Viper

Spectral-element flow solver

Reference Manual

Dr Gregory J Sheard

20 February 2009

Table of Contents

Chapter 1: Overview

This Chapter provides an introduction to both the Viper package itself, as well as this manual. In addition, the Getting Started guide describes what is required to begin using Viper.

Background theory behind the numerical algorithms implemented by Viper is described in *Chapter 2*. In *Chapter 3*, mesh generation and conversion is described. In *Chapter 4*, the configuration of simulations is described, and *Chapter 5* details the execution of simulations and the solution methods employed by the solver. *Chapter 6* treats the visualization and post-processing of data, and *Chapter 7* describes each of the commands available to use within Viper. A bibliography for further reading is provided in *Chapter 8*.

About Viper

Viper is a Computational Fluid Dynamics (CFD) package that solves the time-dependent incompressible Navier—Stokes equations in either two or three dimensions.

Viper uses a spectral-element method to discretize the Navier—Stokes equations in space, and employs a third-order accurate backwards multistep method to evolve the solutions in time.

Audience for this Manual

This manual is intended for users of the Viper software – it contains descriptions of the commands and functionality of the Viper package, as well as information on how to generate and convert meshes for simulation, and how to extract and process useful data from the computed solutions.

Readers are assumed to have an Undergraduate-level background in fluid mechanics. This is not a Developer's Manual – no information about the underlying source code is provided. Readers will not find details about the subroutines, variables and modules behind the package, but they will find information about third-party source code contributions and libraries that Viper employs.

Getting Started

To run simulations, users will need the Viper executable (the latest executable files compiled for various platforms are available from http://www.flair.monash.edu.au/viper/). Windows users will also require the Tecplot Dynamically Linked Library **`tecio.dll`**, also available from the same URL.

By default, Viper searches for a configuration file **`viper.cfg`**, and if this file is not located in the current directory, the user is prompted to supply an alternative path/file name. The contents of the configuration file are described in *Chapter 4*.

Once the configuration file is found, Viper processes the commands given in the file to establish the conditions for the simulation. The configuration supplies the mesh file name, and it establishes parameter values, initial and boundary conditions for the simulation. Once the configuration phase is complete, the user is prompted to supply input commands.

An example of a simple list of commands to execute a simulation is as follows:

```
init
step 100
save
tecp
stop
```

These commands do the following: Initialise a simulation to allow time integration to proceed (`init`), integrate forward in time by 100 time steps (`step 100`), save the flow field solution to a default file `ff_out.dat` (`save`), output a binary file for post-processing and plotting using the Tecplot visualization package (`tecp`), and exit Viper (`stop`). A detailed description of all of the available commands recognised by Viper is given in *Chapter 7*.

Rules for inputting text into Viper

Viper employs text input and processing routines that allow for comments, and permit numerical values to be entered in any format recognised by FORTRAN. The same rules apply for command line input as well as macro and configuration file input:

- **Commented lines: I**f a line begins with a "`#`" followed by a space, it is regarded as a comment, and is ignored by Viper. Note: The blank space following the hash is essential. E.g.,
  ```
  # This is a comment
  #This is not a comment
  ```

- **Comments within a line:** If the user wishes to add a comment within a line, then they can do so by enclosing text in round brackets: "`(`" and "`)`". E.g., The following text would be read as "`Viper reads this, but not this.`"
  ```
  Viper reads this, but (Viper ignores this)not
  this.
  ```

- **Numerical input:** If users wish to enter an integer, it can be entered with or without a negative sign, but can only contain numbers (no decimal points, alphabetical characters, etc.). E.g., The following are valid integers:
  ```
  1
  34
  796954
  -343
  ```
  The following are invalid as integers: and may either be rounded by the code, or cause an error, so should be avoided. If floating-point numbers were required, then the following are all valid:
  ```
  .1
  3.
  -4.5
  4.1e-10
  ```

- **Case sensitivity:** Linux systems are case sensitive, whereas Windows systems are not, allowing upper- and lower-case characters to be substituted at will. Therefore, when processing input and output filenames, Viper preserves the capitalization specified by the user. If a user wishes to load a file "`Macro.txt`" and enters “`mACRO.TXT`”, the file will not be found under Linux, resulting in an error, whereas under Windows the file will be located and input without an error. Internally, Viper converts all input variable names to lower case, so users should be aware that under Linux, Viper makes no distinction between variables with the same name, but different capitalisation: i.e., “`DT`” is treated as “`dt`”.

- **Verbatim text:** To input a string of characters as a single entry, the text should be enclosed by single quotes. This is especially important to avoid brackets in mathematical expressions being confused with an in-line comment, or blanks being confused for the end of the function. E.g. 1: Viper would misread `sin(23*x)` as `sin`, ignoring the bracketed component, whereas it would be input in full if expressed as `'sin(23*x)'`. E.g. 2: Viper would misread `y*t + x^2` as `y*t`, ignoring the component after the blank, whereas it would be input in full if expressed as `'y*t + x^2'`.

Rules for inputting mathematical expressions into Viper

A powerful feature of Viper is the ability to read mathematical expressions input by the user at run time, and evaluate them. Viper employs this capability for the processing of user-defined boundary conditions, functions, initial conditions, integrands for L_2 norms, etc.

Important: If a function is incorrectly structured, or is evaluated incorrectly (e.g., due to an incorrect variable name being supplied), it MAY NOT return an error, and the output will be incorrect. Care must be taken to ensure that functions are input correctly.

The following information outlines the allowable components of mathematical expressions:

Mathematical operators:

Operator	Function
`+`	Addition E.g., `11+24.5`
`-`	Subtraction E.g., `58.5 - 1e3`
`*`	Multiplication E.g., `7.5*t`
`/`	Division E.g., `23/4`
`^`	Power E.g., for x^2, type `x^2`

Parentheses:

Users may enclose parts of their expressions in pairs of round, square, or curly brackets: All opening brackets must have a corresponding closing pair. E.g., `(…)`, `[…]`, `{…}`.

Mathematical functions:
A large number of mathematical functions are available, which form a superset of the intrinsic mathematical functions available in Fortran.

Class	Function	Syntax
Trigonometry	Sine of x	**`sin(x)`**
	Cosine of x	**`cos(x)`**
	Tangent of x	**`tan(x)`**
	Inverse sine of x, $\|x\| \leq 1$	**`asin(x)`**
	Inverse cosine of x	**`acos(x)`**
	Inverse tangent of x	**`atan(x)`**
Hyperbolic	Hyperbolic sine of x	**`sinh(x)`**
	Hyperbolic cosine of x	**`cosh(x)`**
	Hyperbolic tangent of x	**`tanh(x)`**
	Hyperbolic cosecant (1/sinh) of x	**`csch(x)`**
	Hyperbolic secant (1/cosh) of x	**`sech(x)`**
	Hyperbolic cotangent (1/tanh) of x	**`coth(x)`**
Logarithms and exponentials	Base 10 logarithm of x, where $x > 0$	**`log10(x)`**
	Natural logarithm of x, where $x > 0$	**`log(x)`**
	Logarithm of x (base n, where $n > 0$ and $x > 0$)	**`logn(x,n)`**
	Exponential number raised to the power x	**`exp(x)`**
Other	Square root of x, $x \geq 0$	**`sqrt(x)`**
	Absolute value of x	**`abs(x)`**
	Maximum value of x or y	**`max(x,y)`**
	Minimum value of x or y	**`min(x,y)`**
	Delta function (1 if $x = 0$, 0 otherwise)	**`delta(x)`**
	Step function (0 if $x < 0$, 1 otherwise)	**`step(x)`**
	Hat function (1 if $\|x\| \leq 0.5$, 0 otherwise)	**`hat(x)`**
	Round to nearest whole number	**`anint(…)`**
	Random number in the range $[0, x)$ Note: The result of this function is treated as always time- and space-varying	**`rand(x)`**

The code also facilitates a number of Bessel, Gamma and Error functions, though these are not presently fully implemented in the code. See Dr Greg Sheard if you require these functions.

Finally, conditional statements can be input using the function

`if( condition, then, else ),`

which evaluates the conditional statement **`condition`**, and then evaluates the expression **`then`** or **`else`**, when conditional statement is true or false, respectively. The conditional statement can be constructed using the following relations:

Condition	Symbol
Less than (<)	**`<`**
Less than or equal to (≤)	**`<=`**
Greater than (>)	**`>`**

Greater than or equal to (≥)	`>=`
Equal to (=)	`=` or `==`
Not equal to (≠)	`!=`

Implicit and user-defined variables

A number of variable and parameter names are reserved by Viper. These include the spatial coordinates `x`, `y` and `z`, time `t` and time step `dt`, velocity components `u`, `v` and `w`, the kinematic static pressure `p`, the reciprocal kinematic viscosity `RKV`, and the shear rate `SR`. These variables can be used in mathematical expressions input into Viper either on the command line (such as during the `int` or `l2` commands), or in the configuration file (such as in `btag` statements). Users should consult the specific entries for each command to see which of the implicit variables are allowed.
In addition to the implicit variables, Viper also facilitates the creation of "user-defined variables". User-defined variables are defined using the `gvar_usrvar` statement in the configuration file, and assign a user-specified name to a number or mathematical expression to be evaluated at run-time. User-defined variables can appear in subsequent mathematical expressions, including within subsequent `gvar_usrvar` statements.

Unresolved Bugs

Memory leak during `init` routine:
Platforms: All
Symptoms: Each call to `init` adds a small amount of memory to the overall memory used by Viper. Not usually a problem, as it is small, and `init` is typically only called once per simulation. However, if users wish to run a macro with many calls to `init`, this issue will eventually lead to an exhaustion of available RAM.
Workaround: Limit the number of calls to `init` in each Viper session.

Resolved Bugs

Segmentation fault during `save` or `tecp` calls with large meshes – FIXED 15 AUGUST 2007:
Platforms: ia32 linux
Symptoms: If meshes have a very large number of elements (numbering in the thousands), then segmentation faults can occur during calls to `save` or `tecp`. This appears to be a compiler or O/S-related issue, as it cannot be reproduced on either ia64 linux or the ia32 Windows platforms.
Workaround: Use either the ia32 Windows or ia64 linux version of Viper to generate the simulation or Tecplot data.
Resolution:
This bug was actually caused by the code exceeding the available stack size on the Linux systems the code was running on. These segmentation faults can be avoided by increasing the size of the stack. On the Linux command line, type

```
        \> limit stacksize unlimited
```

or if the user is running a job from a script file, add this line to the script file prior to the command used to invoke Viper.

Crashing during `tecp` or `getminmax -e` calls – WORKAROUND 12 AUGUST 2007; FIXED 26 SEPTEMBER 2007

Platforms: ia64 linux

Symptoms: The **`tecp`** and **`getminmax -e`** commands call a routine to evaluate the strain rate magnitude. This routine uses eigenvalue/eigenvector routines to determine the rate-of-strain field. The problem appears to lie in the ia64 linux implementation of the Intel Math Kernel Library functions being called. The error does not always occur – it is flow-dependent somehow.

Workaround: Use either the ia32 linux or ia32 Windows versions of Viper.

Resolution (26 September 2007):

Consultation with APAC verified that this bug was caused by the Intel MKL routines performing IEEE arithmetic checking which were causing floating-point exceptions in the code. These have since been avoided by compiling with the `–fpe3` compiler option, which was being overridden by the APAC system.

Chapter 2: Background

This Chapter provides background theory for the fluid flow solvers and analysis tools implemented within Viper.

The Navier—Stokes Equations

The motion of all fluids is described by the Navier—Stokes equations. Applying a conservation-of-momentum principle yields

$$\rho\mathbf{g} - \nabla p + \nabla \cdot \boldsymbol{\tau}_{ij} = \rho \frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t},$$

where $\mathbf{g}$ is the gravity acceleration vector, p is a scalar pressure field, ∇ is the gradient operator, $\boldsymbol{\tau}_{ij}$ is the viscous stress tensor, $\mathbf{u}$ is a velocity vector, and t is time. The velocity time derivative is sometimes referred to as the substantial derivative, which is defined

$$\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u}.$$

The fluid must also satisfy a conservation-of-mass argument, which can be expressed as

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

Newtonian and non-Newtonian Fluids

A significant simplification to the momentum equation of the general Navier—Stokes equations is possible, if viscous stresses are assumed proportional to strain rates and the coefficient of viscosity, μ. For a simple shear flow, this can be written

$$\tau = \mu \frac{\mathrm{d}u}{\mathrm{d}y}.$$

Fluids that satisfy this assumption are classified as Newtonian fluids, and a remarkably large number of fluids are well-described by this relationship, including air and water. Fluids that do not satisfy this relationship are classified as non-Newtonian, and include many polymers, emulsions and suspension fluids, including blood.

Incompressible Flow

If the flow has constant density in space and time, it can be regarded as incompressible. If there is no fluid interface (such as a free surface), the gravity term can be omitted, as its action is constant everywhere in the flow. Combining this simplification with the incompressibility condition yields momentum and continuity for a Newtonian fluid

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla P + \nu \nabla^2 \mathbf{u}, \tag{1a}$$

$$\nabla \cdot \mathbf{u} = 0. \tag{1b}$$

where we introduce a kinematic pressure and kinematic viscosity

$$P = \frac{p}{\rho}, \nu = \frac{\mu}{\rho}.$$

Finally, equation (1a) can be used to reveal the single most important parameter describing the viscous behaviour of Newtonian fluids, the Reynolds number. The Reynolds number can be written

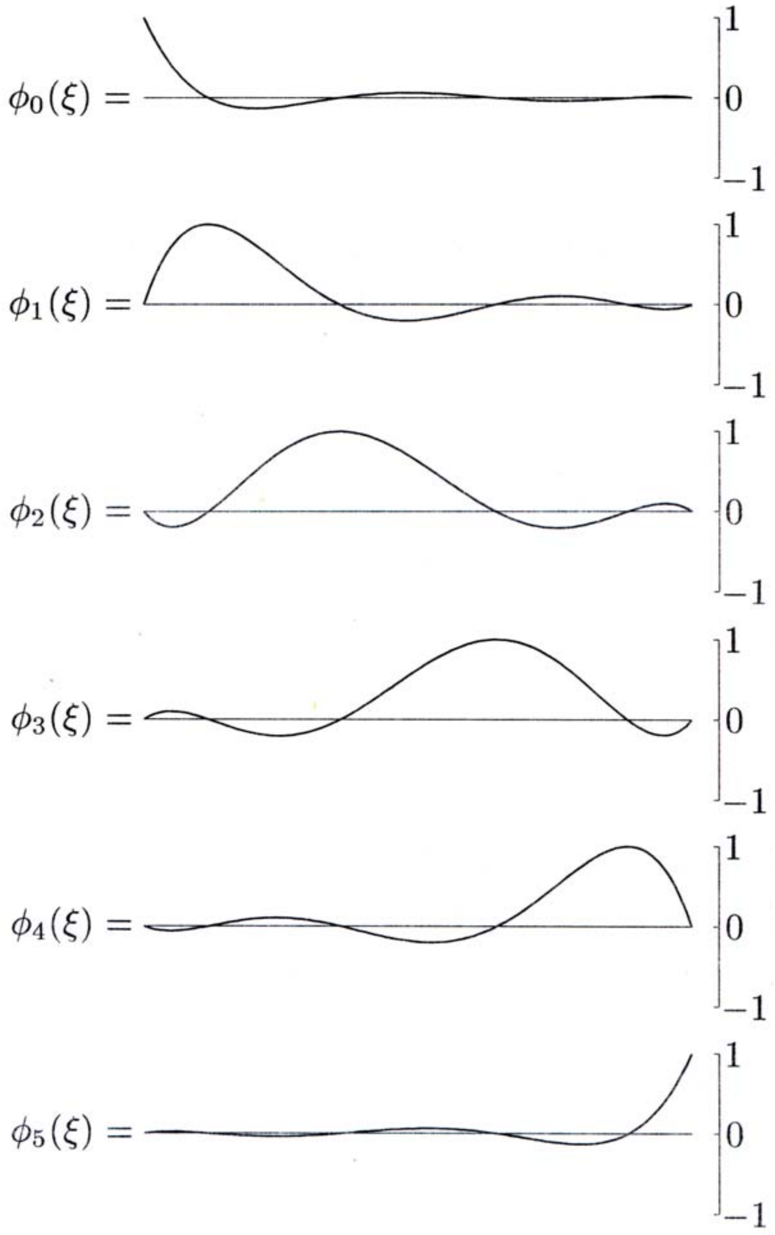
$$Re = \frac{U_\infty D}{\nu},$$

where U_∞ is a reference speed, and D is a reference length scale.
Equation (1a) comprises several terms, which from left to right are the velocity time derivative term, the advection term, the pressure term, and the viscous diffusion term. Viper solves this equation using an operator-splitting technique (Karniadakis, Israeli & Orszag 1991), where the advection, pressure, and diffusion terms are solved individually at each time step. This procedure will be described in more detail later.

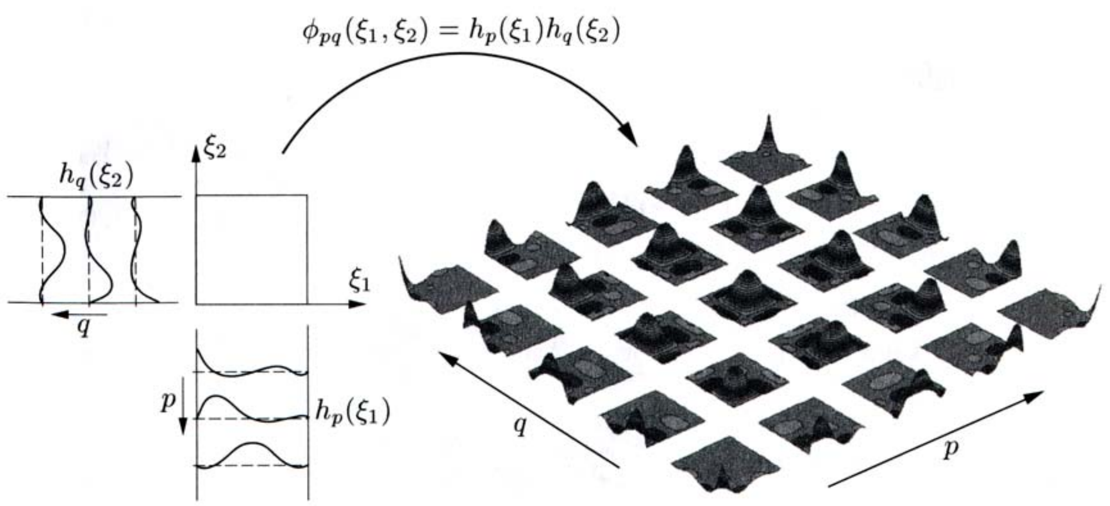
The Spectral-Element Method and Spatial Discretization

The spectral-element method is a class of finite element methods, which is used to solve partial differential equations by discretizing a spatial domain into small regions (elements), over which a high-degree polynomial basis is employed. This is an improvement over the traditional finite element method, which employs a piecewise linear basis.
The partial differential equations being solved are recast in weak form by applying the Galerkin method (a form of the method of weighted residuals). The Galerkin method replaces the continuous partial differential equation with an integral equation, which when approximated by numerical quadrature techniques, produces a set of ordinary differential equations which may be solved in a standard fashion.
Integration is performed within each element using highly efficient Gaussian quadrature methods, and the global solution is coupled between elements by enforcing a continuous solution across element interfaces.
The spectral-element method differs from the finite-element method in that higher-order functions are used as basis functions within each element, and efficient Gaussian quadrature rules can be employed within each element to approximate the integral contributions. Viper employs a *nodal* formulation, in which Lagrangian tensor-product polynomial basis functions are employed within each element. These functions are interpolated over a grid of points on each element, which correspond to the quadrature points for Gauss-Legendre-Lobatto (GLL) quadrature. The GLL quadrature points include points fixed at the element edges/faces to facilitate a

continuous solution between adjacent elements. In one dimension, GLL quadrature is exact for polynomials of degree 2*n*-3, where *n* is the number of quadrature points. Illustrations of the nodal polynomial expansion basis employed by Viper are shown below.

