra sees a differing velocity field for the face.

- Locating flow features that are not vortices.
  This method finds patterns of swirling flow (of which a vortex core is the prime example). There are other situations where swirling flow is detected, specifically in the formation of boundary layers. Most implementations of this technique do no process cells that touch solid boundaries to avoid producing line segments in these regions. But this does not always solve the problem. In some cases (where the boundary layer is large in comparison to the mesh spacing) this boundary layer generation is still found.

- Sensitive to other non-local vector features.
  Critical point theory gives one classification for the flow based on the local flow quantities. 3D points can display a limited number of flow topologies including swirling flow, expansion and compression (with either acceleration or deceleration). The flow outside this local view may be more complex and have aspects of all of these components. The local classification will depend on the strongest type. Also if there are two (strong) axes of swirl, the scheme will indicate a rotation that is a combination of these rotation vectors based on the relative strength of each. This has been reported by Roth and Peikert[11] where the overall vortex core strength was not much greater that the global curvature of the flow. The result was that the reported core location was displaced from the actual vortex.

The first point can be addressed by re-casting the algorithm to be face-based instead of cell-based. Enforcing the cell piercing to match at cell faces insures that the line segments generated will produce a contiguous core. This can be done via the following modification to the algorithm:

1. Compute the $\underline{V}$ at each node.
   This requires much more storage – 9 words are needed for each node in the flow field. This has the advantage that the stencil used for the operation is larger than the cell and therefore the result will be generally smoother.

2. Average the node tensors (on the face) to produce a face-based $\underline{V}$.
   This insures that the same tensor is produced for the two cells touching the face.

3. Perform the eigen-analysis on the face tensor.
   If the system signifies swirling flow, determine if

the swirling axis cuts through the face by looking at the *reduced velocity*. If, so mark the location on the face.

This is not a good result in terms of CPU cycles, in particular for structured blocks, where each individual hexahedron is broken up into 6 tetrahedra (5, the minimum does not promote face matching). This means that for each element in the mesh a minimum of 12 eigen-analyses are required.

This performance problem suggest another, related, technique:

1. Compute the velocity gradient tensor at each node.
2. Perform the eigen-analysis on the node tensor. The tensor can be overwritten with the critical point classification and the swirl axis vector for rotating flow.
3. Average the swirl axis vectors for the nodes that support the tetrahedral face. This should only be done if all nodes on the face indicate swirling flow. Some care needs to be taken to insure that the sense of the vectors are the same. Determine if the swirling axis cuts through the face, and if so, mark the location on the face.

For tetrahedral meshes, the reduction of compute load is by a factor of 5 to 6 over the original method (there are roughly 5.5 tetrahedra per node in 'good' unstructured grids). For structured blocks, where the number of nodes is about equal to the number of hexahedra, the eigen-analyses count is on the order of one per cell.

Separation and Attachment

Helman and Hesselink[12,13] have developed a visualization scheme for generating separation surfaces using only the vector field. The scheme starts by finding the critical points on the surface of the object. Streamlines are integrated along the principal directions of certain classes of critical points and then linked to the critical points to produce a 2-D skeleton of the flow topology near the object. Streamlines are integrated out to the external flow starting from points along certain curves in the skeleton. These streamlines are then tessellated to generate the separation surfaces.

Kenwright[14] developed a local technique for extracting separation and attachment lines based eigen-analysis of the velocity gradient tensor. This local scheme detects two types of separation; open and closed. Closed separation lines originate and terminate at Critical Points, i.e., the type found by Helman and Hesselink's algorithm. Open separation lines do not need to start or end at critical points and cannot be detected by vector field topology methods. Kenwright's technique applies critical point theory to the surface flow on triangular elements near a body. The three velocity vectors at the vertices of each triangle are used to construct a 2 dimensional linear vector field with a constant velocity gradient tensor. Tensors with two real eigenvalues produce flow patterns that contain separation and attachment lines. For linear vector fields, these lines are straight, parallel to the eigenvectors, and pass through the Critical Point. If one of these lines happens to cross its associated triangle, the line segment bounded by the triangle is collected for rendering. By repeating this process in every triangle, the independent line segments combine to form larger separation or attachment lines.

Boundary Layers & Wakes

There has been little success in the ability to automatically locate boundary layers from the output of CFD solvers. In fact, many algebraic and one equation turbulence models need to know the edge of the boundary layer. In most all cases vorticity is used as the marker. This has been determined to be inadequate.

Boundary layers (and wakes) display two features; (1) the generation of vorticity and (2) the fluid is under shear stress. This suggests a marker that is a function of both $\vec{\Omega}$ and shear.

$\underline{S}$ from Equation (3) can be used to compute the rate of shear stress. This stress tensor contains both the bulk and shear stresses and is dependent on the coordinate system. To extract a single scalar that is coordinate system invariant and has the bulk terms removed it is necessary to diagonalize this tensor (again another eigen-analysis). The result is always 3 real eigenvalues ($\underline{S}$ is symmetric positive definite). These eigenvalues (λ_{s1}, λ_{s2}, λ_{s3}) produce a vector which signifies the 'principle axis of deformation'. By employing techniques from Solid Mechanics, the norm of the second principal invariant of the 'stress deviator' can be used as a measure of the shear. This is

$$\sqrt{\frac{(\lambda_{s1}-\lambda_{s2})^2+(\lambda_{s1}-\lambda_{s3})^2+(\lambda_{s2}-\lambda_{s3})^2}{6}}. \quad (7)$$

The boundary layer equations are essentially 2D. Using only the 2 strongest eigenvalues empirically gives better results. Therefore Equation (7) reduces to

$$\lambda_{s1}-\lambda_{s2}. \quad (8)$$

A scalar field can be constructed from a function of $|\vec{\Omega}|$ and the value from Equation (8). This node-based field can be passed through an iso-surface algorithm. This marks those areas that have a certain amount of both rotation and shear stress. This empirical approach has shown good results in detecting boundary layers and wakes but has the following drawbacks:

- A function of shear and rotation is *ad hoc.*
- The value is not non-dimensional, but has units of inverse time.
  This means that the iso-surface value used to define the edge of the layer changes from case to case. This scalar needs to be multiplied by some characteristic time associated with the problem.
- The value used for the iso-surface is not specified via theory.

Conclusions

The ability to automatically detect flow features from the results of CFD codes is closely tied to the velocity gradient tensor $\underline{V}$. Decomposition into the rotational and irrotational parts allows examination of some aspects of the local flow. Critical Point Theory can also be used to map the eigenvalues of $\underline{V}$ onto the complex plane thus providing a local classification of the flow. These aspects can be used to attempt to automatically detect fluid flow features by following the local topology of the vector field to gain a global view of the features.

Acknowledgments

This work was partially sponsored by NAS at NASA Ames Research Center with David Kao as the Technical Monitor and by Army Research Labs with Stephen Davis as the Technical Monitor. Additional support came from the IBM SUR Project and the IBM UUP Project.

References

[1] D. Darmofal and R. Haimes. Visualization of 3-D Vector Fields: Variations on a Stream. AIAA Paper 92-0074, 1992.

[2] W. Schroeder, C. Volpe and W. Lorensen. The Stream Polygon: a Technique for 3-D Vector Field Visualization. Proceedings of IEEE Visualization '91, 1991.

[3] W. deLeeuw and J. vanWijk. A Probe for Local Flow Field Visualization. Proceedings of IEEE Visualization '93, 1993.

[4] M. Chong, A. Perry and B. Cantwell. A General Classification of Three-Dimensional Flow Fields. Phys. Fluids A, vol. 2, pp. 765-777, 1990.

[5] A. Globus, C. Levit and T. Lasinski. A Tool for Visualizing the Topology of Three-Dimensional Vector Fields. Proceedings of IEEE Visualization '91, 1991.

[6] G. Bancroft, F. Merritt, T. Plessel, P. Kelaita, R. McCabe, A. Globus. FAST: A Multi-Processing Environment for Visualization of CFD. Proceedings of IEEE Visualization '90, 1990.

[7] T. Delmarcelle and L. Hesselink. Visualization of Second Order Tensor Fields and Matrix Data. Proceedings of IEEE Visualization '92, 1992.

[8] D. Lovely and R. Haimes. Shock Detection from Computational Fluid Dynamics Results. AIAA Paper 99-????, 1999.

[9] D. Sujudi and R. Haimes. Identification of Swirling Flow in 3-D Vector Fields. AIAA Paper 95-1715, 1995.

[10] D. Kenwright and R. Haimes. Vortex Indentification – Applications in Aerodynamics. Proceedings of IEEE Visualization '97, 1997.

[11] M. Roth and R. Peikert. Flow Visualization for Turbomachinery Design. Proceedings of IEEE Visualization '96, 1996.

[12] J. Helman and L. Hesselink. Analysis and Representation of Complex Structures in Separated Flows. SPIE Proceedings, Vol 1459, 1991.

[13] J. Helman and L. Hesselink. Visualizing Vector Field Topology in Fluid Flows. IEEE Computer Graphics and Applications, May 1991.

[14] D. Kenwright. Automatic Detection of Open and Closed Seperation and Attachment Lines. Proceedings of IEEE Visualization '98, 1998.

Shock Detection from Computational Fluid Dynamics Results

David Lovely*
Massachussetts Institute of Technology, Cambridge, MA 02139

Robert Haimes†
Massachussetts Institute of Technology, Cambridge, MA 02139

In complex flow regimes, it may be difficult for an analyist to find the location of shock discontinuities from within a Computational Fluid Dynamics (CFD) solution. They do not correspond to locations where the mach number is unity, and the high gradients associated with the discontinuity can be difficult to detect because of numerical smoothing performed in order to obtain the solution.

An algorithm is introduced that uses the physics of the CFD solution to locate shocks in transient and steady state solutions. The test was validated with simple 1 and 2 dimensional models, then extend to more realistic 3 dimensional flows. A set of filtering algorithms was developed to remove any false shock indications.

Results indicate that both the stationary and transient shock finding algorithms accurately locate shocks, but need filtering to compensate for lack of sharp discontinuities found in CFD solutions.

Nomenclature

P	Pressure
ρ	Density
a	Speed of sound
U	Speed
M	Mach number
$\vec{M} = \frac{\vec{q}M}{U}$	Mach vector
γ	ratio of specific heats
$\vec{q}$	Velocity

Introduction

Shock waves are compression waves in flow fields that may occur when the velocity of the fluid exceeds the local speed of sound. The state of the fluid as described by the pressure, velocity and other primitive variables can change radically across a shock boundary of only a few molecular mean free paths wide. These discontinuites are of interest to designers because their strength and location affects drag on aircraft, the functioning of inlets, the efficiency of nozzles and a host of other design problems in fluid mechanics. When these problems are simulated numerically with CFD codes, the discontinuites often persist, but become harder to detect because of the numerical properties of the solution. The problem is analogous to finding edges in an image, the purpose of both applications is to find discontinuities in a scalar field that contain large spacial changes in the scalar along with noise and smoothing. In the case of CFD solutions, the noise in the solution is analgous to dispersion and the smoothing is related to dissipation. Figure 1 shows the how the pressure across the idealized pressure discontinuity at a shock differs from an actual shock. Dissipation blurs the edges of the shock, making it hard to determine it's extent. Dispersion creates the high frequency noise on the edges of the shock.

There are two approaches to detecting shock discontinuites. The first approach is to view it as an edge detection problem and apply the methods that have been developed in that field. Alternatively the physics of shocks can be used to create a detection algorithm that is not applicable to the more general problem. This paper takes the physics approach since there is not a generally agreed on 'best' edge detection algorithm and by looking at the physics, it might be possible to formulate a more rigerous detector.

In the past, shock waves have been extracted with an edge detection technique which has been described by Ma, Rosendale, and Vermeer.[1] This technique searches for inflection points in the pressure or density fields by finding areas where the laplacian of the scalar quantity goes to zero. This works because scalar quantities like pressure have their maximum rate of increase at the shock and the second derivative of the scalar goes to zero, as shown in the figure 1. However, this detector will pick up other features that are not shocks. There can be expansion waves in a flow solution that are similar to shock features, but pressure and density decrease along streamlines through the expansion. The second derivatives of pressure and density are also zero in quiescent flow like regions far from a body. Both these features have to be removed from the marked areas. One advantage of

*Graduate Student
†Principal Research Engineer

this approach is that it also captures moving shocks in a transient solution, since pressure and density are invariant quantities, unchanged when the frame of reference is attached to a moving shock.

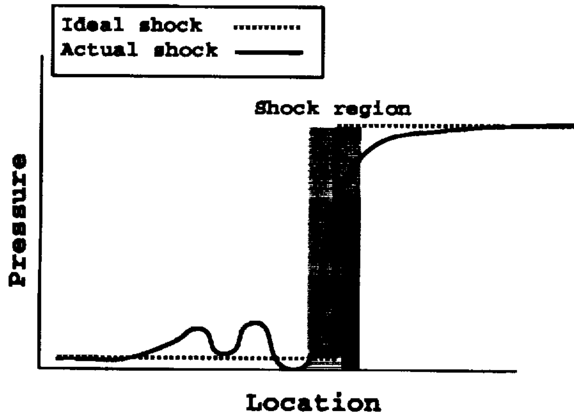


Fig. 1 Shock inflection points

The second method is the one advocated in this paper, which is to use the pressure gradient to find the value of the mach number normal to a shock. The approach was outlined by David Darmofal[3] and also implemented in Plot3D.[5] Since the shock surface normal will be aligned with the pressure gradient vector, the mach number in the direction of this vector is the normal mach number. A shock is then located where this normal mach number is greater than or equal to one.