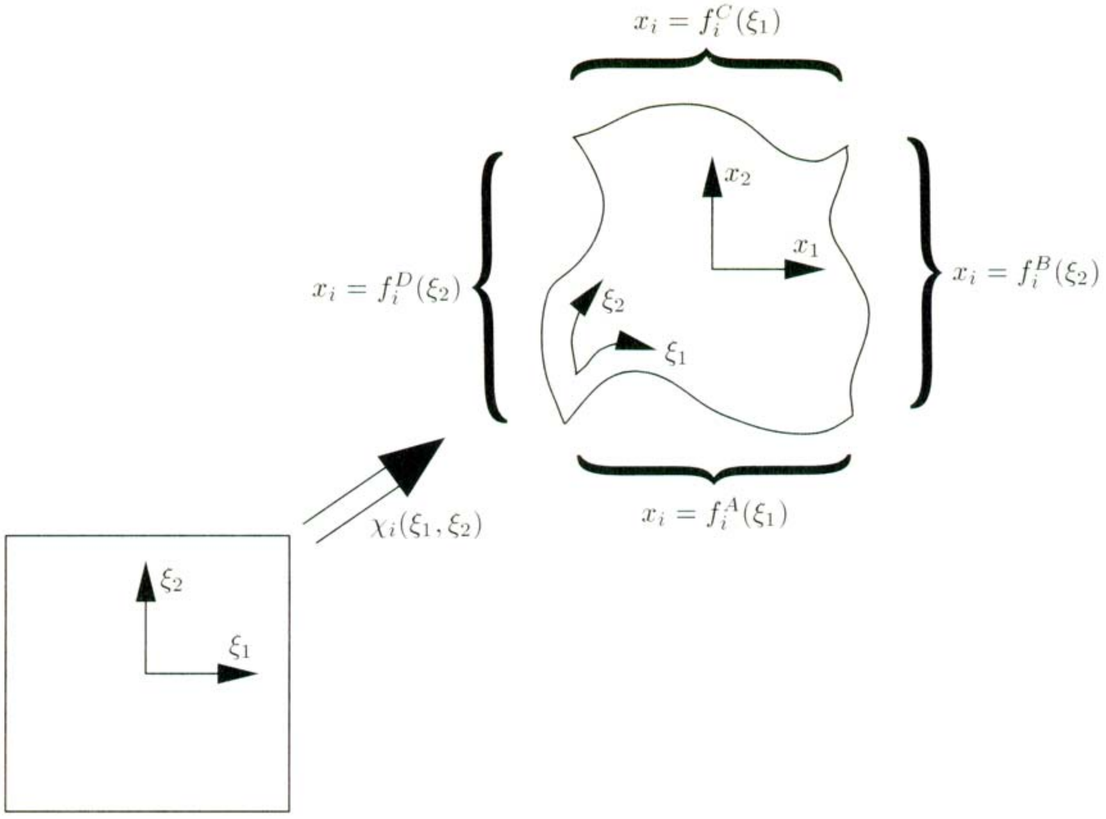


One-dimensional nodal expansion modes for a polynomial of degree 6 (from Karniadakis & Sherwin 2005).



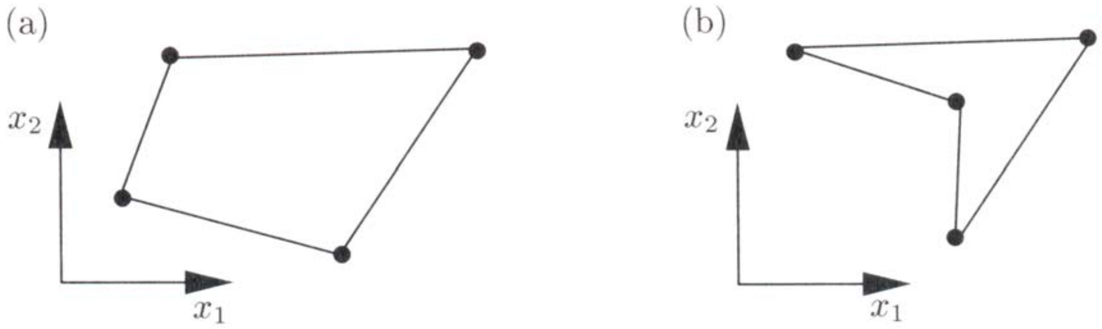
Construction of a two-dimensional nodal expansion basis from the product of two one-dimensional expansions of degree 5 (from Karniadakis & Sherwin 2005).

Viper accepts quadrilateral (four-sided) elements in two dimensions, and hexahedral (six-faced) elements in three dimensions. General curvilinear elements are mapped onto a bi-unit square for implementation of the standard GLL quadrature rules, as illustrated below.



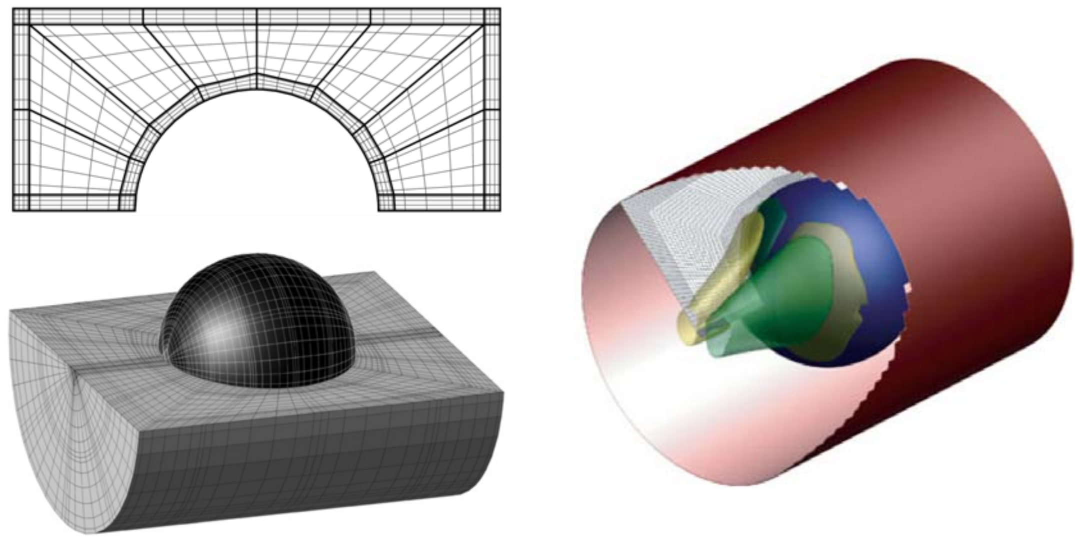
Mapping of a bi-unit square onto a general curvilinear quadrilateral element (from Karniadakis & Sherwin 2005). An analogous mapping onto a bi-unit cube is conducted for three dimensional hexahedral elements.

A result of the mapping procedure is a restriction on the allowable distortion of elements. No element corner is permitted to have an inner angle equal to, or greater than, 180°. Examples of valid and invalid quadrilateral elements are shown below.



Examples of (a) valid and (b) invalid quadrilateral elements (from Karniadakis & Sherwin 2005).

The use of element mapping permits geometries of considerable complexity to be modelled using a spectral-element discretization, and the combination of a high-degree basis and the highly accurate Gauss-Legendre-Lobatto quadrature rules provides excellent spatial convergence properties. Exponential convergence (an increasing rate of error reduction with increasing resolution) is often achieved in practical spectral-element computations (Karniadakis, Israeli & Orszag 1991; Blackburn & Sherwin 2004; Karniadakis & Sherwin 2005; Sheard & Ryan 2007). To illustrate the flexibility of curvilinear quadrilateral and hexahedral elements in discretizing sometimes complicated geometries, meshes are reproduced below from Sheard & Ryan (2007).



Left: Meshes employed for two- (top) and three- (bottom) dimensional computations of the axisymmetric and three-dimensional pressure-driven flows past spheres moving through a tube, respectively (Sheard & Ryan 2007). The upper half of the three-dimensional mesh has been removed to reveal the meshed surface of the sphere. Right: An isosurface plot showing streamwise vorticity in the flow, which demonstrates the existence of non-axisymmetric flow.

Time Integration

The Navier—Stokes equations are integrated forward in time using an operator splitting scheme referred to as a stiffly-stable scheme when first proposed for high-order computation of incompressible fluid flows by Karniadakis, Israeli & Orszag (1991), and later recognised as a class of backwards-multistep schemes by Blackburn & Sherwin (2004).

Operator splitting schemes employ the basic idea that if some equation of the form

$$\frac{\partial \mathbf{u}}{\partial t} = L\mathbf{u}$$

where L is some operator that can be written as a sum of m pieces,

$$L\mathbf{u} = L_1\mathbf{u} + L_2\mathbf{u} + \ldots + L_m\mathbf{u},$$

then the solution that updates the variable **u** from time step n to $n + 1$ can be calculated by summing the contribution of each operator on **u** separately (Press *et al.* 2002).
Backwards-multistep methods are based on backwards differentiation: that is, the time derivative is evaluated at time $n + 1$ (or approximated at time $n + 1$ by a combination of sufficient values at previous times to achieve the desired order of accuracy), and the appropriate-order backwards difference scheme dictates the combination of **u** values at previous times required to find $\mathbf{u}^{n+1}$.
For the incompressible Navier—Stokes equations, Karniadakis, Israeli & Orszag (1991) propose a three-step time splitting scheme

$$\frac{\hat{\mathbf{u}} - \sum_{q=0}^{J-1} \alpha_q \mathbf{u}^{n-q}}{\Delta t} = \sum_{q=0}^{J-1} \beta_q \mathbf{N}\left(\mathbf{u}^{n-q}\right)$$

$$\frac{\hat{\hat{\mathbf{u}}} - \hat{\mathbf{u}}}{\Delta t} = -\nabla P^{n+1}$$

$$\frac{\gamma \mathbf{u}^{n+1} - \hat{\hat{\mathbf{u}}}}{\Delta t} = \nu \nabla^2 \mathbf{u}^{n+1},$$

where $\mathbf{N}(\mathbf{u})$ is the non-linear advection operator, and for third-order accuracy in time ($J = 3$), the required coefficients are:

Coefficient	Value
γ	11/6
α_0	3
α_1	-3/2
α_2	1/3
β_0	3
β_1	-3
β_2	1

Table: Third-order backwards-multistep scheme coefficients.

The first substep involves solving the advection term explicitly. The second substep first requires evaluation of the kinematic pressure, P. We first take the divergence of both sides, and enforce the incompressibility constraint on the intermediate velocity field $\hat{\hat{\mathbf{u}}}$ as

$$\nabla \cdot \left(\frac{\hat{\hat{\mathbf{u}}} - \hat{\mathbf{u}}}{\Delta t} \right) = \nabla \cdot \left(-\nabla P^{n+1} \right)$$

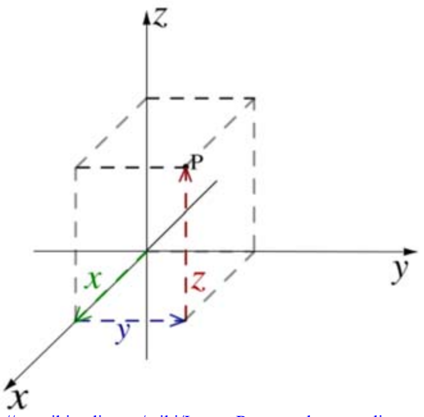
$$\therefore \frac{\nabla \cdot \hat{\hat{\mathbf{u}}} - \nabla \cdot \hat{\mathbf{u}}}{\Delta t} = -\nabla^2 P^{n+1}$$

$$\therefore -\frac{\nabla \cdot \hat{\mathbf{u}}}{\Delta t} = -\nabla^2 P^{n+1}.$$

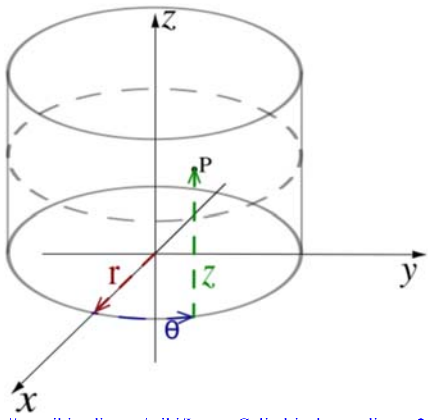
The intermediate velocity field $\hat{\mathbf{u}}$ is calculated during the first substep, so this equation can be solved as a Poisson equation for the kinematic pressure *P*, where appropriate high-order Neumann boundary conditions for pressure are imposed on homogeneous boundaries, and Dirichlet pressure boundary conditions are imposed in the standard fashion. This pressure field can then be used to find the second intermediate velocity field $\hat{\hat{\mathbf{u}}}$.
The third substep involves solving a Helmholtz equation for the final velocity field $\mathbf{u}^{n+1}$, where boundary conditions for the velocity field are imposed.

Coordinate Systems

The preceding equations are presented in vector form for generality. The component forms of these equations vary depending on the coordinate system being employed. Viper has the capability to compute flows in either a Cartesian (*x*, *y*, *z*) or a cylindrical (*z*, *r*, *θ*) coordinate system. These are illustrated below:



http://en.wikipedia.org/wiki/Image:Rectangular_coordinates.svg,



http://en.wikipedia.org/wiki/Image:Cylindrical_coordinates2.svg

In three dimensions, the derivative operators acting on a scalar field are written in Cartesian coordinates as

$$\nabla = \left\langle \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right\rangle, \nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2},$$

and divergence of a vector field is written as

$$\nabla \cdot (\ ) = \frac{\partial}{\partial x}(\ ) + \frac{\partial}{\partial y}(\ ) + \frac{\partial}{\partial z}(\ ).$$

In cylindrical coordinates, the derivative operators are written

$$\nabla = \left\langle \frac{\partial}{\partial z}, \frac{\partial}{\partial r}, \frac{1}{r}\frac{\partial}{\partial \theta} \right\rangle, \nabla^2 = \frac{\partial^2}{\partial z^2} + \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial}{\partial r} \right) + \frac{1}{r^2}\frac{\partial^2}{\partial \theta^2},$$

and the divergence operator is written

$$\nabla \cdot (\ ) = \frac{\partial}{\partial z}(\ ) + \frac{1}{r}\frac{\partial}{\partial r}(r(\ )) + \frac{1}{r}\frac{\partial}{\partial \theta}(\ ).$$

Discrete forms of the Advection Operator

The advection operator for the incompressible Navier—Stokes equations can be expressed in several forms by applying vector identities. These include the convection form ($(\mathbf{u} \cdot \nabla)\mathbf{u}$), the rotation form ($(\nabla \times \mathbf{u})\mathbf{u}$), and the skew-symmetric form ($\frac{1}{2}(\mathbf{u} \cdot \nabla)\mathbf{u} + \frac{1}{2}\nabla(\mathbf{uu})$). These forms are exactly equivalent in a continuous sense, but are not precisely equivalent in a discrete sense. Zang (1991) describes the implications of using each of these forms in numerical computations, and the following table summarises the conservation properties of, and the number of derivative operations required to compute, each of these terms.

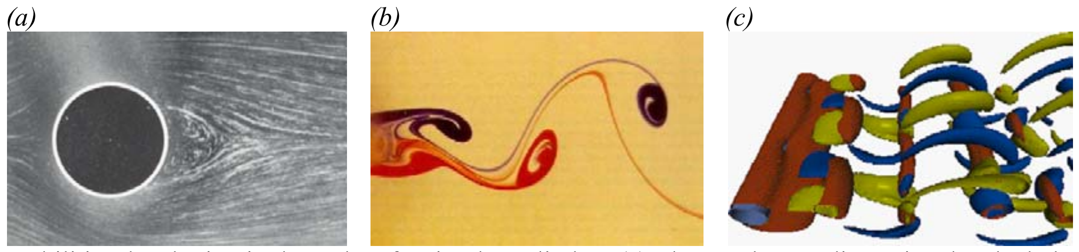
Form of advection operator	**Conserves (in inviscid limit)**	**Number of derivative operations (2D / 3D)**
Convective	Nothing	4 / 9
Rotation	Momentum and kinetic energy	4 / 6
Skew-symmetric	Momentum and kinetic energy	8 / 18

Viper implements all three of these forms of the advection operator (though the rotation form is replaced by the convection form in cylindrical coordinates) with the `advect` command. By default, the skew-symmetric form is employed, which is considered to reduce aliasing errors, though users might consider employing the convection form instead, which was shown by Blackburn & Sherwin (2004) to produce results that converged slightly more rapidly than the skew-symmetric form with increasing spatial resolution.

Stability Analysis

Broadly, stability analysis is the study of the state of systems, and their stability. Many canonical fluid flows develop as a result of instabilities, which often emerge through the solution becoming dependent on an additional dimension. For instance, below a Reynolds number $Re \approx 46$, the flow past a straight circular cylinder is two-dimensional and time-independent. As the Reynolds number is increased beyond this Reynolds number, the flow becomes unstable to temporal disturbances, and the wake alters to the classical von Kármán vortex street, which is again two-dimensional, but is now time dependent (being periodic in time).