One disadvantage of this shock finding method is that it will fail to capture some moving shocks in transient solutions. The mach vector is calculated in the CFD solution relative to the model frame of reference. But the shock calculation needs the mach vector in the shock frame of reference. For example, a shock traveling in a shock tube can exist when the fluid in the tube is moving at a velocity less than the speed of sound. Only when the mach number is calculated with respect to the moving shock would this shock be detected. Fortunately, correction terms can be added to the basic equation to compensate for the change in reference frame and make the detector valid for moving shocks.

Shock Visualization

Shock waves in invicid flow have a thickness on the order of a few molecular mean free paths, so they can be visualized as surfaces in a three dimensional flow. In moving to a three dimensional numerical model, the previous work by Ma, Rosendale, and Vermeer[1] has tried to preserve the surface representation. Similar work by Pagendarm and Seiz[4] refers to shocks as "invisible surfaces" that should be made visible by computer rendering. But, because of the descritization and numerical effects of modeling such sharp discontinuities, the shock location is actually spread out into a three dimensional region of space, not just a surface. Even in areas of strong shocks and highly refined grids, this region often looks like a slice of the model volume, with two close and almost parallel surfaces enclosing the shock. The boundaries of the region are not guarenteed to enclose the entire discontinuity, but it will lie within that region. Both Pagendarm and Ma, Rosendale[1,4] try to resolve this problem by applying a threshold to the pressure or density gradient magnitude to filter out one of the surfaces. Hesselink, Levy, and Batra[6] take an alternative approach to filtering. They reject all the triangles in the shock surface that have normals that are not aligned with the pressure gradient. However, filtering out one surface is not always desirable since the thickness and shape of the region conveys information about the model, the solution, and how well the shock has been detected. Elements marked in the shock region can also be used in a mesh refinement procedure. The only way to collapse the shock region into a surface is to replace the computed variation of temperature density and velocity in this region with a sharp step function that does not violate the mass and momentum conservation principles. However this is not practical since the detected region does not correspond to the actual extent of the shock, making the jump in density, temperature and pressure across the shock is difficult to find. Even though physical shocks can be thought of as surfaces, shocks in numerical models are regions in space.

Stationary Shocks

The stationary shock algorithm was developed with a knowledge of the shock geometry shown in figure 2.

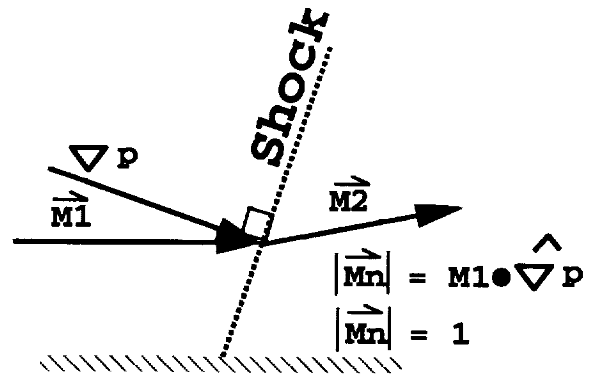


Fig. 2 Shock detection test quantity

For any shock, the mach number normal to the shock has a value of at least one just before the shock. This normal mach number can be computed on each node and used as a test value for determining the shock location. The pressure gradient is always normal to the shock, so it was used to find the shock orientation. The pressure gradient was approximated for each node, and then dotted with the mach vector to calculate a shock test value at each node. The locations where the test value equals one forms a boundary surrounding the shock location.

This test excludes areas of expansion, since the pressure gradient and the mach vector will have opposite senses, and their dot product will be less than zero.

For three dimensional models, an iso-surface of $M_n = 1$ was used to visualize the results. The shock feature is surrounded by the $M_n = 1$ iso-surface, and has a thickness associated with it. In the two dimensional case, contours of the normal mach number were created, and the $M_n = 1$ curve forms a boundary for a shock region.

Test Cases

Grid Study

The supersonic ramp test case had the geometry of figure 3. The shaded area is the region of the model that was used to plot the results and grids.

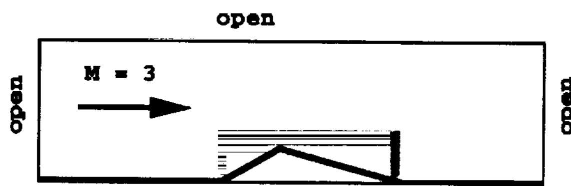


Fig. 3 Ramp model

Three different uniform grids were generated for the same geometry to determine the sensitivity of the detector to grid size. Figures 4, 5, and 6 show the grids that were used for the experiment.

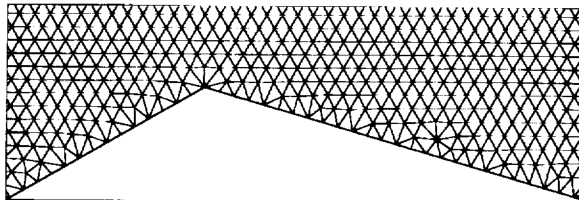


Fig. 4 coarse grid model

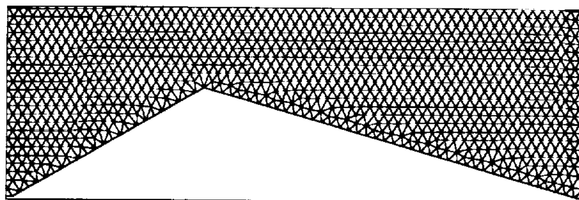


Fig. 5 fine grid model

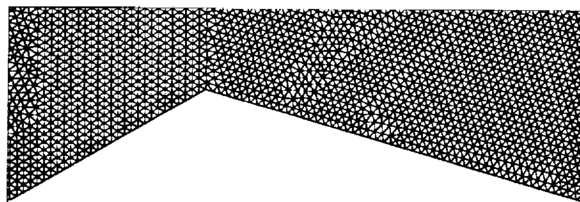


Fig. 6 20000 element model

The results of the three runs were similar except for the thickness of the indicated shock. The larger the element size, the larger the indicated shock. Figures 7, 8

Number of elements	Shock thickness (num cells)
5272	3
13801	3
20942	3

Table 1 Grid study results

and 9 are contour plots of the normal mach number, starting at Mn =1 in the region of the wedge.

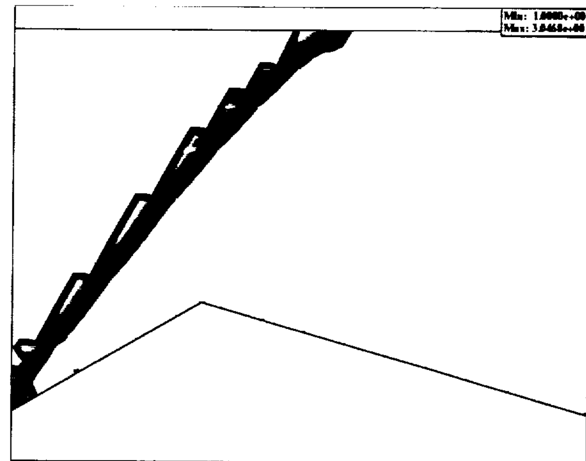


Fig. 7 Shock countours on the coarse grid

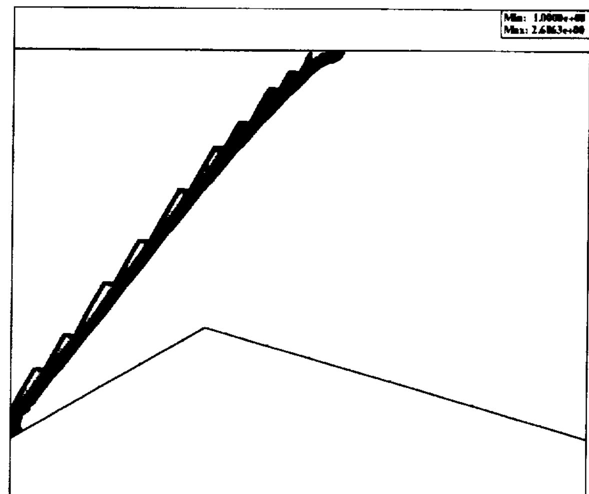


Fig. 8 Shock countours on more refined grid

The results in table 1 show that the shock algorithm displays shock thickness as a linear function of element size. The shape of the shock region not only locates the shock, but can point to a lack of mesh refinement in the area.

The numerical model used to solve the problem also greatly effects the shape of the shocks. Some CFD solution algorithms are better suited to capturing the sharp discontinuities. The larger the effective dissipation, the more spread out the shock and the more cells involved.

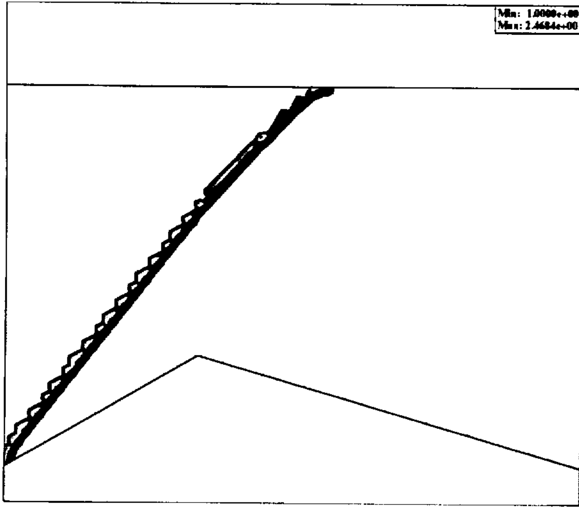


Fig. 9 Shock countours on highly refined grid

Transient Modifications

The assumptions made in constructing the previously described shock finding technique no longer apply when the shock is moving. To locate a shock with the previous method, it is necessary to assume that the observer is traveling with the shock and all velocities are measured with respect to this translating frame of reference. However, since all velocities are calculated in the model frame, there has to be a correction applied to the test equation to account for the moving shock. The equation 1 shows what this term must be, basically a time derivative of the pressure.

$$\frac{1}{|\nabla P|}\frac{1}{a}\frac{Dp}{Dt} = \frac{1}{|\nabla P|}\frac{1}{a}\frac{dp}{dt} + \frac{\vec{M}\cdot\nabla P}{|\nabla P|} \qquad (1)$$

It is more computationally expensive to approximate time derivatives directly, since that would require the storage of multiple time steps. So, the time derivative of pressure was calculated based on relations that equate it to a spacial variation of the state variables. Equation 2 applies to isentropic flows.

$$dp = a^2 d\rho \qquad (2)$$

This equation is then used along with the conservation of mass equation to produce an equation for an invariant test quantity that can be used to locate moving shocks.

$$\frac{dp}{dt} = -a^2 \nabla\cdot(\rho\vec{q}) \qquad (3)$$

From equations 2 and 3, equation 1 becomes:

$$\frac{1}{|\nabla P|}\frac{1}{a}\frac{Dp}{Dt} = -a\frac{1}{|\nabla P|}\nabla\cdot(\rho\vec{q}) + \frac{\vec{M}\cdot\nabla P}{|\nabla P|} \qquad (4)$$

A shock is then located when this quantity equals or exceeds one.

In the general case, pressure can be related to the internal energy and velocity of the flow with equation 5.

$$p = (\gamma-1)[\rho E - \frac{1}{\rho}(\rho\vec{q})\cdot(\rho\vec{q})] \qquad (5)$$

Taking the time derivative of this quantity will yield the required correction term, equation 6.

$$\frac{dp}{dt} = (\gamma-1)[\frac{d}{dt}(\rho E) - \vec{q}\cdot\frac{d}{dt}(\rho\vec{q}) + \frac{1}{2}q^2\frac{d\rho}{dt}] \qquad (6)$$

Substituting equation 3 into equation 6, and replacing the time derivatives with their equivalents from the mass and momenum equations yields equation 7.

$$\frac{dp}{dt} = (\gamma-1)[-\nabla\cdot(\rho\vec{q}H) + \vec{q}\cdot(\nabla P + \nabla\rho\vec{q}\vec{q}) - \frac{1}{2}q^2\nabla\cdot(\rho\vec{q})] \qquad (7)$$

This is the generalized correction term needed for transient problems.

Test Cases

Translating normal shock in a tube

A model of a moving normal shock in a channel was created to investigate the behavior of the shock finding algorithm and the effect of the transient modification. Two separate runs were done, the first had the initial conditions shown in the figure 10.