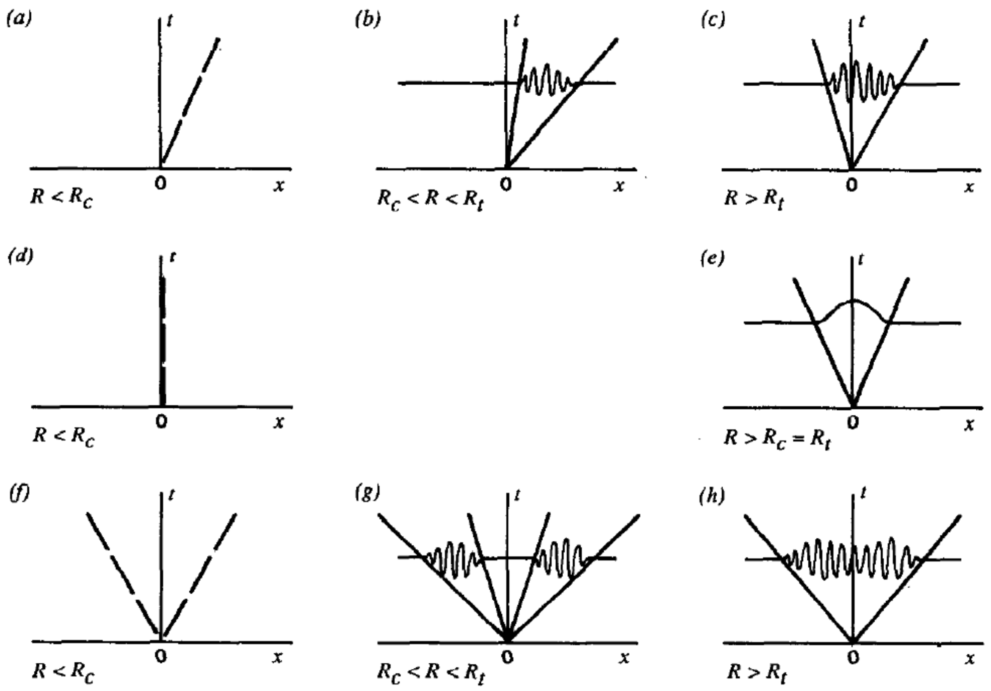
A subsequent transition occurs at $Re \approx 190$, where the two-dimensional Kármán vortex street becomes unstable to three-dimensional sinuous disturbances in the spanwise direction along the cylinder. The image below shows the various wake states through these transitions.



Instabilities developing in the wake of a circular cylinder. (a) The steady two-dimensional wake below $Re = 46$ (Van Dyke 1982), (b) the periodic two-dimensional Kármán vortex street above $Re = 46$, and (c) the three-dimensional "Mode A" wake above $Re \approx 190$ (Thompson, Hourigan & Sheridan 1996).

Absolute and Convective Instabilities

Instabilities can be categorised as being either *local* or *global*, depending on whether the instability develops on a local velocity profile, or the whole flow field, respectively. The terms *absolute* and *convective* are then used to further describe the evolution behaviour of the instability. An absolutely unstable disturbance will spread in all directions and contaminate the entire flow, whereas in a convectively unstable flow the disturbances are washed (convected) away from their point of origin. Given some control parameter R, and considering two critical values, R_c (transition from stable to convectively unstable flow), and R_t (point at which the flow becomes absolutely unstable), the sketches in the subsequent figure outline the various responses of systems, depending on their stability.



Instability responses. (a-c) Single travelling wave: (a) stable, (b) convectively unstable, (c) absolutely unstable. (d-e) Stationary mode: (d) stable, (e) absolutely unstable. (f-h) Conterpropagating travelling waves: (f) stable, (g) convectively unstable, (h) absolutely unstable. Figure reproduced from Huerre & Monkewitz (1990).

Some exciting work is currently underway in the field of numerical convective linear stability analysis (e.g., Blackburn, Barkley & Sherwin 2008), but currently the stability analysis capabilities of Viper are restricted to global instabilities, as described in the next section.

Global Stability Analysis

Numerically, a global stability analysis inspects the evolution of a small disturbance to an underlying base flow. The formulation of this technique begins by decomposing the velocity and pressure fields ($\mathbf{u}, p$) into a two-dimensional base flow ($\overline{\mathbf{U}}, \overline{p}$) and a three-dimensional disturbance ($\mathbf{u}', p'$),

$$\begin{aligned} \mathbf{u} &= \overline{\mathbf{U}} + \mathbf{u}', \\ p &= \overline{p} + p'. \end{aligned}$$

Substituting these into equation (1), cancelling the base flow terms, and neglecting products of the (small) perturbation field yields the *linearised* Navier—Stokes equations

$$\frac{\partial \mathbf{u}'}{\partial t} + \left(\overline{\mathbf{U}} \cdot \nabla\right)\mathbf{u} + \left(\mathbf{u} \cdot \nabla\right)\overline{\mathbf{U}} = -\nabla P' + \nu \nabla^2 \mathbf{u}', \qquad (2a)$$

$$\nabla \cdot \mathbf{u}' = 0. \qquad (2b)$$

Equation (2) differs from equation (1) only in the advection term, and thus an almost identical solution algorithm can be efficiently employed to integrate the disturbance field forward in time.
A further simplification is possible by decomposing the disturbance field into a Fourier series expansion in the spanwise direction,

$$\mathbf{u}'(x, y, z, t) = \int_{-\infty}^{\infty} \hat{\mathbf{u}}(x, y, t) e^{\mathrm{i}\beta z} \mathrm{d}\beta,$$

which then allows us to decouple modes with a different spanwise mode number, β.

$$\mathbf{u}'(x, y, z, t) = \left\langle \begin{matrix} \hat{u}(x, y, t) e^{\mathrm{i}\beta z}, \\ \hat{v}(x, y, t) e^{\mathrm{i}\beta z}, \\ \hat{w}(x, y, t) e^{\mathrm{i}\beta z} \end{matrix} \right\rangle,$$

$$p'(x, y, z, t) = \left\langle \hat{p}(x, y, t) e^{\mathrm{i}\beta z} \right\rangle.$$

The stability behaviour has then been reduced to a two-parameter problem in *Re* and β. An important note in terms of the numerical implementation, is that perturbation fields with different wavelengths only couple with the base flow, so each can be computed independently.
Simplistically, the stability properties for a particular pair of values of *Re* and β is determined by integrating the perturbation field forward in time, and monitoring the growth or decay of the field. Strictly, for *T*-periodic base flows (for steady base

flows, the same technique applies, but the time period T can be arbitrarily selected), the perturbation field evolves over one period subject to an operator **A** as

$$\mathbf{u}'_{n+1} = \mathbf{A}(\mathbf{u}'_n).$$

The eigenvalues of **A** correspond to the Floquet multipliers of the system, $\mu = \exp(\sigma T)$, where σ is the growth rate of the instability. The stability of the base flow ($\overline{\mathbf{U}}, \overline{p}$) is determined by the magnitude of the Floquet multiplier, $|\mu|$. If $|\mu| > 1$, then the flow is unstable to perturbations of the chosen spanwise wavelength at the prescribed Reynolds number, and is stable if $|\mu| < 1$.

A number of methods are available to determine the eigenvalues (and corresponding eigenvectors) of **A**, though due to the size of the systems typically under investigation, **A** is not constructed explicitly. Instead, the base flow and perturbation field are integrated in time, and the perturbation field after successive periods is inspected to determine the eigenspectrum of the system. Barkley & Henderson (1996) and others propose a block-power method based on modified Arnoldi iteration to determine the leading eigenvalue of the system, and Sheard, Thompson & Hourigan (2003) employed a power method to resolve the magnitude of the Floquet multiplier of the fastest-growing mode.

Viper facilitates both Arnoldi and power methods to solve the large-scale eigenvalue problems presented by a global linear stability analysis. An implicitly restarted Arnoldi method (Sorensen 1995; Lehoucq, Sorensen & Yang 1996) is implemented in the ARPACK package, which is called by Viper using the **`arnoldi`** command.

The power method (used in Sheard, Thompson & Hourigan 2003; Sheard & Ryan 2007) isolates the fastest-growing mode, and subsequently computes the magnitude of the Floquet multiplier, by evolving the perturbation field over sufficient periods to allow the modes with smaller growth rates to wash out of the solution. The perturbation field is normalised at each period (permitted due to the linearity of the solution) to avoid the solution diverging as a result of its exponential behaviour. Ultimately, the perturbation field comprises only the fastest-growing mode, and the amplification factor applied to this mode from one period to the next corresponds to the magnitude of the Floquet multiplier, $|\mu|$. The main limitations of the power method are that it cannot resolve the complex components of the leading Floquet multiplier, and it can only find the eigenvalue corresponding to the fastest-growing mode.

The linear Floquet stability analysis technique implemented by Viper is capable of determining the global stability of two-dimensional (or axisymmetric) flows to three-dimensional (non-axisymmetric) linear disturbances that are spanwise (azimuthal)-periodic. This facility is implemented using the **`floq`** command, and calculations employing either an implicitly restarted Arnoldi method, or the power method, are invoked using the **`arnoldi`** or **`stab`** commands, (described in *Chapter 7*), respectively.

Scalar Transport & the Boussinesq Approximation for Buoyancy-Driven Flows

It is sometimes useful to follow the propagation of a scalar quantity through a transient or steady flow field, either for the purposes of flow visualization, or to

simulate the transport of scalar quantities in a flow (such as the transport of oxygen in a bioreactor, for instance).
Viper facilitates two mechanisms for scalar transport: one method introduces a scalar field, which is evolved subject to an advection-diffusion transport equation, and the other method seeds the flow with passive tracer particles, whose positions are updated along with the flow solution.
The advection-diffusion approach is also employed by a facility for computing buoyancy-driven flows by means of a Boussinesq approximation (see command **`bouss`**). For computations employing this facility, the scalar field acts as a normalised temperature field, and the diffusion coefficient represents a thermal diffusion coefficient.

Advection-Diffusion

The transport of a passive scalar field *s* on an evolving flow field **u** is described by

$$\frac{\mathrm{D}\phi}{\mathrm{D}t} = \frac{\partial \phi}{\partial t} + (\mathbf{u} \cdot \nabla)\phi = \upsilon_s \nabla^2 \phi, \qquad (3)$$

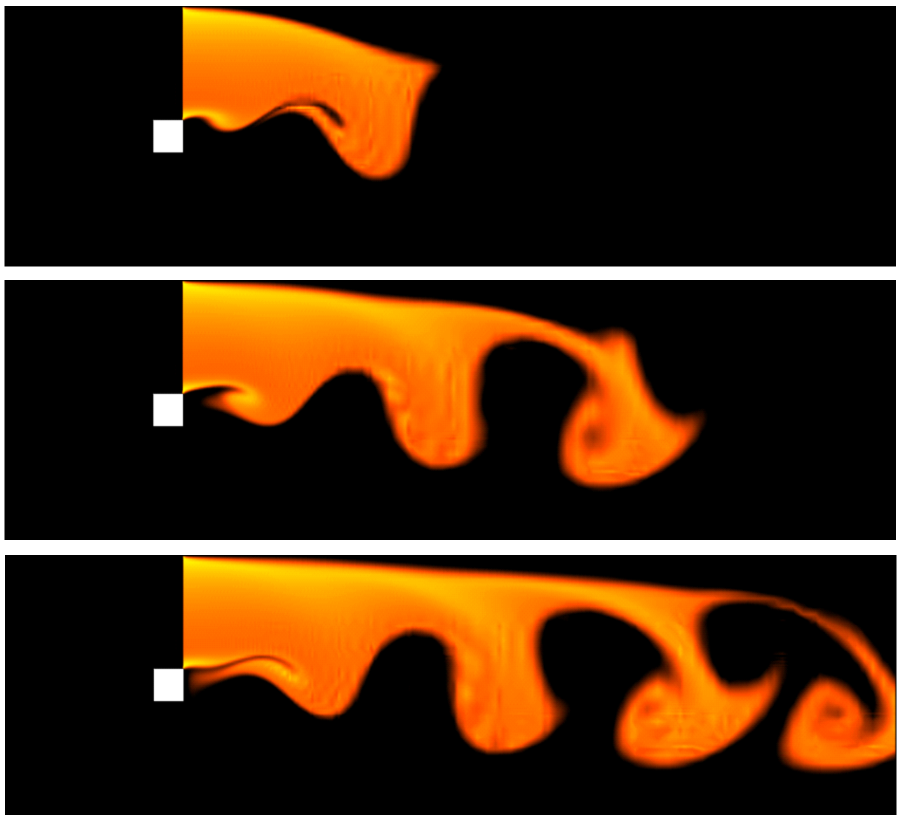
where υ_s is the coefficient of diffusion for the scalar field. This equation can be solved in a number of ways. Physically, this equation describes the movement of the scalar field in time with the flow field, plus diffusion of the scalar field. The numerical solution of this equation can be problematic, as the value of the scalar field at locations in the flow that do not necessarily correspond to grid points can be required. However, in the Auxiliary Semi-Lagrangian technique employed by Viper, no interpolation is required.
The technique first proceeds by integrating an auxiliary advection equation

$$\frac{\mathrm{D}\bar{\phi}}{\mathrm{D}\tau} = \frac{\partial \bar{\phi}}{\partial \tau} + (\mathbf{u} \cdot \nabla)\bar{\phi} = 0,$$
$$t^n \le \tau \le t^{n+1},$$

where

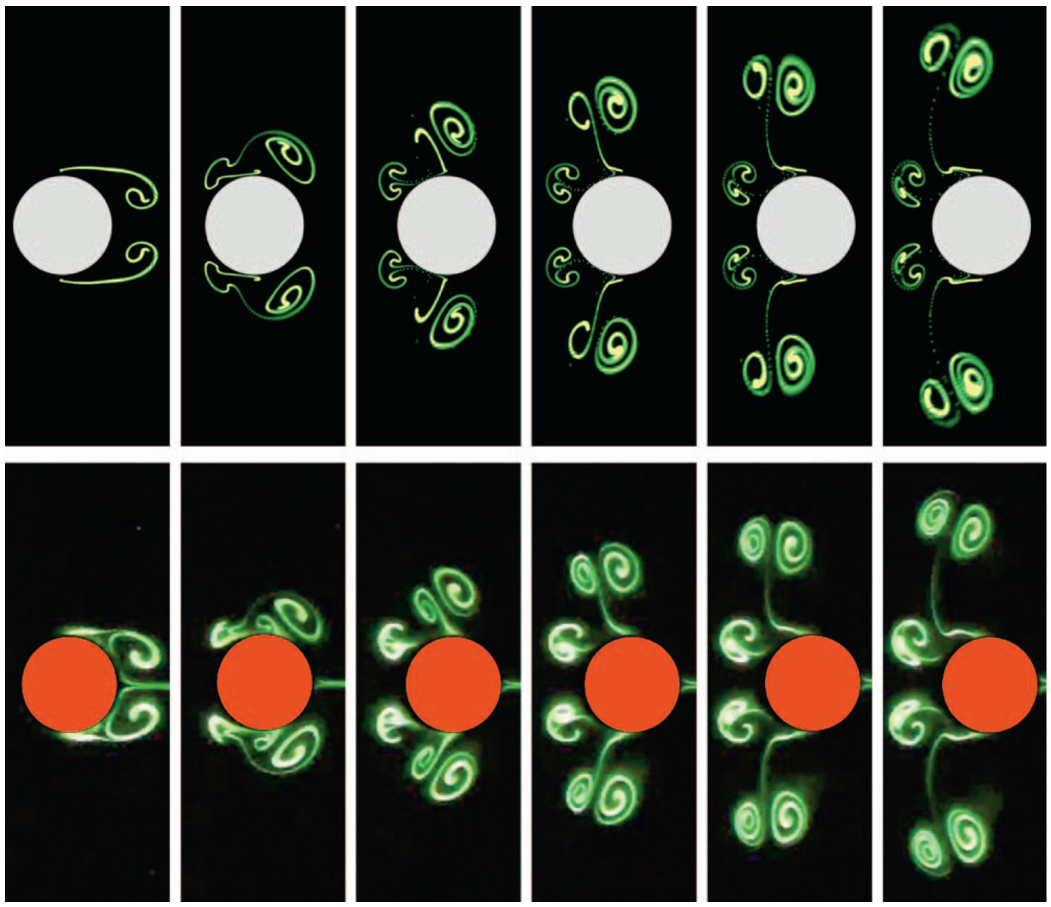
$$\bar{\phi}(x, y, z, t^n) = \phi(x, y, z, t^n),$$

and finally, a diffusion step is performed to complete the solution of equation (3).
This method is best suited for problems involving continuously varying scalar fields present throughout the flow. In Viper, advection-diffusion of a scalar field is initiated by specifying boundary conditions for a scalar field (see **`viper.cfg`** commands **`btag`** and **`gvar_scalar_diff`**), and the command **`scalar`**.
The image sequence below demonstrates the capability of this scalar transport function. Shown are contours of scalar field concentration, and the scalar field is advected on a periodic wake behind a square cylinder in a channel, with a low diffusion specified.


Contours of scalar field concentration, demonstrating fluid mixing behind a square cylinder at *Re* = 90 in a channel with blockage ratio 1/8.

Passive Tracer Particle Tracking

The simulated evolution of passive tracer particles is facilitated by means of a nearly-4th-order Runge—Kutta technique proposed by Coppola, Sherwin & Peiró (2001). This tool is extremely adept at simulating the planar laser-induced fluorescence (PLIF) technique of dye visualization used to great effect by Williamson (1996); Leweke, Thompson & Hourigan (2004). The image below compares experimental dye visualization of an arresting sphere with a numerical simulation produced using Viper, and visualised using the Tecplot package.



A time sequence (from left to right) comparing simulated particle tracking computations (top) and experimental dye visualization (bottom) for an arresting cylinder at *Re* = 500 with a translation distance of two cylinder diameters (Sheard, Leweke, Thompson & Hourigan 2007).

The particle tracking algorithm updates particle positions within each element in parametric space using a 4th-order Runge—Kutta time integration scheme. When a particle crosses an element boundary, a series of first-order sub-steps is employed to step to and across the element interface(s). As the step size is typically small compared to the size of the elements, the technique nearly preserves the 4th-order temporal accuracy of the Runge—Kutta scheme.
Particles can either be injected at a single point or at several points within the flow, or the entire flow field can be seeded with a uniform distribution of particles. Visualization of particles can be performed either by outputting the discrete particle locations in physical space to a text file, or by plotting the particle concentration using the Tecplot package as per the image reproduced here. For Tecplot output, a particle concentration is calculated based on a localised summation of particles subject to a Gaussian mask about each data point. The variance of the Gaussian mask used varies based on the local mesh refinement.

Viper Solvers

Viper provides several solvers for computing a range of fluid flow problems. To compute flow in two-dimensional domains (either in Cartesian or cylindrical coordinate systems, computations are performed on a two-dimensional mesh

comprising quadrilateral (four-sided) spectral elements. The stability of two-dimensional flows to three-dimensional instability modes can be determined by means of the global linear stability analysis capabilities of the code. In these computations, the base flow, and individual Fourier modes of three-dimensional perturbation fields are each computed on a two-dimensional mesh.
Three-dimensional computations may be performed either using hexahedral (six-faced) spectral elements for general geometries, or a Fourier expansion of a two-dimensional domain for geometries which have a symmetry in the out-of-plane direction (either z for Cartesian or θ for cylindrical coordinate system computations).