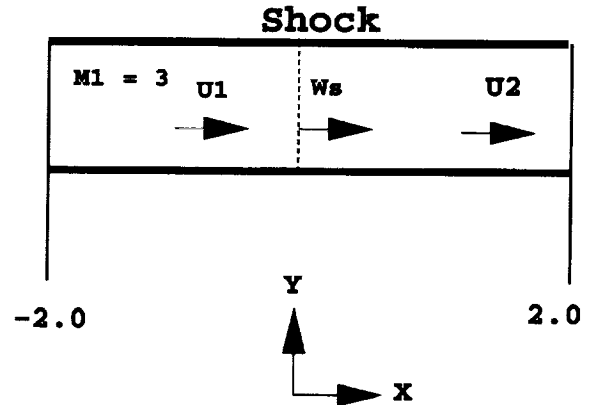


Fig. 10 Transient shock model

Shock relations were used to set up the initial conditions for the model. The formulas for moving normal shock waves with constant C_p and C_v were applicable in this case. However, these formula assume that the upstream speed, U_1 is 0, so a correction had to be made to produce the correct initial conditions for the flow. For the first run, the pressure ratio was chosen to be less than the pressure ratio required for a standing shock in M=3 flow (10.33). This required that the shock move toward the right in the positive X direction. The speed of the shock traveling into a stationary fluid, W, was calculated with the equation 8. Since the upstream velocity was not zero, the actual speed of the shock had to be corrected with equation 9.

$$W = a_1\sqrt{\frac{\gamma+1}{2\gamma}(\frac{P_2}{P_1}-1)+1} \qquad (8)$$

$$W_s = U_1 - W \qquad (9)$$

The shock test scalar was calculated with the isentropic transient correction equation (equation 4), that was simplified for this particular example. The Y component of velocity was assumed to be zero in this case, so the divergence term simplified to the following.

$$\frac{1}{|\nabla P|}\frac{1}{a}\frac{Dp}{Dt} = -a\frac{1}{|\nabla P|}(\frac{d}{dx}u\rho) + \frac{\vec{M}\cdot\nabla P}{|\nabla P|} \qquad (10)$$

The derivative of the density times the X velocity was then expanded, yielding the final equation.

$$\frac{1}{|\nabla P|}\frac{1}{a}\frac{Dp}{Dt} = -a\frac{1}{|\nabla P|}(\frac{du}{dx}\rho + \frac{d\rho}{dx}u) + \frac{\vec{M}\cdot\nabla P}{|\nabla P|} \qquad (11)$$

Results indicated that the pressure variation across the shock has some interesting features that are problematic for the shock finding algorithm. The shock started at X=0, and is moving to the right in the positive X direction. As it moves, the shorter wavelengths that makeup the initial discontinuity move at a slower speed than the longer wavelengths. This difference in speed is a numerical artifact of the time stepping method used in the CFD solver. As time progresses, high frequency pressure oscillations show up behind a moving shock, as shown in the figure 11.

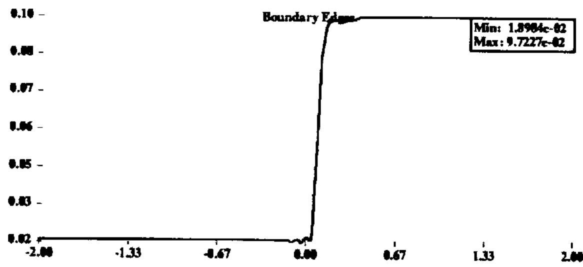


Fig. 11 Pressure distribution

Figure 12 is a plot of the shock scalar with the isentropic transient correction. A filter has been applied to eliminate the pressure gradients caused by dispersion. The dashed line on this and following figures represents the threshold for a shock. Everything above one is marked as a shock region.

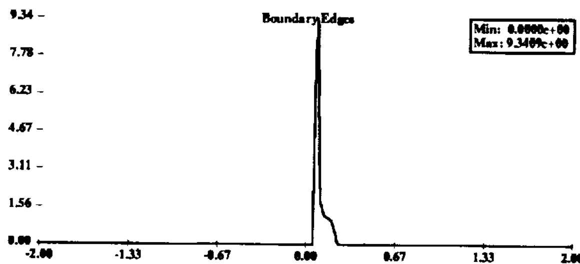


Fig. 12 Shock scalar

Figure 13 is a plot of the same quantity after a few more iterations. The shock is clearly moving to the right, as expected.

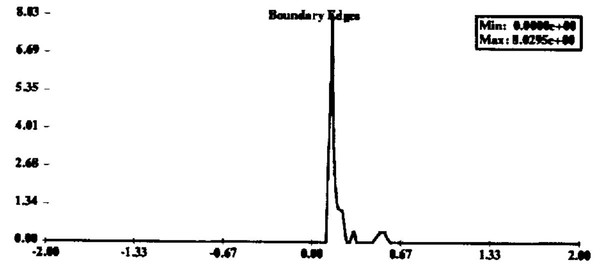


Fig. 13 Shock scalar at delta t

The results of this experiment point to the importance of choosing a threshold value for the magnitude of the pressure gradient. Because of the slower wave speeds of the higher frequency pressure waves, oscillations in the pressure gradient take place behind the shock. These pressure gradient increases are enough to skew the results and show up in the shock detection values as a group of shocks behind the actual position as shown in figure 14.

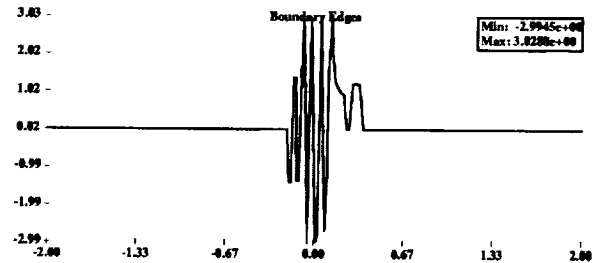


Fig. 14 Shock scalar on unfiltered pressure gradient

From the above experiment, it was noted that it did not matter if the transient correction was used or not, a shock would still be indicated. This was because the upstream mach number was greater than one, making the uncorrected normal mach number greater than one. A change in initial conditions was made to see what happens when the upstream mach number is less than one. The upstream mach number was set to 0.9,and the pressure difference was set to 3.0, which will produce a normal shock moving in the negative X direction of figure 10.

The results are shown in the following two figures. In the figure 15, the normal mach number is plotted against the x-axis. Notice that the normal mach number does not exceed one, which by the previous test indicates that no shock is present in the flow. However, the shock test scalar in figure 16 does exceed one at the shock, indicating that the shock is indeed present. The correction term has made it possible to locate the shock.

The experimental results were then compared to the theoretical location of the shock at any given time, calculated with the following equation, where x is the location, t is time and W_s is calculated with equations 8 and 9.

$$X = W_s t \qquad (12)$$

Figure 17 is a plot of the difference between theo-

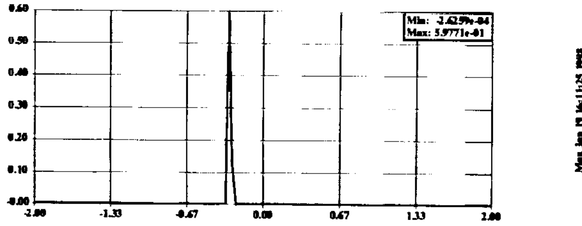


Fig. 15 Normal mach number vs location

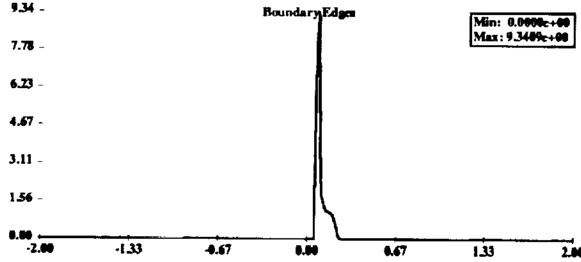


Fig. 16 Shock test variable vs. location

retical and measured shock location vs. time. The experimental shock location was measured two ways, first by the location of the maximum pressure gradient magnitude, and secondly by central location of the shock region.

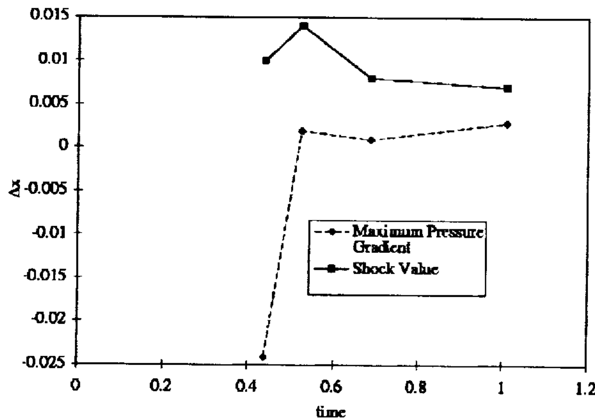


Fig. 17 Theoretical vs. experimental shock location

Since the shock region actually covers about 3 cells in width, or 0.09 units, the theoretical shock location and pressure gradient maximum both occur well within the region at all times.

Filtering

The previous results show that this shock finding algorithm produces some falsely indicated shock regions. This is partially due to small numerical errors in the gradient away from the shock. As the gradient of pressure goes to zero in the far field, errors in the shock test function build as shown in the following equation, where δ is a small variation in the quantity.

$$\frac{M_n(\nabla P + \delta \nabla P)}{|\nabla P + \delta \nabla P|} \rightarrow \frac{\delta \nabla P}{|\delta \nabla P|} \rightarrow O(1) \qquad (13)$$

To remove these false indications, three filtering techniques have been applied to the problem. The techniques start with an iso-surface constructed where the value of the shock test scalar is 1.0 and reject a subset of the triangles that makeup that surface. The first technique enforces the property that the pressure gradient is normal to the shock surface. This technique removes all shock surface triangles that do not pass the test in the following equation, where n is the normal of the surface triangle, and c is a threshold value between 0 and 1.

$$\frac{|\nabla P \cdot n|}{|\nabla P|} \leq c \qquad (14)$$

The second technique removes all iso-surface triangles that fall below a certain pressure gradient magnitude threshold. This technique is based on the fact that shock discontinuites should only occur in regions of relatively high gradients. The problem is to determine the meaning of a 'relatively' large gradient and set an appropriate threshold. To accomplish this, all the triangles were divided into groups based on the value of the pressure gradient magnitude at their centers. The count of triangles in each group forms a curve. The threshold was chosen where the derivative of this curve goes to zero. This method of setting a threshold assumes that the actual shock surface is located in a region of high gradients and that changing the threshold by some small amount will not change the number of triangles in the surface.

Note that setting a pressure gradient magnitude threshold can be done before the shock iso-surface is constructed by applying the filter to the nodal shock scalar values. If the node does not pass the pressure gradient magnitude test, the shock value can be set to 0. Applying a filter to the nodal values has the advantage of producing a more connected shock surface, without disjoint triangles missing.

A third technique removes all the triangles that do not have jumps in density and temperature that correspond to normal shock relations. For a moving normal shock, equations 15 and 16 state the relationships between the pressure jump across the shock and density and temperature ratio, respectively.

$$\frac{\rho_2}{\rho_1} = \frac{1 + \frac{\gamma+1}{\gamma-1}\frac{p_2}{p_1}}{\frac{\gamma+1}{\gamma-1} + \frac{p_2}{p_1}} \qquad (15)$$

$$\frac{T_2}{T_1} = \frac{p_2}{p_1}\frac{\frac{\gamma+1}{\gamma-1} + \frac{p_2}{p_1}}{1 + \frac{\gamma+1}{\gamma-1}\frac{p_2}{p_1}} \qquad (16)$$

The difficulty with this filtering technique is that the extent of the shock needs to be known before hand, so that the pressure, temperature and density ratios can be accurately measured. To overcome this problem, a number of measurements of the pressure temperature and density were made on both sides of the surface triangles, then the two measurements that best fit the shock relations were used to determine if

the triangle should be rejected. A fitness function was constructed that, given the measured pressure ratio at two points on either side of the triangle, compares the measured temperature ratio to the temperature ratio computed with equation 16 and the measured density ratio with the computed density ratio of equation 15 and produces a value of 1.0 in the case where the measurements match theory, and less than 1.0 otherwise.

$$f(p_1, p_2, T_1, T_2, \rho_1, \rho_2) <= 1.0 \qquad (17)$$

If none of the points on either side of the triangle can produce a fitness value of greater than some threshold, the triangle is removed.

To compare the filtering techniques, a measure of difference between the shock test value fields was constructed. The comparison works from a baseline that is assumed to be the actual shock, then the comparison is run against the results of the unfiltered algorithm and the solution with the various types of filtering. The difference measure is constructed by gathering the centerpoints of all the triangles in the shock surface, F, then all the points that have a shock value greater than 1 in the basline and test solution are removed from F. The total difference value is the sum of the absolute value of the difference between the values of the baseline and test solution at the points in C. The final comparison function takes the total difference and divides it by the final number of triangles in the shock surface.

$$C = F - B \bigcap F \qquad (18)$$

Figures 18 19 20 show the results of applying some of the filtering techniques, and why they are necessary. In this case, the solution being examined is an invicid converged flow on a 300,000 element f18 aircraft model traveling in subsonic flow. Because the solution has run to convergence, the stationary and transient shock finding algorithms should produce the same results, which as figure 19 shows, is not the case. The transient algorithm generates noise and false indications. However, these false signals can be removed with filtering to produce a shock surface that is very similar to the stationary shock finding results. Figure 20 shows the results of running the transient shock finder through a pressure gradient magnitude filter.