Running Simulations in Parallel

When running Viper on shared-memory systems with multiple available processors or threads, faster compute times are achievable with some of the solvers, which have been parallelized using the OpenMP application programming interface (http://OpenMP.org/). In the OpenMP model of parallelization, one thread is designated the "Master" thread, and this thread controls the computation. Any serial segments of the programme are computed on this thread. When a parallel region of the code is reached, the Master thread spawns additional threads on available processors, and distributes the computational work among these threads as dictated by the OpenMP commands included in the code.
Speedup is a measure of the benefit available from parallel computing, and is defined as a ratio of the time taken to run a simulation over a single processor to the time taken to run the same simulation over multiple processors. Optimal speedup would equal the number of available processors, though unfortunately there are practical limitations to how much speedup is available in real computations. There is an increasing memory and processor overhead involved with the establishment of parallel threads when more threads are available, so to gain a good benefit from parallel computing, the amount of work to be done in parallel must be significant to overcome the performance degradation due to overhead.
To gain the most benefit from parallel computations, care is required to ensure that an appropriate number of parallel threads are used. For instance, if the number of parallel tasks in a parallel region is 5, and the computation is run over two threads, then one thread must perform three of the tasks, and the other will compute only two. Thus one of the threads will sit idle waiting for the other to finish the additional computation. In terms of speedup, this means that even if the computation was ideal (no overhead), the maximum available speedup would be 5/3 = 1.667, not 2 as may have been hoped. Avoiding idle threads is the only technique available for end-users to maximise their speedup and efficiency in parallel computations using Viper. The sections below provide advice on how to best select the number of threads for their computations.

Parallel base flow simulations

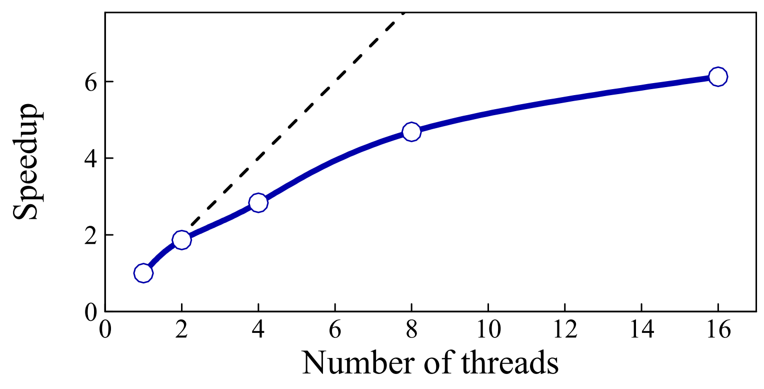
Currently only a limited speedup is achievable when running two-dimensional quadrilateral and three-dimensional hexahedral simulations. This is due to the substantial time spent computing global operations such as coupled boundary system matrix solutions. For these simulations, a small speedup is achieved when running over two processors as compared to one, though computations are found to actually slow down beyond that number.

Parallel linear stability analysis computations

The global linear stability analysis algorithm operates by computing the base flow and each of the perturbation fields separately. Each of the perturbation fields is completely decoupled from the others, depending only on the base flow velocity fields from previous time steps. Therefore, a significant performance gain can be achieved by running these simulations in parallel, with the computation of the required fields being shared between available processors. The total number of time integration solutions required at each time step is $N_p + 1$, where N_p is the number of active perturbation fields (as we also need to evolve the base flow[1]).

The maximum number of threads that should be used when computing linear stability analysis computations is $N_p + 1$. To avoid idle threads, users should compute with either this number of threads, or whole factors of this number. For example, if a linear stability analysis computation was analysing 7 perturbation fields, then the total number of fields being computed is 8, and computations should employ 8, 4, 2, or 1 thread. Less efficient speedup would be achieved for computations using 7, 6, 5, or 3 threads.

The figure below plots the speedup achieved for a linear stability analysis test case. The test case featured a spectral-element mesh featuring 343 elements of polynomial degree 8, and 15 perturbation fields were active.



A plot of the speedup for a linear stability analysis test case in which 15 perturbation fields were being evolved in conjunction with the base flow. The dotted line indicates ideal (linear) speedup.

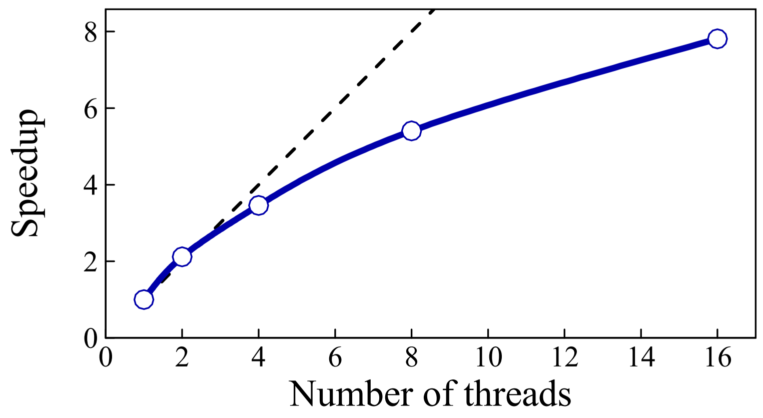
Parallel spectral-element/Fourier computations

The spectral-element/Fourier algorithm computes a three-dimensional solution where the variation in the flow in the out-of-plane direction is represented by a Fourier series. In the formulation employed in Viper, the Fourier modes are only coupled during the advection step: the pressure and diffusion steps can be computed in a decoupled fashion. Therefore a substantial speedup is available as each Fourier mode can be computed on a separate thread.

Spectral-element/Fourier computations are initialised using the **`fourier`** command, and the number of Fourier planes is specified at this time. The number of Fourier planes corresponds to the number of sample points in the Discrete Fourier Transform. Any number of planes greater than 2 is permitted. Viper uses the Discrete Fourier Transform code supplied with the Intel Math Kernel Library, so users are not

[1] Note that if users are performing stability analysis on a frozen base flow (using the **`freeze`** command), then there are effectively only N_p fields to be computed to complete each time step.

restricted to numbers of planes in powers of two. Due to the conjugate symmetry property of discrete Fourier transforms of real data (no imaginary component), the negative frequency modes need not be explicitly computed. With a number of planes N_f, the number of Fourier modes being computed is $N_f/2 + 1$, where integer division is used (round down to the nearest whole number). For example, if a user wishes to compute a spectral-element/Fourier computation with 31 planes, this corresponds to 16 modes, and therefore simulations would best be performed on 16, 8, 4, 2, or 1 thread. As with linear stability analysis calculations, poorer performance will result if the number of threads was not a factor of 16.
The figure below plots the speedup achieved for a spectral-element/Fourier test case. The test case was computed in cylindrical coordinates, and contained 30 Fourier planes. The mesh had 385 elements of polynomial degree 5.



A plot of the speedup for a spectral-element/Fourier three-dimensional test case with 30 Fourier planes. The dotted line indicates ideal (linear) speedup.

Important OpenMP environment variables

Viper is compiled using the Intel Fortran compiler version 10.1, which implements extensions to the default set of OpenMP environment variables.
By default, OpenMP implementations specify a small stack for each thread, which is often exceeded during simulations using Viper. The OpenMP per-thread stack size is restricted in size even if an unlimited stack size is specified on the system. The defaults are 2 Mb on 32-bit systems, and 4 Mb on 64-bit systems. The environment variable **`KMP_STACKSIZE`** can be established to specify a non-default size of the stack. For example, under Linux the user could type

```
\> setenv KMP_STACKSIZE 100m
```

to establish a per-thread stack size of 100 megabytes. Suffixes **`b`** (bytes), **`k`** (kilobytes), **`m`** (megabytes), **`g`** (gigabytes), or **`t`** (terabytes) are used to specify the units.

Most modern desktop machines have either multi-threaded or multi-core processors. Running large simulations on these platforms can produce unexpected crashes due to a limited per-thread stack size. This can be overcome by creating the environment variable **`KMP_STACKSIZE`**, and setting its value as required. To do this, go to **`Control Panel`** → **`System`** → **`System Properties`**. From the **`Advanced`** tab, click **`Environment Variables`**. Then under **`System variables`**, click **`New`**, and enter the details as required. You may also use this procedure to explicitly set the **`OMP_NUM_THREADS`** environment variable if parallel computation is performing strangely.

On shared-memory systems with Non-Uniform Memory Architecture (NUMA) configurations, such as the Altix Cluster (AC) at the Australian Partnership for Advanced Computing (APAC) National Facility, a performance boost may be achieved if threads are assigned to specific physical threads, cores, or processors on the system. This occurs because there is a significant difference in the time taken to access memory located near different processors on such systems. The environment variable **`KMP_AFFINITY`** can be used to bind threads to physical processing units. Experimentation has found that a small speedup is achieved if users set the **`KMP_AFFINITY`** environment variable in the following fashion:

```
\> setenv KMP_AFFINITY granularity=fine,compact
```

On Linux systems this can either be set in the users' **`.login`** file, or for specific jobs by including this line in their queue script file. The following example shows a queue script file used to launch a Viper job using 16 threads on the AC machine at APAC:

```
#!/bin/csh
#PBS -P h66
#PBS -q normal
#PBS -l walltime=2:00:00,vmem=2024MB,ncpus=16:16
#PBS -wd
setenv OMP_NUM_THREADS 16
setenv KMP_AFFINITY granularity=fine,compact
cd /short/hxx/user/job1/
../viper < macro.txt > /short/hxx/user/job1/output.txt
```

Chapter 3: Pre-Processing

To conduct a CFD computation, some pre-processing is usually required. For simulations performed using Viper, the pre-processing phase entails the construction of meshes using a mesh generation package, and if necessary, converting these meshes into a format accepted by Viper.

Accepted Mesh Formats

Viper currently accepts conforming meshes comprising quadrilateral (4-sided) or hexahedral (6-faced) elements. Quadrilateral meshes are employed for two-dimensional, axisymmetric, or three-dimensional spectral-element/Fourier computations. Hexahedral meshes are employed for three-dimensional computations in general geometries. Conforming meshes require that adjacent elements meet edge-to-edge or face-to-face.

The format for mesh files used by Viper is a text-based format which first lists the vertex coordinates, and then describes the elements, their connectivity, and the boundary numbers of each edge/face.

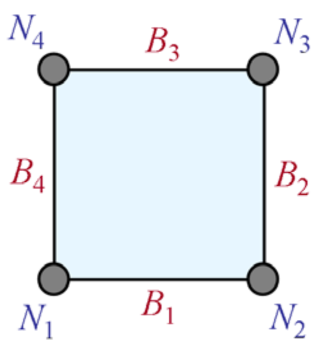
The following outlines the required mesh format:

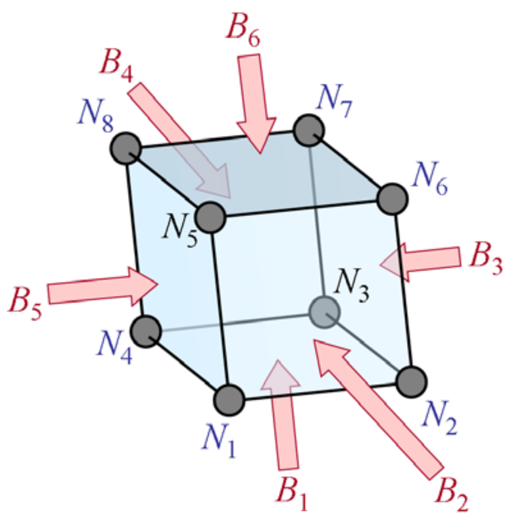
```
Nvert
x1, y1, [z1,] 1
x2, y2, [z2,] 2
:
:
xNvert, yNvert, [zNvert,] Nvert
Nelem
1, N1, N2, N3, N4, [N5, N6, N7, N8,] B1, B2, B3, B4,
    [B5, B6,] 1
2, N1,...,N4/N8 (2D/3D), B1,...,B4/B6 (2D/3D), 1
:
:
Nelem, <Vertex numbers of element corners>, <Boundary
    numbers of element edges>, Region
```

The following definitions apply:

• **Nvert**	Number of mesh vertices
• **Nelem**	Number of mesh elements
• **Region**	Fluid region (currently not used)
• **xn**, **yn**, **zn**	Spatial (x, y, z) coordinates of mesh vertices
• **N1-N8**	Ordered numbering of vertices at element corners
• **B1-B6**	Ordered numbering of boundaries on element edges/faces

The numbering convention employed when constructing elements from mesh vertices is outlined below for quadrilateral (left) and hexahedral (right) elements. The corresponding numbering of boundary edges/faces is also shown.





Converting from Gambit

The Gambit mesh generation package can be used to generate meshes for use in Viper. Conversion utilities available from the FLAIR Intranet (http://www.flair.monash.edu.au/intranet/) convert Gambit mesh files exported in the FIDAP format (`.FDNEUT` files) to the Viper text-based mesh format.
From Gambit, the conversion process is as follows:

1. Create a mesh comprising either quadrilateral (4-sided 2D) or hexahedral (6-faced 3D brick) elements.
2. Set the **Solver** type to **FIDAP**
3. Define boundary conditions, using different names for each uniquely numbered boundary.
4. Save mesh: Select FILE > EXPORT > MESH to save mesh with `.FDNEUT` extension.
5. Exit Gambit.
6. Rename file to default `mesh_in.FDNEUT` for conversion.
7. Invoke the appropriate conversion tool (2D or 3D).
8. A new text file is created containing mesh information readable by Viper.

Chapter 4: Configuring Simulations

Prior to running a simulation, a configuration file must be created to provide Viper with the necessary information to establish and solve the flow correctly. This information must be contained in a text file named **`viper.cfg`**, which should be located in the directory in which Viper is invoked.
The **`viper.cfg`** file contains the following information:

- Location of the mesh file,
- Values for simulation parameters (e.g., **`dt`**, **`RKV`**, **`N`**),
- User-defined functions,
- Initial and boundary conditions.

The commands used to supply these details to Viper are described in the following section.

Commands recognised in the `viper.cfg` file

btag

Syntax: **`btag <tag_num> <var> <boundary_type_ID> [<param1> <param2> <param3>]`**

Function: **Defines the condition to be imposed on a particular boundary.**

Description:

The **`btag`** command is used to link boundary tag numbers in the mesh file **`<tag_num>`** with a type of boundary (defined by an ID number **`<boundary_type_ID>`** recognised by Viper. Currently, Viper accepts the following boundary ID numbers:

1. Constant Dirichlet boundary (values of components of flow variables are given by **`<params>`**).
2. Static user-defined Dirichlet boundary (components are expressed as mathematical expressions that are functions of spatial coordinates **`x`**, **`y`**, **`z`**, and the reciprocal kinematic viscosity, **`RKV`**).
3. Transient user-defined Dirichlet boundary (components are again expressed as mathematical expressions, which here can also be functions of time, **`t`**).
4. Periodic boundaries (*x*-direction only). This boundary requires boundary edges/faces to be identical on a pair of periodic boundaries.
5. Symmetry boundary (no velocity normal to the boundary, and zero shear stress along the boundary – this condition is inexactly imposed at the conclusion of each time step).

Viper permits the separate prescription of velocity and pressure boundary conditions on a boundary through the **`<var>`** string, which can be set to "**`vel`**" or "**`p`**"(case insensitive), for velocity and pressure, respectively.
In addition, if **`<var>`** takes the value "**`s`**", then a scalar field will be established, which will then be computed using an Auxiliary Semi-Lagrangian Advection-Diffusion approach (for further details on this method see Maday, Patera & Rønquist, *J. Sci. Comp.*, 1990). Users should also then specify the coefficient of scalar diffusion using **`gvar_scalar_diff`**.

The following are examples of the use of `btag`:
e.g. 1:

```
\> btag 5 vel 3 ‘x*cos(t)’ ‘2.0’ ‘3.0’
```

Specifies that boundary number 5 (in the mesh file) will be prescribed a transient user-defined Dirichlet velocity condition with velocity components $u = x\cos(t)$, $v = 2.0$, and $w = 3.0$.

e.g. 2:

```
\> btag 4 p 1 0.5
```

Specifies that boundary number 4 will be prescribed a fixed Dirichlet pressure condition with $p = 0.5$ on the boundary.

gvar_curve

Syntax: `gvar_curve <bndry>`
Function: **Specifies a boundary number on which to apply automated boundary curvature.**
Description:
The domain boundary number `<bndry>` corresponds to the boundary number as defined in the `btag` statements in the `viper.cfg` file. Continuous blended curves comprising circular arcs are constructed along edges corresponding to boundary number `<bndry>`. Continuous curvature is not enforced for adjacent edges on a single element to avoid illegal element mappings. In 3D, an edge-curvature-preserving interpolation is applied to generate the curved surface on each boundary face.

gvar_dt

Syntax: `gvar_dt <value>`
Function: **Sets the time step Δt.**
Description:
The time step Δt is set to `<value>`, where `<value>` must be greater than 0.0, otherwise the default value $\Delta t = 0.005$ is used instead.

gvar_init_field

Syntax: `gvar_init_field <u_fn> <v_fn> <w_fn> <p_fn>`
Function: **Sets an initial velocity/pressure field for a simulation.**
Description:
Viper solves the time-dependent Navier—Stokes equations forward in time from some initial condition, subject to imposed boundary conditions. If no initial velocity field is set, Viper begins computing from a zero interior velocity field. This facility allows user-specified functions for the velocity fields to be specified, which can, in some cases, make simulations more stable or more efficient, by permitting an improved “first guess” of the velocity field to be used.

If the user subsequently calls `load` to load a velocity field from a saved file, then that velocity field is used to begin the computation, rather than what is specified by `gvar_init_field`.
Functions `<u_fn>`, `<v_fn>`, `<w_fn>` and `<p_fn>` are input for each of the velocity components *u*, *v* and *w*, and the kinematic static pressure *p*.
These functions accept variables time `t`, spatial coordinates `x`, `y`, and `z`, and the reciprocal kinematic viscosity `RKV`. In two dimensions, `z` is assumed to be zero.

gvar_init_scalar_field

Syntax: `gvar_init_scalar_field <s_fn>`
Function: **Sets an initial scalar field for a simulation**
Description:
If the user subsequently calls `load` to load a velocity field from a saved file, then that velocity field is used to begin the computation, rather than what is specified by `gvar_init_field`.
A function `<s_fn>` is input for the initial scalar field distribution at the beginning of the computation.
This functions accept variables time `t`, spatial coordinates `x`, `y`, and `z`, and the reciprocal kinematic viscosity `RKV`. In two dimensions, `z` is assumed to be zero.

gvar_kink

Syntax: `gvar_kink <elem> <vertex>`
Function: **Specifies a node at which to allow a curvature discontinuity on a boundary in 2D.**
Description
A *kink*, or a discontinuity in curvature, is permitted at the mesh node corresponding to element `<elem>` and vertex `<vertex>` in 2D. `<elem>` must be a positive integer, which is set to the largest element number if `<elem>` is greater than the number of elements, and `<vertex>` is a positive integer between 1 and 4. This feature is used to avoid attempts by the automated curvature algorithm in Viper to create unrealistic curvature, such as around a deliberately sharp corner in a mesh. An example of this is the sharp trailing edge of an aerofoil.

gvar_monitor (deleted)

Description:
This command is no longer implemented. Use command-line function `monitor` instead.

gvar_movref

Syntax: **`gvar_movref <u> [<v> <w>]`**

Function: **Specifies time-varying functions for the velocity of a moving reference frame.**

Description:

The user inputs time-varying functions for the velocity of a moving reference frame. Functions can include variables time **`t`**, reciprocal kinematic viscosity **`RKV`**, and any user-specified functions.

At least one function (for the *u*-velocity) must be supplied. Functions for *v* and *w* components are optional.

If this command is included in the **`viper.cfg`** file, this facility adjusts the velocity fields at each time step to accommodate a time-varying moving reference frame. The sign convention is such that if the user wishes for the velocities within the computational domain to be adjusted to match a time-varying boundary condition, both should be specified with the same sign.

gvar_n

Syntax: **`gvar_n <value>`**

Function: **Sets the element polynomial degree (*p*-resolution).**

Description:

The element polynomial degree is set to an integer **`<value>`**, where **`<value>`** must be equal to, or greater than, 2. The maximum allowable polynomial degree is restricted only by system resources. Increasing this value improves spatial resolution of computations on a mesh, though users should note that this incurs costs due to larger and slower calculations, and less stable calculations, requiring a smaller time step.

gvar_nnvisc

Syntax: **`gvar_nnvisc <function>`**

Function: **Sets a function for a non-Newtonian viscosity.**

Description:

A mathematical expression **`<function>`** is input, which is a function of the shear rate **`SR`**, and the spatial coordinates **`x`**, **`y`**, and **`z`**. In two-dimensional computations, **`z`** is assumed to be zero.

gvar_re (obseleted)

Description:

As of 10 August 2007 this command has been renamed to **`gvar_rkv`**, reflecting the name change of the reciprocal kinematic viscosity parameter from **`Re`** to **`RKV`**.

gvar_rkv

Syntax: `gvar_rkv <value>`
Function: **Sets the reciprocal kinematic viscosity RKV.**
Description:
The reciprocal kinematic viscosity parameter **RKV** is set to **`<value>`**. If the simulation imposes a unit reference velocity, and employs a mesh with a unit reference length, then the Reynolds number of the simulation is equal to the value of the **RKV** parameter.

gvar_scalar_diff

Syntax: `gvar_scalar_diff <coeff>`
Function: **Sets the diffusion coefficient for transport of a scalar field.**
Description:
The parameter <coeff> specifies the coefficient of diffusion for the advective-diffusive transport of a passive scalar field on a fluid flow. The scalar field *S* is integrated using an auxiliary semi-Lagrangian advection-diffusion algorithm (e.g., see Maday, Patera & Rønquist, *J. Sci. Comp.*, **5**(4), 263-292, 1990).

gvar_usrvar

Syntax: `gvar_usrvar <func_name> <function>`
Function: **Creates a user-defined mathematical function.**
Description:
A function named **`<func_name>`** is created, and is assigned to a mathematical expression **`<function>`**, which is which is a function of time **`t`**, spatial coordinates **`x`**, **`y`**, **`z`**, the reciprocal kinematic viscosity **RKV**, plus any previously created user-defined functions.
A character string is required for each of **`<func_name>`** and **`<function>`** parameters, which must be enclosed in single quotes (`'`) if they include parentheses. The new function **`<func_name>`** must be given a unique name, which cannot conflict with any of the implicit variables recognised by Viper. The expression **`<function>`** is a string specifying the function that is evaluated whenever other functions featuring **`<func_name>`** as a variable are evaluated.

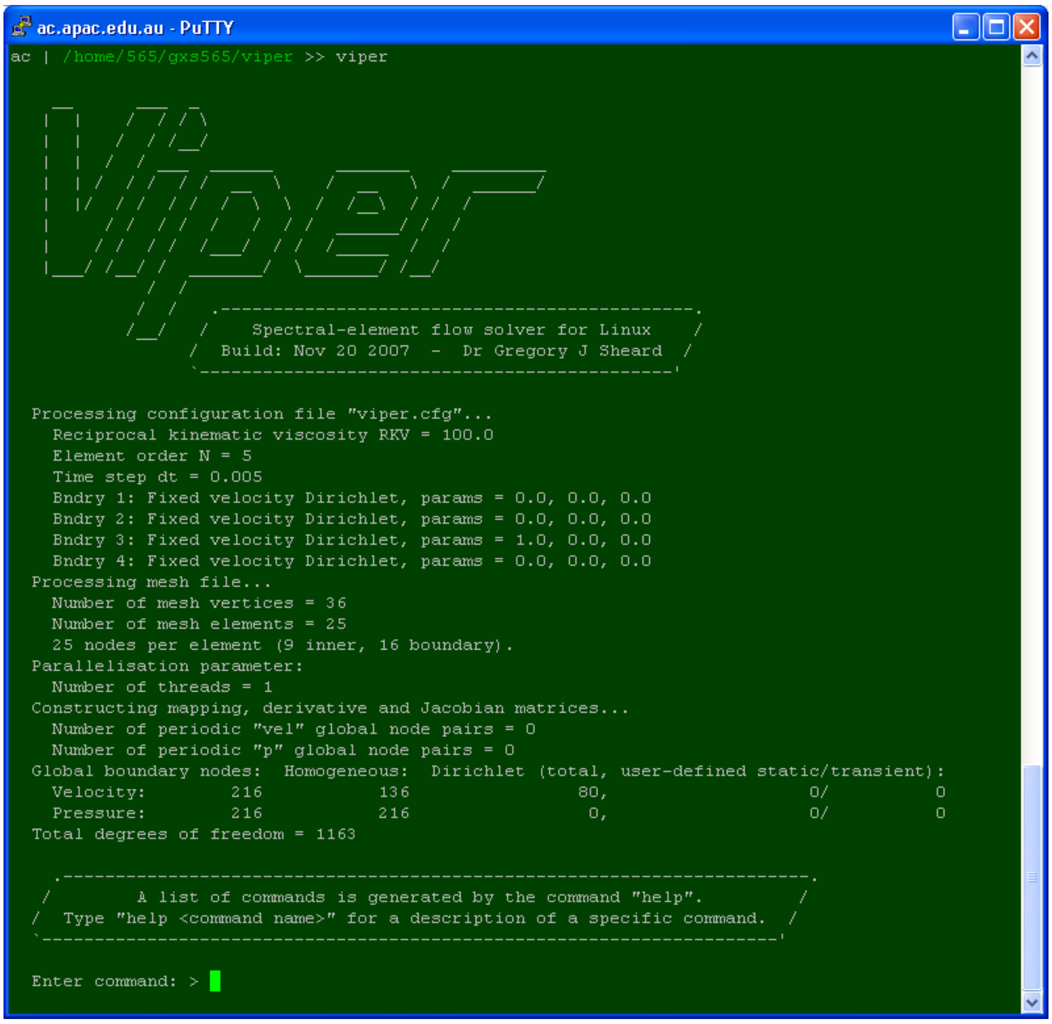
Chapter 5: Running Simulations

Once a suitable configuration file is established to define the problem to be solved, Viper is relatively easy to use. Instructions can either be input interactively by the user, or supplied to the code in a macro file. While Viper executables exist for use under a Windows operating system as well as Linux platforms, it is a command-line application: there is no Graphical User Interface (GUI).

When invoked, Viper automatically seeks the configuration file **`viper.cfg`**, and if not found, it prompts the user for a file containing appropriate configuration instructions. Once a suitable file is located, Viper then proceeds to process the contents of the configuration file, during which the mesh data is input, boundary and initial conditions are established, and various mapping and indexing arrays are generated.

These processes are accompanied by output printed to the screen, which should be checked carefully if the process fails, or the subsequent simulation produces undesirable or unexpected results.

Finally, the user is instructed on how to activate the help utility, which can be used to find out what commands are available, and give detailed instructions on their usage. An example of screen output upon launching Viper is shown below.



```
ac | /home/565/gxs565/viper >> viper

    __      ___ _
   |  |    / / / \
   |  |   / / /__/
   |  |  / /   ___  ______     _______   _________
   |  | / / / / / /  __  \   /  ___  \  /  _______/
   |  |/ / / / / /  /  \  \ /  /___\  / /  /
   |    / / / / /  /   / / /  _______/ /  /
   |   / / / / /  /___/ / /  /______  /  /
   |__/ /__/ /  _______/  \_________/ /__/
           / /
          / /   .---------------------------------------------.
         /__/   /    Spectral-element flow solver for Linux    /
               /  Build: Nov 20 2007  -  Dr Gregory J Sheard  /
               `---------------------------------------------'

  Processing configuration file "viper.cfg"...
    Reciprocal kinematic viscosity RKV = 100.0
    Element order N = 5
    Time step dt = 0.005
    Bndry 1: Fixed velocity Dirichlet, params = 0.0, 0.0, 0.0
    Bndry 2: Fixed velocity Dirichlet, params = 0.0, 0.0, 0.0
    Bndry 3: Fixed velocity Dirichlet, params = 1.0, 0.0, 0.0
    Bndry 4: Fixed velocity Dirichlet, params = 0.0, 0.0, 0.0
  Processing mesh file...
    Number of mesh vertices = 36
    Number of mesh elements = 25
    25 nodes per element (9 inner, 16 boundary).
  Parallelisation parameter:
    Number of threads = 1
  Constructing mapping, derivative and Jacobian matrices...
    Number of periodic "vel" global node pairs = 0
    Number of periodic "p" global node pairs = 0
  Global boundary nodes:  Homogeneous:  Dirichlet (total, user-defined static/transient):
    Velocity:        216          136                 80,                    0/         0
    Pressure:        216          216                  0,                    0/         0
  Total degrees of freedom = 1163

    .----------------------------------------------------------------------.
   /        A list of commands is generated by the command "help".        /
  /  Type "help <command name>" for a description of a specific command.  /
  `----------------------------------------------------------------------'

  Enter command: >
```

An example of the screen output after Viper is launched: the configuration file **`viper.cfg`** has successfully been located and processed, and Viper awaits input from the user.

This chapter describes a number the tasks and features that can be employed when using Viper.

Saving and Loading flow field data using restart files

Sometimes a simulation has not finished before a user needs to end their session at a terminal, and sometimes hardware faults or divergence within a computation can cause a simulation to fail, potentially losing hours of valuable work. Viper facilitates a buffer against these potential calamities by allowing the user to save the computed flow fields at instants in time to restart files. This is implemented with the **`save`** and **load** commands.

The **`save`** command can be used at the end of, or many times during, a simulation, to store the velocity fields for a possible restart of the computation in a later session. At the beginning of a subsequent Viper session, the **`load`** command can be used to read in the saved velocity fields, allowing the simulation to proceed from where it was saved.

Restart files are also useful in allowing the user to initiate a computation from a saved solution, but run it at a different parameter (such as the Reynolds number).

Using Macros and Loops

The macro facility provides an alternative to manually (interactively) entering commands during a Viper session. This is especially useful if the user wishes to run jobs remotely (such as on high-performance computing facilities), or if there is a lengthy list of complex commands the user may wish to execute several times.

Macros are simply text files containing a list of commands recognisable by Viper. Each command must appear on its own line, and spaces and tabs are treated the same. The macro file can have any name or extension the user wishes.

Input control can be passed to a macro file either from within Viper, or when launching Viper. Within Viper, the **`macro`** command is used to open and execute commands within a supplied macro file. From the Linux shell / Windows command prompt, the user can execute Viper with instruction to take input from the macro file, rather than the keyboard, by using the left angled bracket feature of both operating systems, i.e.:

```
\> viper < macro
```

launches the Viper executable **`viper`**, and input is piped from the file named **`macro`**.

Macro files can be nested – it is possible to include the command **`macro`** within a macro file.

For repetitive tasks, Viper has the ability to execute a sequence of commands in a loop. This is facilitated using the **`loop`** command, which permits the user to specify their required number of iterations. Additional loops can be nested within parent loops, and macro files can also be called from within loops. Therefore, powerful and complicated sets of instructions can be executed with very few user-input keystrokes.

For example, a macro file could be established, named **`macro1.txt`**, containing the following:

```
axi
init
step 1000
save -f save.dat
tecp -f tecplot.plt
stop
```

The user could then invoke Viper, and use the macro command to read from the macro file, by typing

```
macro macro1.txt
```

Macros can be combined with loops for some considerable flexibility. Imagine two macro files, **`macro2.txt`** and **`macro3.txt`**, containing:

`macro2.txt` **commands:**	**`macro3.txt`** **commands:**

`init`
`loop 3`
`macro macro3.txt`
`endl`
`stop`
	`step 100`
	`save -s -f save.dat`
	`tecp -s -f tecplot.plt`
	`flowrate`
	`forces 2`
	`forces_bndry2.dat`

From within Viper, if the command

```
macro macro2.txt
```

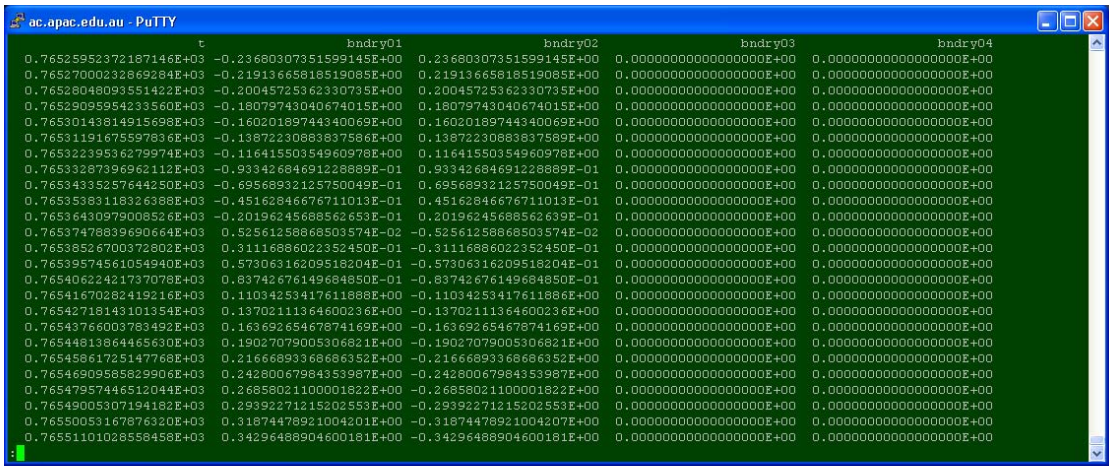
is called, the macros and **`loop`** command make this equivalent to typing the following list of commands:

```
init
loop 3
step 100
save -s -f save.dat
tecp -s -f tecplot.plt
flowrate
forces 2 forces_bndry2.dat
step 100
save -s -f save.dat
tecp -s -f tecplot.plt
flowrate
forces 2 forces_bndry2.dat
step 100
save -s -f save.dat
tecp -s -f tecplot.plt
flowrate
forces 2 forces_bndry2.dat endl
stop
```

Chapter 6: Post-Processing

Once a simulation has been completed, the output usually requires some form of post-processing to be converted into useful results. Viper outputs data in two primary formats: ASCII files and Tecplot binary files.

Text-based (ASCII) files typically contain time history data of various quantities, with each line in the file containing data at time increments through the computation. For instance, the command **`flowrate`** is used to output the flow rate through each boundary on a mesh, and the example below shows the content of such an output file for a mesh with four boundaries, two of which (boundaries 3 and 4) are impermeable (no flow through them):



```
                      t               bndry01                bndry02                bndry03                bndry04
 0.76525952372187146E+03 -0.23680307351599145E+00  0.23680307351599145E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76527000232869284E+03 -0.21913665818519085E+00  0.21913665818519085E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76528048093551422E+03 -0.20045725362330735E+00  0.20045725362330735E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76529095954233560E+03 -0.18079743040674015E+00  0.18079743040674015E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76530143814915698E+03 -0.16020189744340069E+00  0.16020189744340069E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76531191675597836E+03 -0.13872230883837586E+00  0.13872230883837589E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76532239536279974E+03 -0.11641550354960978E+00  0.11641550354960978E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76533287396962112E+03 -0.93342684691228889E-01  0.93342684691228889E-01  0.00000000000000000E+00  0.00000000000000000E+00
 0.76534335257644250E+03 -0.69568932125750049E-01  0.69568932125750049E-01  0.00000000000000000E+00  0.00000000000000000E+00
 0.76535383118326388E+03 -0.45162846676711013E-01  0.45162846676711013E-01  0.00000000000000000E+00  0.00000000000000000E+00
 0.76536430979008526E+03 -0.20196245688562653E-01  0.20196245688562639E-01  0.00000000000000000E+00  0.00000000000000000E+00
 0.76537478839690664E+03  0.52561258868503574E-02 -0.52561258868503574E-02  0.00000000000000000E+00  0.00000000000000000E+00
 0.76538526700372802E+03  0.31116886022352450E-01 -0.31116886022352450E-01  0.00000000000000000E+00  0.00000000000000000E+00
 0.76539574561054940E+03  0.57306316209518204E-01 -0.57306316209518204E-01  0.00000000000000000E+00  0.00000000000000000E+00
 0.76540622421737078E+03  0.83742676149684850E-01 -0.83742676149684850E-01  0.00000000000000000E+00  0.00000000000000000E+00
 0.76541670282419216E+03  0.11034253417611888E+00 -0.11034253417611886E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76542718143101354E+03  0.13702111364600236E+00 -0.13702111364600236E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76543766003783492E+03  0.16369265467874169E+00 -0.16369265467874169E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76544813864465630E+03  0.19027079005306821E+00 -0.19027079005306821E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76545861725147768E+03  0.21666893368686352E+00 -0.21666893368686352E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76546909585829906E+03  0.24280067984353987E+00 -0.24280067984353987E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76547957446512044E+03  0.26858021100001822E+00 -0.26858021100001822E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76549005307194182E+03  0.29392271215202553E+00 -0.29392271215202553E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76550053167876320E+03  0.31874478921004201E+00 -0.31874478921004207E+00  0.00000000000000000E+00  0.00000000000000000E+00
 0.76551101028558458E+03  0.34296488904600181E+00 -0.34296488904600181E+00  0.00000000000000000E+00  0.00000000000000000E+00
:
```

The contents of the ASCII output file created after a number of calls to **`flowrate`**.

Notice that results are stored in these files at a very high precision (approximately 17 significant figures) to ensure that all the precision of the *double-precision* arithmetic of the code is preserved in the output.

Commands which can be used to create ASCII data files include (see their entries in the subsequent Command List for more information):

```
flowrate
forces
getminmax
int
l2
monitor
```

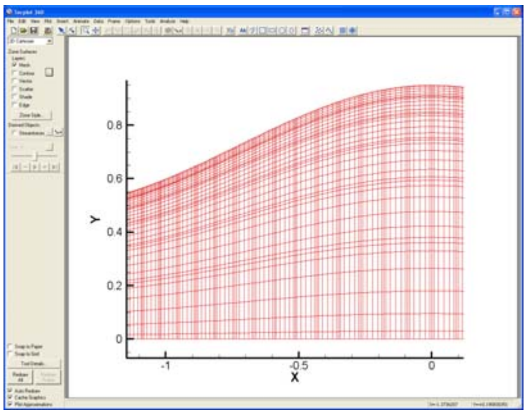
Furthermore, ASCII file output is generated when invoking global linear stability analysis (using **`floq`** and **`arnoldi`** or **`stab`**), as well as simulated particle tracking (using **`track`** commands).