Table 2 is a collection of the results from passing the transient shock finder through various combinations of the three filtering algorithms.

The results in table 2 indicate that the pressure gradient magnitude filter alone is better than when used in combination with the other two. This became more evident in looking at the graphical output, where there would be a fairly continuous shock surface at the end of the pressure gradient magnitude filter, with some additional outliers. Applying the normal filter or jump condition filter to these results did not get rid of the outliers, but only removed some of the surface triangles enclosing the shock region, making the final result

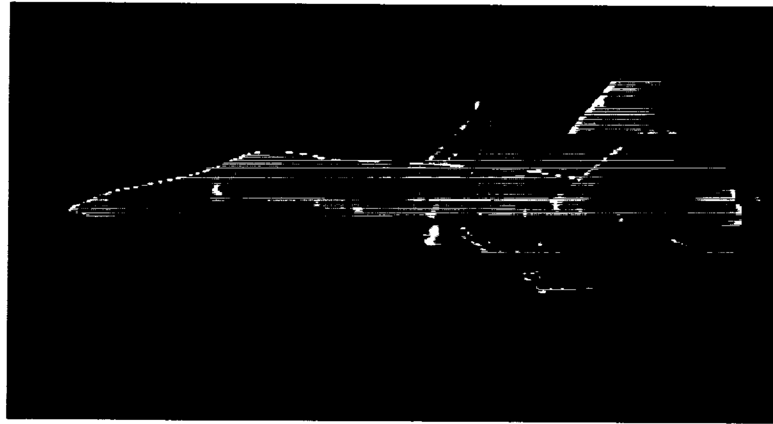


Fig. 18 Stationary shock algorithm on converged solution

Fig. 19 Transient shock algorithm without filtering

Fig. 20 Transient shock algorithm with pressure gradient magnitude filtering

test case	Comparison function value
no filtering	0.57
$\lvert \nabla P \rvert$ filter	0.105
$\lvert \nabla P \rvert$ and normal filter	0.223
$\lvert \nabla P \rvert$ and jump filter	0.104
jump filter	0.567

Table 2 Filtering study results

worse than than the result from the pressure gradient magnitude filter alone.

Transonic Aircraft

The transient modifications were tested on a relatively large (1 million element) model of an aircraft traveling in transonic conditions (mach 0.85) to see if the unsteady shock finding algorithm could produce useful results on a realistic model. Figure 21 shows the results of applying the steady state shock finding algorithm to the converged solution. An iso-surface has been constructed where the normal mach number equals one, and this iso-surface is painted with the pressure gradient magnitude as an indication of the shock strength. Note that the thickness of these shock regions is quite small due to the highly refined mesh. Figure 22 is of the same model, but the iso-surface is now plotted on the results from the unsteady shock algorithm.

Since the solution has converged, the unsteady and steady shock results should be identical. While this was true of the larger shock features, the unsteady algorithm also produced some false indications. It was necessary to threshold out these regions with the pressure gradient magnitude. Even when this was done, some false indications remained, especially at the leading and trailing edges of the wing. This result was in agreement with the one dimensional results, where it was also necessary to filter out the gradients caused by dispersion in the results.

The shock finder captured some fairly complex shock structures that would be difficult to find almost any other way.

Conclusions

The stationary shock finding algorithm does not produce a shock surface that would reflect the shape of the shock in the physical flow, but because of numerical approximation, shows a shock region. The thickness of the region can give information about the quality of the solution and location of needed mesh refinement. Any disperson and dissipation in the solution is reflected in the shape and size of the shock region.

The nature of the solutions and the suceptability of the shock detector to small errors in the solutions makes filtering a requirement, especially in transient solutions. Filtering to enforce jump conditions and shock direction is not as attractive and effective as a simple threshold on the pressure gradient magnitude. The heuristic method of automatically determining this threshold that is presented in this paper was effective on the models tested.

Shock finding based on fluid dynamic principles is practical, with some advantages over more general edge detection, but still requires filtering of the results.

Acknowledgments

This has been an extension of the work done by David Darmofal[3] and professor Mark Drela was instrumental in finding the unsteady extensions to the basic algorithm.

Funding for this project has been provided by NASA Ames, through grant number NCC2-985, and by the Army under grant DAAG55-97-1-0394. Additional support for this research was obtained from the IBM SUR Project and the IBM UUP Project.

References

[1]Kwan-Liu, Ma Van Rosendale, John Vermeer, Villem "3D Shock Wave Visualization on Unstructured Grids"

[2]Anderson, John D Jr.,"Modern Compressible Flow with Historical Perspective" McGraw-Hill, 1982

[3]Darmofal, David,"Hierarchal Visualization of Three-Dimensional Vortical Flow Calculations" MIT 1991

[4]Pagendarm, H-G B. Seitz: in P Palamidese(ed.):"Visualization in Scientific Computing" Ellis Horwood Workshop Series, 1993

[5]Walatka, P.P. Burning, P.G. Elson, Pierce L.,"Plot3D Users Manual, NASA TM 101067" July 1992

[6]L. Hesselink, Y. Levy, Rajesh Batra,"Automatic Flow Feature Extraction for use in Computational Steering of Aerodynamic Design Processes" http://www-leland.stanford.edu/ rbatra/ics/ics.html