For visualization of the computed flow fields, Viper generates binary data files suitable for plotting using the Tecplot package (see `www.tecplot.com` for more information). These files should carry the default extension **`.plt`**, though files with extension **`.dat`** can also be opened with Tecplot. To generate a Tecplot binary file, use the command **`tecp`**, but note that specialist Tecplot plotting files are also generated when computing a global linear stability analysis using either **`tec_floq`** or **`arnoldi`**.

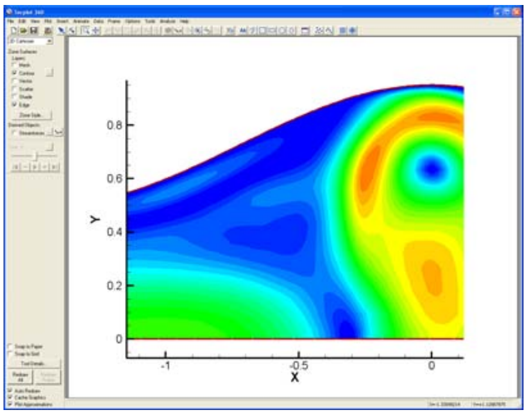
Visualizing Flow Fields with Tecplot

Flow fields visualised using Tecplot contain the spatial coordinate and connectivity data defining the mesh, plus data fields corresponding to various quantities. Users have some control over which variables are stored – see the **`tecp`** command description for more information.
The images below show examples of visualization of data in Tecplot. Shown is a portion of a larger two-dimensional computational domain, and plotted are the mesh, flooded contours of velocity magnitude (one of the numerous quantities available), velocity vectors, and streamlines.

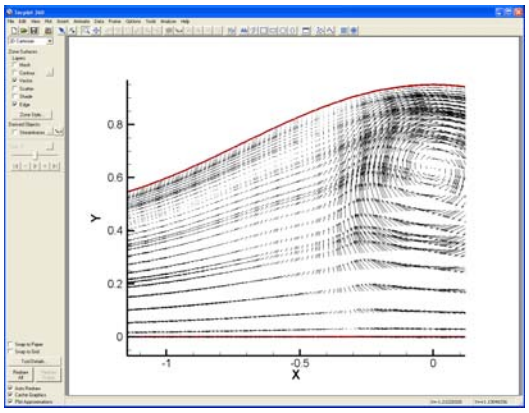
(a)



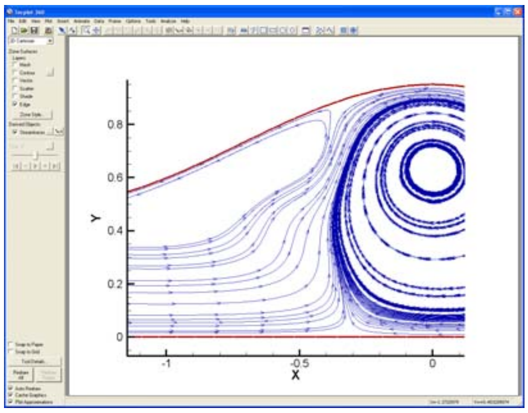
(b)



(c)



(d)



Visualization of a portion of the computational domain of a two-dimensional simulation. (a) The mesh, (b) flooded contours of instantaneous velocity magnitude, (c) velocity vectors, and (d) velocity streamlines.

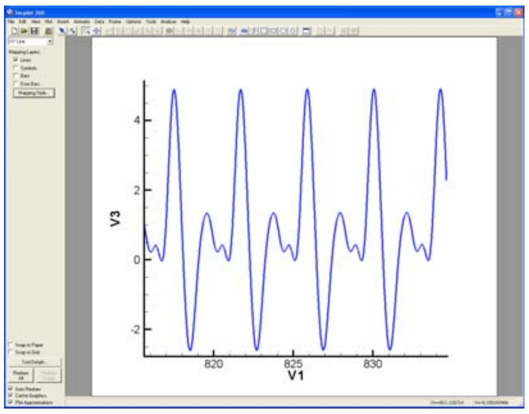
Users are encouraged to experiment with Tecplot, as there are many possibilities for plotting available, and with some practice, first-class figures can be generated.

Plotting ASCII Data Files

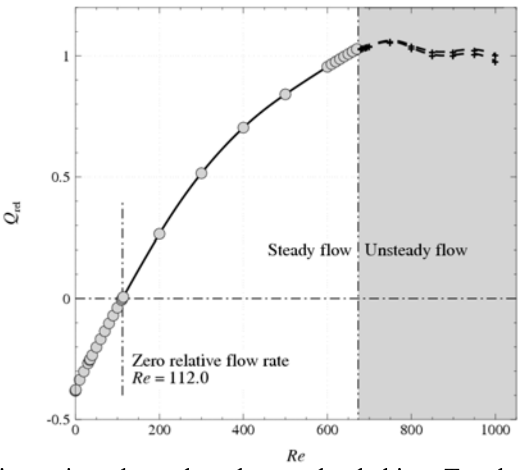
The Tecplot package can also be used for plotting the data contained in the ASCII data files, as by default, Tecplot can read the columnar data format presented in these files.
From a Windows desktop, users can right-click on an ASCII data file (with the **`.dat`** extension), and can select *Open With* → *Tecplot*.

Alternatively (and on Linux systems), these files can be loaded from within Tecplot in the standard fashion.

(a)



(b)



Graphing data with Tecplot: (a) A screenshot showing a time-dependent data set loaded into Tecplot with default plotting options. (b) A plot from Sheard & Ryan (2007) generated using Tecplot.

In the above figure, both the default appearance of plotted data in Tecplot, and an example of a published plot, are shown to illustrate that a substantial flexibility in appearance and style can be obtained using features of the plotting software.

Chapter 7: Command List

Viper recognises a number of commands which are used to initialise, run, and obtain output from, a simulation. A description of each command similar to those given here can be obtained while running Viper by invoking the Help facility, i.e.:

```
\> help <command_name>
```

where `<command_name>` is the name of the command for which a description is required. A list of available commands can be generated simply by typing:

```
\> help
```

Commands in the following list are sorted alphabetically. Each entry contains the following information:

Syntax:	The command, plus any `[optional]` `<parameters>` or `-options` that can be supplied.
Function:	A brief description of the action performed by the command.
Description:	A more detailed description of the functionality of the command.

Advect

Syntax: `advect <form>`

Function: **Select the form of the advection operator.**

Description:

The advection term of the Navier—Stokes equations can be written in a number of forms which are equivalent in a continuous sense, though not in a discrete sense. Viper implements the *rotational*, *skew-symmetric* and *convective* forms of the advection term. These are invoked by typing the following:

`advect convective`

The *convective* form conserves neither momentum nor kinetic energy in the inviscid limit, and requires 4 or 9 derivative operations for two- or three-dimensional computations, respectively.

`advect rotational`

The *rotational* form conserves both momentum and kinetic energy in the inviscid limit, and only requires 4 or 6 derivative operations in two- or three-dimensional computations, but has been shown to cause significant aliasing errors. If this option is selected during axisymmetric computations, the *convective* form is used instead.

`advect skew`

The skew-symmetric form is the default, and it conserves both momentum and kinetic energy in the inviscid limit. It requires 8 or 18 derivative operations in two- or three-dimensional computations, but despite this cost, it has been shown to be the preferred form in spectral, Galerkin and collocation methods for minimising aliasing errors.

The *advect* command can also be used to switch the advection term off (or back on again) during computations of the base flow (does not apply to perturbation fields during Floquet stability analysis:

`advect on`

Turn on computation of the advection term.

`advect off`

Turn off computation of the advection term.

Note that switching off the advection term reduces the equations being solved to the creeping flow equations.

See also: **`diff`**, **`pres`**.

Arnoldi

Syntax: **`arnoldi <Neigs> <Nits> [<file_prefix>]`**

Function: **Perform an Arnoldi iteration of global linear three-dimensional stability analysis.**

Description:

If Floquet stability analysis is being performed (call **`floq`** prior to **`init`**), this command performs an iteration of the Implicitly Restarted Arnoldi Method, which is used to compute several of the leading complex eigenvalues (Floquet multipliers) and the corresponding eigenvectors (perturbation velocity fields for the Floquet modes) of the linear operator **A**, which describes the effect of integrating the perturbation field forward in time by one period, *T*.

- The **`<Neigs>`** parameter is an integer specifying the number of leading eigenvalues that are to be computed (typically only a handful are desired).
- The **`<Nits>`** parameter is an integer specifying the number of Arnoldi vectors that are generated at each iteration. The relation **`<Nits>`** ≥ 2 + **`<Neigs>`** must be satisfied, but otherwise **`<Nits>`** should be kept reasonably small to reduce the storage cost of the method.
- The optional string **`<file_prefix>`** is added to the beginning of the output files created upon convergence of the eigenvalues. This is essential to avoid files accidentally being overwritten if multiple jobs are being run in the same directory.

Presently, this facility can only be employed on a single spanwise/azimuthal wavelength. This approach is far more powerful than the stability analysis capability provided by the **`stab`** command, which only returns the magnitude of the leading Floquet multiplier. The **`arnoldi`** command returns the complex components of several of the leading modes.

Once the **`arnoldi`** routine converges on the requested number of eigenvalues, the eigenvector fields are saved to Viper restart files **`<file_prefix>save_floq_eigXXXX.dat`**, and to Tecplot binary files **`<file_prefix>tecp_floq_eigXXXX.plt`**. The converged Floquet multipliers are printed to screen (or **`STDOUT`**), and to a file named **`<file_prefix>floq_mult_eigs.dat`**.

On the first occasion that this command is called in a Viper session, an Arnoldi restart file **`<file_prefix>saved_arnoldi_eigs.dat`** is searched for. If it exists, the state of a previously saved Arnoldi iteration is loaded, and the computation continues from that position.

At the conclusion of every **`arnoldi`** call, the current state of the Implicitly Restarted Arnoldi Iteration is saved to a file named **`<file_prefix>saved_arnoldi_eigs.dat`**. This feature allows the user to

perform an Arnoldi stability analysis over several Viper sessions. Users should note that if a file of the same name exists in the working directory, it will be overwritten without prompting the user.
See also: **`floq`**, **`stab`**.

Autocorrf

Syntax: **`autocorrf [-f <filename> -x <x> <y>]`**
Function: **Return the autocorrelation of each SE/Fourier velocity component at a point.**
Description:
This command outputs the time (t), the supplied spatial coordinates, and the autocorrelation of each velocity component along the span at a physical point on the mesh.
Notes:

- Unlike the **`monitor`** command, **`autocorrf`** interpolates the flow quantities to the requested location, rather than just output the values at the nearest mesh node.
- Furthermore, the points are calculated and output to file at the time that **`autocorrf`** is called.
- **`autocorrf`** can only be called after **`init`**.

Given a discrete Fourier transform of the spanwise variation of a velocity component $\mathcal{F}u$, the autocorrelation s calculated first by taking the product of $\mathcal{F}u$ and its complex conjugate, and then by finding the inverse discrete Fourier transform of this product.
The following options are available:

`-f <filename>`
Used to specify a filename **`<filename>`** (including extension) to save the flow values to. If omitted, the default filename is **`samplef.dat`**.

`-x <x> <y>`
Used to specify the (x, y) coordinates of a point in the computational domain at which the Fourier coefficients are to be determined.

See also: **`energyf`**, **`samplef`**.

Axi

Syntax: **`axi`**
Function: **Toggles between cylindrical and Cartesian coordinate systems (2D only).**
Description:
Two-dimensional computations may be carried out in either a Cartesian (the default; x-y-z) or a cylindrical (z-r-θ) coordinate system. This command is used to toggle between the two modes.
If cylindrical coordinates are switched on, then the computations are performed in an axisymmetric sense, where $y = 0.0$ is taken to be the symmetry axis $r = 0.0$. Therefore, the user should ensure that no mesh vertices include a negative y-coordinate, as this will produce unpredictable, incorrect, and non-physical results.

Notes:

- **`axi`** has no effect on three-dimensional computations, which are currently restricted to Cartesian coordinates only,
- **`axi`** can be toggled at any time, though the computation will need to be re-initialized prior to further time stepping. Care should be taken to ensure that post-processing commands (e.g., **`forces`**, **`flowrate`**, **`tecp`**, etc.) are called with the appropriate **`axi`** setting.

See also: **`wvel`**.

Bouss

Syntax: **`bouss <density_gradient> <gravity>`**

Function: **Implement a Boussinesq approximation for density-driven convection.**

Description:

Notes:

- *This command requires that the scalar advection-diffusion field is active, as this field represents the temperature field.*
- *The scalar diffusion coefficient must be set appropriate to the diffusion properties of whatever medium is being evolved (e.g. thermal diffusion coefficient for temperature, etc.).*

This command implements density-driven convection by means of a Boussinesq approximation.
The Boussinesq approximation is valid for small density variations, as under these conditions the density difference enters only through the gravity term. For simplicity, the acceleration due to gravity is taken to act in the positive x-direction, which corresponds to the positive axial direction in the cylindrical formulation of the code. The x-direction momentum equation is modified by adding the gravity term:

$$\rho' g_x,$$

where ρ' is the density normalised by a reference density ρ_0, and is expressed

$$\rho' g_x = 1 + \left.\frac{\mathrm{d}\rho'}{\mathrm{d}T'}\right|_{T=T_0} (S-1).$$

The <density_gradient> parameter $\left(\left.\frac{\mathrm{d}\rho'}{\mathrm{d}T'}\right|_{T=T_0} \right)$ is supplied to the **`bouss`** command, and represents the dimensionless gradient of the density-temperature profile of the working fluid, evaluated at the reference temperature, and normalised by the reference density and temperature. The temperature of the working fluid is represented by the scalar field S, and is taken to be normalised by a reference temperature ($S = T/T_0$). While this implementation is designed to implement a temperature-based density-driven convection, in fact any density-driven convection can be incorporated in this fashion (e.g. density variation due to solute concentration, etc.) provided that the basis can be transported as a scalar field.

Chref (obsolete)

Syntax: `chref <u-chng> <v-chng> [<w-chng>]`
Function: **Adjust the reference frame in which the fluid is moving.**
Description:
This command changes the Cartesian velocity components in the interior of the computational domain to allow for an alteration in the reference frame of motion. For example, if a body is initially at rest, then impulsively begins moving to the right (positive *x*-direction), an appropriate call would be

```
\> chref -1.0 0.0 0.0
```

which adds -1.0 to all the interior *u*-velocities. The user must independently adjust any boundaries whose velocity also alters (e.g., the inlet and transverse boundaries for the impulsively started cylinder case).
Note that this command has been superseded by the more powerful `gvar_movref` facility implemented through the `viper.cfg` configuration file.

Diff

Syntax: `diff`
Function: **Toggle diffusion substep on/off during time integration.**
Description:
Time integration is carried out by solving each of the advection, pressure and viscous diffusion terms consecutively. This function is used to switch off computation of the diffusion term. The default setting of this feature is ON. This facility is primarily provided as a debugging tool.
Note that switching off the diffusion term alters the equations being solved by Viper.
See also: `pres`, `advect`.

Energyf

Syntax: `energyf [-f <filename>]`
Function: **Compute norms of energy in each Fourier mode in an SE/Fourier 3D simulation.**
Description:
An energy norm is computed for each Fourier mode of a three-dimensional spectral-element/Fourier computation. For each Fourier mode (k), the energy norm is given by the integral

$$\oint_{\Omega} \left\| \hat{\mathbf{u}}_k \right\|^2 \mathrm{d}\Omega,$$

where the k^{th} Fourier mode coefficients of the velocity field are given by $\hat{\mathbf{u}}_k$, and Ω is the computational domain in the spectral-element plane (either x-y or z-r).
The computed energy norms are written to a text file with default filename **`energyf.dat`**, or if provided, the optional filename **`<filename>`**.
See also: **`autocorrf`**, **`samplef`**.

Exit

Syntax: **`exit`**
Function: **Exits Viper.**
Description:
Viper terminates immediately, and any unsaved work will be lost. This command performs the same action as **`stop`** and **`quit`**.
See also: **`quit`**, **`stop`**.

Floq

Syntax: **`floq <m1> [<m2> <m3> ... <mNfloq_modes>]`**
Function: **Invoke Floquet linear stability analysis and specify mode numbers (2D only).**
Description:
Floquet analysis is a type of linear stability analysis which can be used to determine the stability of a two-dimensional base flow to three-dimensional disturbances which are spatially periodic with a specific spanwise/azimuthal wavelength. This technique requires either a periodic or time-independent two-dimensional base flow.
This command must be called prior to a call to **`init`**, as it is used to specify a number of spanwise (two-dimensional Cartesian) or azimuthal (axisymmetric) mode numbers for linear stability analysis. This command cannot be invoked in three-dimensional computations. The number of mode numbers which can be analysed simultaneously ($N_{\text{Floq\_modes}}$) is limited only by available computational resources, which increases approximately with ($N_{\text{Floq\_modes}} + 1$). The spanwise/azimuthal wavelength

$$\lambda = \frac{2\pi}{m},$$

m is the spanwise/azimuthal mode number.
During a Floquet analysis computation, the periodic base flow is evolved in the standard fashion, and in addition, a perturbation field associated with each mode number is also evolved. At periodic intervals, either the command **`arnoldi`** or **`stab`** is used to calculate and output the Floquet multiplier(s) associated with each azimuthal/spanwise mode.
See also: **`arnoldi`**, **`stab`**.

Flowrate

Syntax: **`flowrate [<filename>]`**
Function: **Outputs the flow rate through each boundary.**
Description:
The volume flow rates *out* of each boundary are calculated and output to a file named **`<filename>`**. If **`<filename>`** is not specified, the data is written to a default file named **`flowrate.dat`**. If the file exists, the time *t*, and each boundary flow rate is appended as a row of space-separated numbers to the end of the existing text file. Otherwise, a new file is created.

Forces

Syntax: **`forces <boundary> [<filename>]`**
Function: **Calculate global forces imparted on a specific boundary.**
Description:
Calculates the global forces imparted on the boundary numbered **`<boundary>`** in the **`viper.cfg`** file. These forces are output to a file **`<filename>`** (default is **`forces.dat`**), which is appended if it already exists. If no boundary number is given, no calculations or output are performed.

Fourier

Syntax: **`fourier [-f <filter_dist> -n <Nplanes> -mode <mode_number> -span <span>]`**
Function: **Configure the solver for a three-dimensional computation.**
Description:
Three-dimensional computations can be performed on a two-dimensional mesh provided that the geometry is homogeneous in the out-of-plane direction (*z* in Cartesian, *θ* in cylindrical coordinates). This is achieved by expanding the flow variables in the out-of-plane direction using a Fourier expansion.
This command supplies the necessary parameters to initialise a spectral-element/Fourier computation. The following options are available:

`-f <filter_dist>`

In cylindrical coordinates (use **`axi`**), the vanishinly small grid spacing near the axis can lead to an amplified stability constraint on the time step.
By default, a ramp filter is applied to the advection substep from 100% at $r = 0$ to 0% at $r = 10\%$ of the maximum radial dimension in the grid. If a positive value of **`<filter_dist>`** is supplied, a ramp filter is applied extending to *r* = **`<filter_dist>`**. This facility is useful if a filter is helpful, but the default distance is either too small or too large for the model being computed.

`-mode <mode>`

The positive floating point value supplied as <mode> specifies the out-of-plane wavenumber describing the extent of the computational domain in the out-of-plane direction. The mode number relates to the span by **<span>** = 2*π*/**<mode>**. If no span or mode number is supplied, the computation defaults to a span of 2*π*, corresponding to an out-of-plane mode number $m = 1$. If both are given, the computation will employ the most recently given value.

`-n <Nplanes>`

A positive integer is supplied to specify the number of Fourier planes employed in the computations. After Fourier transformation, this corresponds to **<Nplanes>**/2 Fourier modes in the out-of-plane direction. A default of 4 planes is used.
Note: For best efficiency from parallel simulations, computations should be run on **<Nplanes>**/2+1 threads, or factors thereof. For example, if a user wishes to compute with 30 Fourier planes, this corresponds to 15 complex Fourier modes, resulting in 16 separate fields to be computed (including the fundamental mode). Thus simulations would best be run on 16, 8, 4, 2, or 1 thread (or processor).

`-span <span>`

The positive floating point value supplied as **`<span>`** specifies the out-of-plane extent of the computational domain. Users can either specify an out-of-plane span using this option, or they can use the **`-mode`** option to specify this parameter as an out-of-plane wavenumber (useful for computations in cylindrical coordinates). The span is taken as being in length units for Cartesian computations, and in radian for computations using cylindrical coordinates.

If a **`load`** command is called prior to this routine, the two-dimensional solution input during **`load`** is mapped to the three-dimensional velocity field. Users must call **`init`** after a call to **`fourier`**, to prepare for time integration.
See also: **`rand`**.

Freeze

Syntax: **`freeze`**
Function: **Toggles a freeze on time integration of the base flow.**
Description:
The default condition is OFF, which provides for normal time integration of the base flow velocity field when the **`step`** command is used.
Sometimes, though, it is useful to freeze the base flow, while continuing as normal to carry out time integration of perturbation fields in Floquet analysis, or simulated particle tracking. This could either be as a result of the base flow being time-independent (in which case **`freeze`** could be used to save time by not evolving the steady-state flow), or in specific cases where the user may wish to interrogate a frozen snapshot of a normally time-varying flow field.
See also: **`track`**, **`floq`**, **`rotate`**

Getminmax

Syntax: **`getminmax [-f <filename> -p <function> -c <cutoff> -x <level> <tol> -e]`**
Function: **Find location and values of minima and maxima of a user-specified scalar field.**
Description:
A user-specified function <function> is input (using the same mathematical functions available during configuration), and the positions (x, y, z) of maxima and minima, and values of the scalar function at those locations are returned.

Available variables are:

`t`	Current time,
`x`, **`y`**, **`z`**	Spatial coordinates,
`u`, **`v`**, **`w`**, **`p`**	Velocity components (u, v, w) and kinematic static pressure (p),
`RKV`	Reciprocal kinematic viscosity ($1/v$),

`dudx`, **`dudy`**, **`dudz`**, **`dvdx`**, **`dvdy`**, **`dvdz`**, **`dwdx`**, **`dwdy`**, **`dwdz`** (spatial velocity gradients $\frac{\mathrm{d}u}{\mathrm{d}x},\frac{\mathrm{d}u}{\mathrm{d}y},\frac{\mathrm{d}u}{\mathrm{d}z},\frac{\mathrm{d}v}{\mathrm{d}x},\frac{\mathrm{d}v}{\mathrm{d}y},\frac{\mathrm{d}v}{\mathrm{d}z},\frac{\mathrm{d}w}{\mathrm{d}x},\frac{\mathrm{d}w}{\mathrm{d}y},\frac{\mathrm{d}w}{\mathrm{d}z}$),

and any user-specified variables defined during configuration.

Local minima and maxima are located where the gradient vector of the scalar field is zero.

The values of all variables are determined at the current time, and the evaluated locations are output to either the default **`minmax.dat`**, or the optional user-specified **`<filename>`**.

The following options are available:

`-f <filename>`

Used to specify a filename **`<filename>`** (including extension) to save the minima/maxima data to. If omitted, the default filename is **`minmax.dat`**.

`-p <function>`

A user-specified function **`<function`**> is provided to the routine. If omitted, the default is vorticity in the *x*-*y* plane ($\frac{\mathrm{d}v}{\mathrm{d}x}-\frac{\mathrm{d}u}{\mathrm{d}y}$): "**`dvdx-dudy`**".

`-c <cutoff>`

A cutoff value for the square of the magnitude of curvature at turning points. Turning points below this **`cutoff`** threshold are ignored. The square of the magnitude of the curvature for each located turning point is output to screen, so users will be able to *tune* their minima/maxima identification to isolate only those they wish to find on a simulation-specific basis. The default value is $|\text{curvature}|^2 = 0.0$.

`-x <level> <tol>`

A threshold for turning points whose scalar value lies within a certain tolerance **`<tol>`** of a specified value **`<level>`** can be employed with this option. Any turning point with a maximum/minimum scalar field value lying between **`<level>`** - **`<tol>`** and **`<level>`** + **`<tol>`** will be ignored. The defaults are **`<level>`** = 0.0 and **`<tol>`** = 0.0, (i.e., no turning points are ignored).

`-e`

If specified, the magnitude of the rate of strain is computed at the locations found, and this is also output.

Help

Syntax: **`help [<command name>]`**

Function: **Gives assistance to user.**

Description:

If no **`<command name>`** input, a list of available commands is given.

If **`<command name>`** is provided, a detailed description of the command follows.

Immerse

Syntax: **`immerse [-static]`**
Function: **Toggles immersed-boundary computation on/off.**
Description:
Immersed-boundary calculations are performed in two-dimensional simulations when this feature is activated. An external code is currently required to facilitate this feature, and if it is not operational, this code will hang during time stepping, as Viper enters a perpetual loop which only stops when the external code has completed its calculations.
If the **`-static`** option is specified, the immersed boundary is taken to be fixed in time, and thus the information from the forcing input file is processed only once.

Init

Syntax: **`init`**
Function: **Initialize job for time integration.**
Description:
This routine builds all the necessary matrices for time-integration of the flow solution. If **`init`** is called multiple times in a Viper session, all matrices and storage are re-created afresh.

Int

Syntax: **`int <function> [<filename>]`**
Function: **Integrates a user-specified function over the computational domain.**
Description:
A user-specified function **`<function>`** is input (using the same mathematical functions available during configuration), and the value of this function is integrated over the computational domain. Additional available variables are:

`t`	Current time,
`x`, **`y`**, **`z`**	Spatial coordinates,
`u`, **`v`**, **`w`**, **`p`**	Velocity components (*u*, *v*, *w*) and kinematic static pressure (*p*),
`RKV`	Reciprocal kinematic viscosity (1/*v*),

`dudx`, **`dudy`**, **`dudz`**, **`dvdx`**, **`dvdy`**, **`dvdz`**, **`dwdx`**, **`dwdy`**, **`dwdz`** (spatial velocity gradients $\frac{\mathrm{d}u}{\mathrm{d}x},\frac{\mathrm{d}u}{\mathrm{d}y},\frac{\mathrm{d}u}{\mathrm{d}z},\frac{\mathrm{d}v}{\mathrm{d}x},\frac{\mathrm{d}v}{\mathrm{d}y},\frac{\mathrm{d}v}{\mathrm{d}z},\frac{\mathrm{d}w}{\mathrm{d}x},\frac{\mathrm{d}w}{\mathrm{d}y},\frac{\mathrm{d}w}{\mathrm{d}z}$),

and any user-specified variables defined during configuration.
The values of all variables are determined at the current time, and the evaluated integral is output to either the default text file **`integral.dat`**, or the optional user-specified **`<filename>`**.
See also: **`l2`**.

L2

Syntax: `L2 [<filename>]`
Function: **Compute the L_2 norm (integral of velocity magnitude throughout domain).**
Description:
The L_2 norm is defined as

$$L_2 \equiv \oint_\Omega |\mathbf{u}| \mathrm{d}\Omega,$$

where $|\mathbf{u}|$ is the magnitude of the velocity vector, and Ω is the computational domain. This quantity is calculated at the current time, t, and the result is written to either the default text file **`l2norm.dat`**, or if provided, to the optional filename **`<filename>`**.
See also: **`int`**.

Load

Syntax: `load [-f <filename> -k <floq_mode> -m]`
Function: **Load flow field vectors from file.**
Description:
Loads flow field vectors, as well as computation parameters **`t`**, **`dt`**, **`RKV`**, and the number of elements and element polynomial degree from a user-specified file. This command loads flow field data from files created with the command SAVE, and is used to begin a computation from a previously computed solution.
Update 4/11/2006: This command can now read files containing flow fields at the three previous time steps, while also being capable of reading the old current-time saved fields. The new files avoid the annoying perturbation that was added to flows upon re-start.
The following options are available:

`-k <floq_mode>`
Used to specify an integer perturbation field number (i.e., 1, 2, ... , $N_{\mathrm{Floq\_modes}}$, when Floquet analysis is active) to load a saved flow field into. The default is **`<floq_mode>`** = 0, corresponding to the base flow.

`-f <filename>`
Used to specify a filename **`<filename>`** (including extension) to load the flow fields from. If the **`-f`** option is not specified, the default filename **`ff_in.dat`** is used.

`-m`
Specifies that you wish to load spatial coordinates from the file also (this feature is only required if you wish to load data onto a different macro-element mesh.

See also: **`save`**.

Loop

Syntax: `loop <num_iterations>`
Function: **Executes a list of commands <num_iterations> times.**
Description:
Following a call to `loop <num_iterations>`, the user inputs a list of commands to be executed within a loop. The command list is terminated by entering **`endl`** (for "end loop"). Multiple loops can be nested within one another. The looping begins after the final **`endl`** command is supplied.
The commands are stored in a temporary "scratch" file (visible on Linux systems, invisible on Windows systems), which may not be deleted if Viper is terminated while looped commands are being executed. These files are typically named **`fortXXXXX`**, and are safe to delete if Viper is not running in that directory.
See also: **`macro`**.

Macro

Syntax: `macro <filename>`
Function: **Read commands from a file.**
Description:
Specifies a file from which commands are to be input from. The file **`<filename>`** is opened, and commands in the file are executed as if they were entered at the command line. A number of macro files may be nested (i.e., the **`macro`** command can be called from macro files) to improve the flexibility of this function.
See also: **`loop`**.

Mask

Syntax: `mask [-a <function> -k <field>]`
Function: **Apply a user-defined mask function to a specified velocity field.**
Description:
This command can be used to filter, amplify, or in some way modify the *u*, *v* (and *w*) velocity components of a velocity field. Users can choose which field the mask is applied to using the optional **`-k`** parameter. By default, the mask is applied to the base flow (**`k`** = 0). Reference each perturbation field using numbers 1, 2, 3, etc.
The mask is specified using the **`-a`** option, and is supplied by the user as a mathematical function of available intrinsic and user-specified variables, such as **`x`**, **`y`**, **`t`**, **`RKV`**, etc.
The velocity fields are modified by evaluating the product of the velocity field with the mask function (β), as

$$\mathbf{u}_{\text{masked}} = \beta \mathbf{u}_{\text{original}} .$$

If no mask function is specified, the default mask is unity (no change to the velocity field).
This command is especially useful for filtering perturbation fields used in stability analysis. For instance, if the stability of a flow is being computed in a rotating frame, then the velocities far from the centre of rotation can be very large. This can lead to instability when random noise introduced at startup is being advected by high

rotational velocities in the base flow. In this case a Gaussian mask could be used to filter towards zero the perturbation field velocity far from the centre of rotation, providing a better initial field. For example,

```
\> mask -k 1 -a 'exp(-(x^2+y^2))'
```

could be used to isolate startup white noise to the region of interest, possibly improving the speed of convergence of the analysis.

Meshpts

Syntax: **`meshpts [-f <filename>]`**
Function: **Save mesh coordinates to a text file.**
Description:
This outputs the (x, y, z) coordinates and global node number (n) of every coordinate in a mesh, including interpolation points within each element. If no filename is specified, the default **`meshpts.dat`** is used.
The data is stored in text format at a high precision, so for large meshes these files can be very large.

Monitor

Syntax: **`monitor <string> [<x> <y> <z>]`**
Function: **Monitor velocity and pressure time history at a point on the mesh.**
Description:
A file named **`<string>`** is created, and the time history of a point in the velocity field is output at each time step. The point is taken as the nearest mesh node to the spatial coordinates **`<x>`**, **`<y>`**, (and **`<z>`** if in 3D). Existing files of the same name will be overwritten.
If the user wishes to cease recording to open monitor files, they can call monitor with the value of **`<string>`** set to "**`close`**" (case insensitive). This will close all currently open monitor files, and will free the resources allocated to them. Thus multiple time histories can be acquired from a single Viper session.
Note that monitor need only be called once for each file - the files are actually created during initialization (see `help init`).
See also: **`sample`**.

Pgrad

Syntax: **`pgrad <dpdx> <dpdy> [<dpdz>]`**
Function: **Impose components of a pressure gradient on the flow.**
Description:
Constant pressure gradient components can be imposed on a flow by specifying values of the parameters **`<dpdx>`**, **`<dpdy>`** (and **`<dpdz>`** in 3D), which correspond to the linear spatial kinematic static pressure gradient being imposed in each of the x-, y-, and z-directions, respectively.
Viper treats the kinematic static pressure as

$$P_{\text{tot}} = \widetilde{P} - \overline{\frac{\mathrm{d}P}{\mathrm{d}\mathbf{x}}}\mathbf{x},$$

where $\widetilde{P}$ is the disturbance pressure field obtained from the divergence-free projection of the velocity field during the pressure substep, $\overline{\frac{\mathrm{d}P}{\mathrm{d}\mathbf{x}}}$ is a vector of mean pressure gradient components, and $\mathbf{x}$ is the spatial coordinate vector. This feature can be employed to accurately model spatially periodic pressure-driven flows such as pipe flows, etc.

Pres

Syntax: **`pres`**
Function: **Toggle pressure substep on/off during time integration.**
Description:
Time integration is carried out by solving each of the advection, pressure and viscous diffusion terms consecutively. This function is used to switch off computation of the pressure term, which also stops the continuity (conservation of mass) constraint being enforced. The default setting of this feature is ON. This facility is primarily provided as a debugging tool.
Note that switching off the pressure term alters the equations being solved by Viper.
See also: **`diff`**, **`advect`**.

Quit

Syntax: **`quit`**
Function: **Exits Viper.**
Description:
Viper terminates immediately, and any unsaved work will be lost. This command performs the same action as **`stop`** and **`exit`**.
See also: **`exit`**, **`stop`**.

Rand

Syntax: **`rand [<level>]`**
Function: **Add a random perturbation to a 3D spectral-element/Fourier computation.**
Description:
If a spectral-element/Fourier computation has been initialised for time integration, then the **`rand`** command can be used to add some random white noise to the velocity fields to accelerate the development of instability modes or transient flow features. Without a call to **`rand`**, the user relies on noise at the limit of numerical precision to trigger the growth of three-dimensional flow in the non-zero Fourier modes, which can take considerable time.
Users should use **`rand`** with care if they are restarting a simulation (using **`load`**) from a saved spectral-element/Fourier computation, as the added noise will contaminate time histories of flow quantities captured over multiple runs.
If the optional **`<level>`** is not supplied, random noise with a magnitude of 10^{-4} is used.
See also: **`fourier`**.

Rotate

Syntax: `rotate <x> <y> <omega>`

Function: **Specify a rotating frame of reference for stability analysis on a frozen base flow.**

Description:

The command `freeze` artificially stops any time evolution of a flow field. For some flows, such as co-rotating vortex pairs, the base flow would otherwise rotate about some point in the flow. In essence, `freeze` transfers the computation into a frame of reference rotating with the base flow. However, for stability analysis, the evolution of the perturbation field is still computed as if it were in an inertial reference frame. Therefore, Coriolis and centrifugal accelerations due to the rotation are not included in the computation.

The command `rotate` is used to correct for these additional acceleration terms. The command `rotate` only has an effect on the perturbation field(s) of a two-dimensional Cartesian (not asixymmetric) computation where `freeze` has been called.

The command `rotate` takes as input the `<x>` and `<y>` coordinates of the centre of rotation of the computational domain, and the angular velocity of the rotation (`<omega>`, defined positive for anti-clockwise rotation, and expressed in radian per time unit).

The command `rotate` makes the following corrections to the calculation of the perturbation field:

1) The rotational velocity component is subtracted from the base flow, **U**, supplied to the advection term for calculation of the perturbation field evolution, and
2) The correction due to the Coriolis acceleration -2 $\boldsymbol{\omega}$ □ $\mathbf{v}_{\text{rotating}}$ is added to the evolution equations of the perturbation field.

Note that no contribution due to centrifugal effects is required, as this affects the evolution of the base flow.

See also: `freeze`.

Sample

Syntax: `sample [-f <filename> -x <x> <y> <z>]`

Function: **Get flow parameters at a physical location within the computational domain.**

Description:

This command outputs the time (*t*), the velocity components (*u*, *v*, *w*), velocity gradients (d*u*/d*x*, etc.), kinematic static pressure (*p*), and strain rate magnitude at a physical point on the mesh. Note that unlike the `monitor` command, the `sample` command will interpolate the flow quantities to the requested location, rather than just output the values at the nearest mesh node. Furthermore, the points are calculated and outputted to file at the time that `sample` is called. `sample` can only be called after `init`.

This command will append new data to the end of an existing file of the same name, if one exists.

The following options are available:

`-f <filename>`

Used to specify a filename **`<filename>`** (including extension) to save the flow values to. If the **`-f`** option is not specified, the default filename **`sample.dat`** is used.

`-x <x> <y> <z>`

Used to specify the spatial coordinates of a point in the computational domain at which to interpolate the flow values. Only two coordinates may be specified in 2D, and three values must be specified in 3D.

See also: **`monitor`**.

Samplef

Syntax: **`samplef [-f <filename> -x <x> <y> <z>]`**

Function: **Return the Fourier coefficients of the velocity field at a point.**

Description:

During a three-dimensional spectral-element/Fourier computation, this command outputs the time (*t*), the supplied spatial coordinates, and the Fourier coefficients of the velocity field at a physical point on the mesh.

Note that unlike the **`monitor`** command, **`samplef`** will interpolate the flow quantities to the requested location, rather than just output the values at the nearest mesh node. Furthermore, the points are calculated and output to file at the time that **`samplef`** is called. The command **`samplef`** can only be called after **`init`**.

This command will append new data to the end of an existing file of the same name, if one exists.

The following options are available:

`-f <filename>`

Used to specify a filename **`<filename>`** (including extension) to save the flow values to. If the **`-f`** option is not specified, the default filename **`sample.dat`** is used.