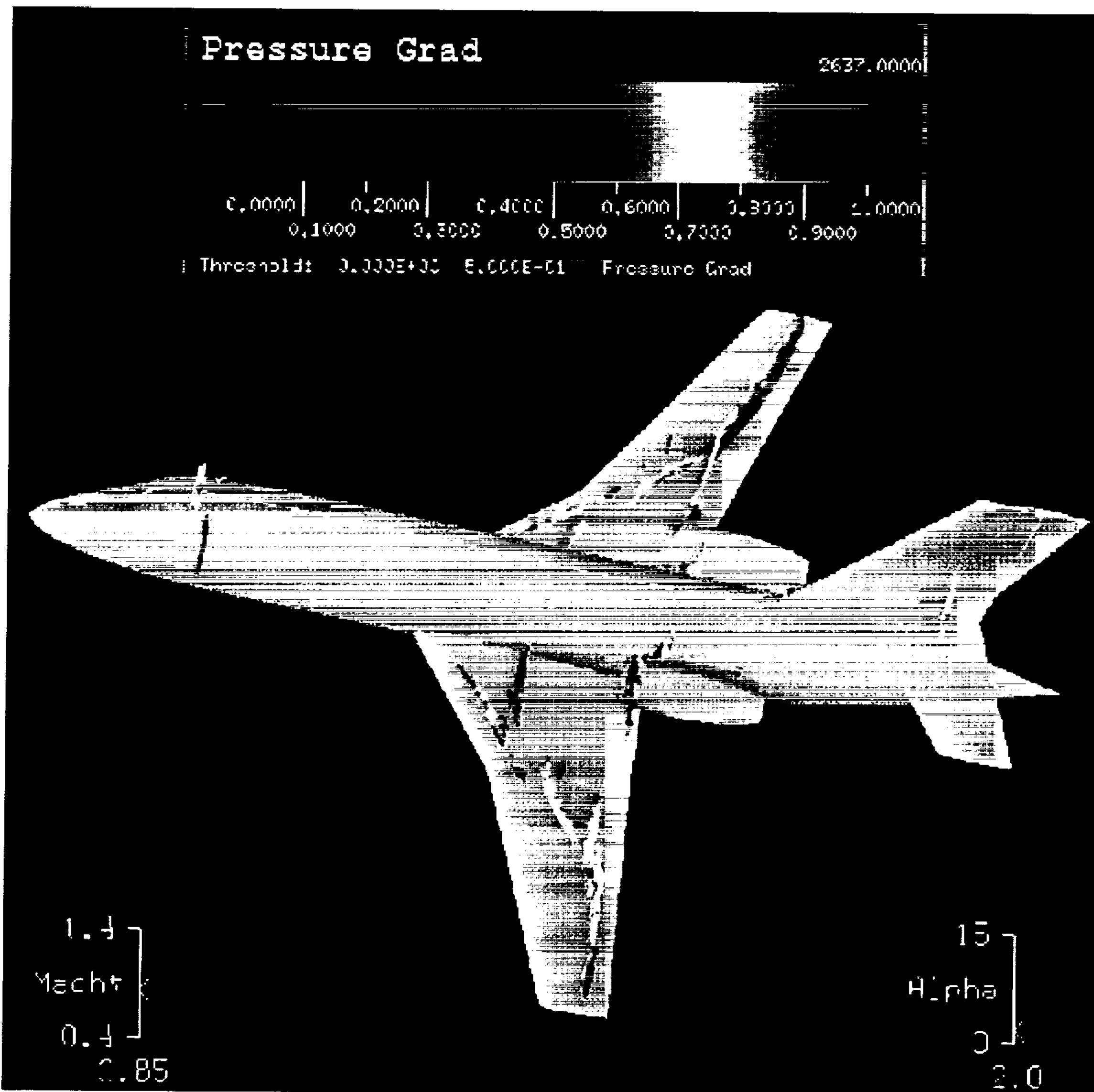


Fig. 21 Steady shock detection results

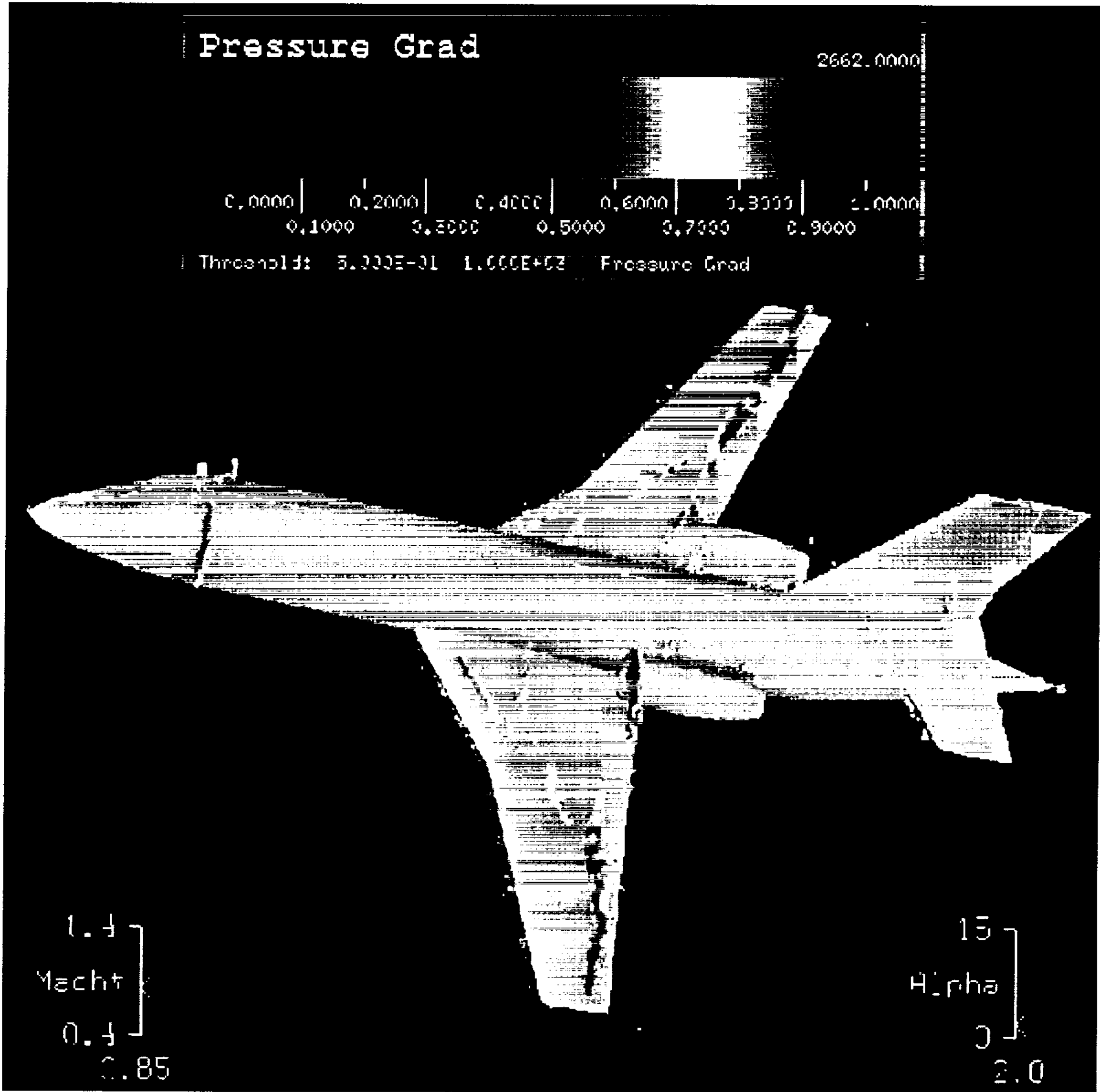


Fig. 22 Unsteady shock detection results