`-x <x> <y>`

Used to specify the spatial coordinates (in the *x-y* or *z-r* plane) of a point in the computational domain at which the Fourier coefficients are to be interpolated.

See also: **`autocorrf`**, **`energyf`**.

Save

Syntax: **`save [-f <filename> -k <floq_mode> -m -s]`**

Function: **Save flow field vectors to file.**

Description:

Saves flow field vectors, as well as computation parameters **`t`**, **`dt`**, **`RKV`**, and the number of elements and element polynomial degree to a user-specified file. These stored fields can then be reloaded using the **`load`** command to re-start a computation. If no filename is specified, then the default output filename **`ff_out.dat`** is used.

Update 4/11/2006: This command now saves files containing flow fields at the three previous time steps. This avoids the annoying perturbation that was added to flows upon re-start.

The following options are available:

`-f <filename>`

Used to specify a filename **`<filename>`** (including extension) to save the binary file to. If the **`-f`** option is not specified, the default filename **`ff_out.dat`** is used.

`-k <floq_mode>`

Used to specify an integer perturbation field number (i.e., 1, 2, ..., $N_{\mathrm{Floq\_modes}}$, when Floquet analysis is active) to load a saved flow field into. The default is **`<floq_mode>`** = 0, corresponding to the base flow.

`-m`

Specifies that you wish to save spatial coordinates to file also (this feature is only required if you wish to load data onto a different macro-element mesh.

`-s`

Used to include a number sequence in the filename. A 4-digit integer (e.g., **`_0001`**, **`_0002`**, **`_0003`**, etc.) is added to the default or user-specified filename, just prior to the file extension, if one is specified. Numbering begins at 1, and increments every time a **`save`** call is made with the **`-s`** option.

See also: **`load`**.

Scalar

Syntax: **`scalar <option>`**

Function: **Used to invoke functions relating to transport of a scalar field.**

Description:

Viper facilitates the transport of a passive scalar field (variable *S*) on a two- or three-dimensional flow field. The transport is computed using an auxiliary semi-Lagrangian advection-diffusion algorithm (see Maday, Patera & Rønquist, *J. Sci. Comp.*, **5**(4), 263-292, 1990).

To activate advection-diffusion transport of the scalar field *S*, the user must set boundary conditions for the scalar field in the **`viper.cfg`** file.

The following scalar transport options can be invoked with <operation> values:

`scalar diff <coeff>`

Defines the coefficient of diffusion for the scalar field. By default, a coefficient of diffusion of **`<coeff>`** = 1.0 is used. The value of this coefficient can be set in the **`viper.cfg`** file (see **`help gvar_scalar_diff`** for more information). A larger value will result in more diffusion (smearing) of the scalar field. A value of zero (pure advection) is not permitted due to numerical stability implications.

`scalar steps <Ntrack_steps>`

Adjusts the number of time steps per scalar field update (requires re-initialization). The code performs an auxiliary advective update of the scalar field every time step, and every **`<Nscalar_steps>`** time steps, the diffusion of the scalar field, as well as its updated distribution, are computed. By default, **`<Nscalar_steps>`** = 10.

See also: **`gvar_scalar_diff`**.

Set

Syntax: **`set <variable> <param_1> [... <param_n>]`**
Function: **Change the value of a configuration variable.**
Description:
Supported variables are:

- **`RKV`** Reciprocal kinematic viscosity (must be a positive floating-point number)
- **`dt`** Time step (must be a positive floating-point number)

The value of the variable and other parameters are input as **`<param>`** values as required. For example, to set the reciprocal kinematic viscosity to 173.5, type:

```
\> set RKV 173.5
```

Stab

Syntax: **`stab [<filename>]`**
Function: **Calculate Floquet multipliers for each linear instability mode.**
Description:
If Floquet linear stability analysis is being performed (call **`floq`** prior to **`init`**), this command calculates an estimate of the magnitude of the Floquet multiplier ($|\mu|$) for each mode, using the power method. The Floquet multiplier is a complex number related to the growth rate σ, and the base flow period T, by

$$\mu \equiv e^{\sigma T}.$$

Viper estimates $|\mu|$ by comparing an integral of the perturbation field at one period to one evaluated at the previous period. Over a sufficient number of periods, all but the fastest-growing mode wash out of the solution. If *N*(*t*) is a perturbation field integral evaluated at time *t*, then

$$|\mu| = N(t+T)/N(t),$$

providing the flow has evolved for a sufficient number of periods to isolate only the fastest-growing mode at the given wavelength.
The current time and the estimated Floquet multiplier magnitudes are writted to a file **`<filename>`**, or to the default **`floq_mult.dat`**.
If users wish to resolve the complex components of an instability mode, or multiple modes at a single wavelength, then they should employ **`arnoldi`** instead of **`stab`**, which determines eigenvalues and eigenvectors using an Implicitly Restarted Arnoldi Method.
See also: **`arnoldi`**, **`floq`**.

Step

Syntax: `step [<num_steps>]`
Function: **Performs a number of time integration steps.**
Description:
The optional integer `<num_steps>` need not be provided. If it is not specified, a single time integration step is computed, otherwise `<num_steps>` steps are computed.
See also: `stopcrit`.

Stop

Syntax: `stop`
Function: **Exits Viper.**
Description:
Viper terminates immediately, and any unsaved work will be lost. This command performs the same action as `exit` and `quit`.
See also: `exit`, `quit`.

Stopcrit

Syntax: `stopcrit <value>`
Function: **Abruptly ceases time integration subject to a stopping criterion.**
Description:
When evolving a solution to a time invariant (steady) state, the `max du` monitor, which monitors the maximum change in velocity between each successive time steps, reduces towards zero. It is sometimes desirable to compute only sufficient time steps to reach a steady state.
To facilitate this, the `stopcrit` command can be called to specify a critical value of `max du`, beyond which no further time stepping is conducted. By default, this function establishes a stopping criterion of 1 □ 10^{-12} (`1e10-12`). If this function is not called, time integration will not be prematurely arrested, regardless of the value of `max du`.
Notes:

1) This criterion also ceases any particle tracking or scalar field evolution.
2) Subsequent calls to `step` (e.g., in a subsequent `loop` iteration, say) will allow time stepping to resume, subject to the same stopping criterion.
3) The stopping criterion can be changed at any time. The stopping criterion can effectively be removed by setting `<value>` to a negative value.
4) After a set of time steps are ceased subject to this criterion, control passes to the next input command.

See also: `step`.

Tecp

Syntax: **`tecp [-f <filename> -n <plot_interp_pts> -k <Floquet_mode> -s -t]`**

Function: **Outputs a Tecplot binary data file.**

Description:

Outputs a Tecplot **`.plt`** binary data file containing spatial coordinates, velocity, pressure and various flow quantities such as vorticity, shear rate, etc. If **`tecp`** is called before **`init`**, only the mesh (*x*, *y*, and *z* coordinates of the mesh) is written to the Tecplot binary file, with the default file name **`tec_mesh.plt`**. If **`init`** has been called, the mesh information and fluid flow variables are output, with the default filename **`tec_out.plt`** being used.

The following options are available:

`-f <filename>`

Used to specify a filename **`<filename>`** (including extension) to save the Tecplot binary file to. If the **`-f`** option is not specified, the default filenames **`tec_out.plt`** or **`tec_mesh.plt`** are used, for post- and pre-initialization calls respectively.

`-k <Floquet_mode>`

Used to specify which velocity field is to be saved. **`<Floquet_mode>`** can be an integer between 0 and the maximum number of Floquet modes being computed. If this option is omitted, the default base flow field (mode zero) is saved.

`-n <plot_interp_pts>`

Used to specify a number of interpolation points along each element dimension for plotting. If this option is omitted, the data is plotted on the spectral element mesh interpolation points. Otherwise, an even distribution of points is used. **`<plot_interp_pts>`** must be an integer of at least 2. This option is helpful for improving the quality of the resulting plots.

`-o <level>`

The integer **`<level>`** is a value from 1 to 3 (default 3) which specifies the level of data included in the Tecplot binary file. The levels include the variables as described below:

1 Spatial coordinates, and velocity and pressure fields,

2 The above, plus velocity divergence, vorticity, velocity magnitude, and particle/scalar fields

3 The above, plus strain rate magnitude and directional components, and the λ_2 field used to identify vortex cores (see Jeong & Hussain, *J. Fluid Mech.* **285**, 69-94, 1995).

Numbers 1 or 2 can be used if less information is required. This will save time and storage space.

`-s`

Used to include a number sequence in the filename. A 4-digit integer (e.g., `_0001`, `_0002`, `_0003`, etc.) is added to the default or user-specified filename, just prior to the file extension, if one is specified. Numbering begins at 1, and increments every time a **`tecp`** call is made with the **`-s`** option.

`-t`

Specifies that data is to be written to an ASCII data file (Tecplot **`.dat`** file) rather than the default **`.plt`** file format.

Tec_floq

Syntax: **`tec_floq [-f <filename> -n <num_floq_planes> -k <Floquet_mode> -s -a <scale_factor> -t]`**

Function: **Generate 3D vorticity plot of Floquet mode for Tecplot.**

Description:

Outputs a Tecplot **`.plt`** binary data file containing spatial coordinates and vorticity components of a three-dimensional reconstruction of a Floquet mode superimposed onto the computed base flow.

The solution must be initialised, and at least one Floquet mode must be active for this command to be invoked.

The following options are available:

`-a <scale_factor>`

Used to specify a scaling factor (can be any real number) to be applied to the perturbation field prior to being superimposed onto the base flow velocity field. By default, a scaling factor of 1.0 is applied, though in many cases a significantly larger number may be required to visibly identify the perturbation on the base flow vorticity contours. Users are advised to experiment with this value to find what works for their specific cases.

`-f <filename>`

Used to specify a filename **`<filename>`** (including extension) to save the Tecplot binary file to. If the **`-f`** option is not specified, the default filename **`tec_floq3d.plt`** is used.

`-k <Floquet_mode>`

Used to specify which Floquet mode is to be plotted. **`<Floquet_mode>`** can be an integer between 1 and the maximum number of active Floquet modes. If this option is omitted, **`<Floquet_mode>`** defaults to the first active Floquet mode.

`-n <num_floq_planes>`

Used to specify the number of planed plotted in the spanwise (or azimuthal if using cylindrical coordinates) direction. If this option is omitted, the data is plotted onto 16 additional planes in the 3rd dimension.

`<num_floq_planes>` must be a positive integer, and powers of 2 are recommended to best symmetrically capture the features of the underlying sinusoidal expansion of the Floquet mode.

`-s`

Used to include a number sequence in the filename. A 4-digit integer (e.g., **`_0001`**, **`_0002`**, **`_0003`**, etc.) is added to the default or user-specified filename, just prior to the file extension, if one is specified. Numbering begins at 1, and increments every time a **`tecp`** call is made with the **`-s`** option.

`-t`

Specifies that data is to be written to an ASCII data file (Tecplot **`.dat`** file) rather than the default **`.plt`** file format.

Track

Syntax: **`track <operation>`**

Function: **Used to invoke functions relating to passive tracer particle tracking.**

Description:

Viper facilitates an accurate and flexible particle tracking facility. A (nearly) fourth-order accurate time integration scheme is used to advance the positions of passive virtual particles in the flow. This scheme employs a 4^{th}-order Runge—Kutta method to advance particles within elements, and a series of linear increments to step to and across element boundaries (see Coppola, Sherwin & Peiró, *J. Comput. Phys.* **172**, 356, 2001). Particles can either be injected from one or many spatial positions in the flow, or the flow can be seeded with a uniform concentration of particles.

The available options for particle tracking can be invoked with the following **`<operation>`** values:

`track steps [<Ntrack_steps>]`

Defines the number of computation time steps (Δt) per particle tracking time steps, where **`<Ntrack_steps>`** is an integer. If **`<Ntrack_steps>`** is omitted, the simulation will default to a value **`<Ntrack_steps>`** = 10.

`track inject_steps <Ninject_steps>`

Sets the number of particle time integration steps per particle injection. The default value is **`<Ntrack_steps>`** = 5.

`track inject`

Tracer injection points are loaded from a text file named **`track_pts`**, which includes the following numbers on a each line: firstly the number of injection points, then the *x*, *y* and *z*-coordinates of each point (only two spatial coordinates need be specified for two-dimensional computations). One injection point is given per line, and a large number of points may be established concurrently. During time integration, a new particle is injected at each of these locations every time particle positions are updated.

`track inject_off`

Ceases tracer injection and erases stored injector information from memory. Further injection can be initiated by calling **`track inject`**.

`track seed [<density>]`

The flow is seeded with an even distribution of tracer particles. Throughout the domain, particles are placed **`<density>`** units apart in the *x*, *y* (and *z*) directions. If **`<density>`** is omitted, a particle spacing of 0.1 is employed. For flows with inlets, the user may wish to maintain particle density by also including a call to **`track load`**, incorporating a rake of injection points.

`track sample [<filename>]`

Saves velocity field information at each particle location to a text file **`<filename>`**. If not supplied, the default filename is **`track_sample.dat`**. Particle information is output line by line, with each line containing: t, *x*, *y*, [*z*,] coordinates, *u*, *v*, [*w*]-velocities, velocity gradients, shear rate, and pressure.

This command will append new data to the end of an existing file of the same name.

`track save [<filename>]`

Saves information about particles to a text file **`<filename>`**. If no filename is given, a default file **`track_out.dat`** is created. Particle information is output line by line, with each line containing: **`<particle_number>`**, *x*, *y*, [*z*,] coordinates, and *u*, *v*, [*w*]-velocities.
This command will overwrite an existing file of the same name.

Vismat

Syntax: **`vismat`**
Function: **Output images showing the structure of the global matrices being solved.**
Description:
The sparse matrices used to solve the global boundary system for the pressure and viscous diffusion substeps can be visualized using this command.
Image files **`laplace_matrix.pgm`** and **`helmholtz_matrix.pgm`** are created, showing the structure of the matrices. Many of the matrices built by Viper are symmetrical so in these cases only the upper or lower diagonal may be visible.
Please inform the developer if you would find a binary image file format preferable for output.

Wvel

Syntax: **`wvel`**
Function: **Toggles *z*/*θ*-component of velocity on or off in two-dimensional computations.**
Description:
By default, Viper only computes in-plane velocity components in two-dimensional simulations (i.e., only *u* and *v*-velocity components in two-dimensional Cartesian coordinates). However, sometimes it is necessary to include the out-of-plane velocity component (i.e., the *θ*-velocity component in swirling flows in a cylindrical coordinate system, or the *w*-velocity component in the interaction of vortices with a non-zero axial velocity along their cores.
A call to **`wvel`** prior to calling **`init`** will activate the out-of-plane velocity component for two-dimensional computations. It has no effect on three-dimensional computations.
Note that the computations will still be two-dimensional – that is, there is still no variation (zero spatial gradients) in the third dimension.
See also: **`axi`**.

Chapter 8: References

Barkley, D. & Henderson, R.D. (1996) Three-dimensional Floquet stability analysis of the wake of a circular cylinder. *J. Fluid Mech.* **322**, 215-241.

Blackburn, H.M. & Sherwin, S.J. (2004) Formulation of a Galerkin spectral element-Fourier method for three-dimensional incompressible flows in cylindrical geometries. *J. Comput. Phys.* **179**(2), 759–778.

Blackburn, H.M., Barkley, D. & Sherwin, S.J. (2008) Convective instability and transient growth in flow over a backward-facing step. *Under consideration for publication in J. Fluid Mech.*

Coppola, G., Sherwin, S.J. & Peiró, J. (2001) Non-linear particle tracking for high-order elements. *J. Comput. Phys.* **172**, 356-386.

Huerre, P. & Monkewitz, P.A. (1985) Absolute and convective instabilities in free shear layers. *J. Fluid Mech.* **159**, 151-168.

Huerre, P. & Monkewitz, P.A. (1990) Local and global instabilities in spatially developing flows. *Annu. Rev. Fluid Mech.* **22**, 473-537.

Jeong, J. & Hussain, F. (1995) On the identification of a vortex. *J. Fluid Mech.* **285**, 69-94.

Karniadakis, G.E. (1990) Spectral element-Fourier methods for incompressible turbulent flows. *Comp. Meth. Appl. Mech. & Engng.* **80**, 367-380.

Karniadakis, G.E., Israeli, M. & Orszag, S.A. (1991) High-order splitting methods for the incompressible Navier—Stokes equations. *J. Comput. Phys.* **97**(2), 414-443.

Karniadakis, G.E. & Sherwin, S.J. (2005) *Spectral/hp Element Methods for Computational Fluid Dynamics (2nd Edition)*. Oxford University Press.

Lehoucq, R.B., Sorensen, D.C. & Yang, C. (1996) ARPACK users' guide: Solution of large scale eigenvalue problems by implicitly restarted Arnoldi methods. Tech. Report from http://www.caam.rice.edu/software/ARPACK/.

Leweke, T., Thompson, M.C. & Hourigan, K. (2004) Touchdown of a sphere. *Phys. Fluids*, **16**(9), Gallery of Fluid Motion.

Maday, Y., Patera, A.T. & Rønquist, E.M. (1990) An operator-integration-factor splitting method for time-dependent problems: application to incompressible fluid flow. *J. Sci. Comp.* **5**(4), 263-292.

Patera, A.T. (1984) A spectral-element method for fluid dynamics: laminar flow in a channel expansion. *J. Comput. Phys.* 54, 468-488.

Press, W.H., Teukolsky, S.A., Vetterling, W.T. & Flannery, B.P. (2002) Numerical recipes in C++: The art of scientific computing. *Cambridge University Press.*

Sheard, G.J., Leweke, T., Thompson, M.C. & Hourigan, K. (2007) Flow around an impulsively arrested circular cylinder. *Phys. Fluids* **19**(8), 083601.

Sheard, G.J., Thompson, M.C. & Hourigan, K. (2003) From spheres to circular cylinders: The stability and flow structures of bluff ring wakes. *J. Fluid Mech.* **492**, 147-180.

Sheard, G.J. & Ryan, K. (2007) Pressure-driven flow past spheres moving in a circular tube. *J. Fluid Mech.* **592**, 233-262.

Sorensen, D.C. (1995) Implicitly restarted Arnoldi/Lanczos methods for large scale eigenvalue calculations. *Tech. Report TR-96-40*. In: Keys, D.E., Sameh, A., Venkatakrishnan, V. (Eds.), *Parallel numerical algorithms.* Dordrecht, Kluwer.

Thompson, M.C., Hourigan, K. & Sheridan, J. (1996) Three-dimensional instabilities in the wake of a circular cylinder. *Exp. Therm. Fluid Sci.* **12**(2), 190-196.

Van Dyke, M. (1982) *An Album of Fluid Motion.* The Parabolic Press.

Williamson, C.H.K. (1996) Three-dimensional wake transition. *J. Fluid Mech.* **328**, 345-407.

Zang, T.A. (1991) On the rotation and skew-symmetric forms for incompressible flow simulations. *Appl. Numer. Math.* **7**, 27-40.