Reference Manual

Gregory J Sheard and Christopher J Camobreco
2 November 2018

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. Viper further includes the capability to solve the advection-diffusion transport of a scalar field in conjunction with the solution of an evolving fluid flow, and this field can be coupled with the momentum equations under the Boussinesq approximation to solve natural convection problems. It also has the ability to model some magnetohydrodynamic phenoma (quasi-2D and quasi-static problems). Furthermore, it is also able to perform linear stability and transient growth analysis.

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 at http://sheardlab.org/. 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 file 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:** If 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 (note that in some builds of Viper **1e-10** may not be valid, but **1.0e-10** always will be):
  ```
  .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
Trigonometric	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, $|x| \leq 1$	`acos(x)`
	Inverse tangent of x	`atan(x)`
	Cardinal (un-normalized) sine function of x	`sinc(x)`
	Sine integral function of x	`sini(x)`
	Cosine integral function of x	`cosi(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)`
	Exponential integral function of x	`expi(x)`
	Logarithmic integral function of x (exponential integral of the natural logarithm of x)	`logi(x)`
Bessel Functions	Bessel function of the 1st kind, of x, order 0	`besj0(x)`
	Bessel function of the 1st kind, of x, order 1	`besj1(x)`
	Bessel function of the 1st kind, of x, order n, for integers $n \geq 0$	`besjn(n,x)`
	Bessel function of the 2nd kind, of x, order 0	`besy0(x)`
	Bessel function of the 2nd kind, of x, order 1	`besy1(x)`
	Bessel function of the 2nd kind, of x, order n, for integers $n \geq 0$	`besyn(n,x)`
	Modified Bessel function of the 1st kind, of x, order 0	`besi0(x)`
	Modified Bessel function of the 1st kind, of x, order 1	`besi1(x)`
	Modified Bessel function of the 1st kind, of x, order n, for integers $n \geq 0$	`besin(n,x)`

	Modified Bessel function of the 2nd kind, of x, order 0	`besk0(x)`
	Modified Bessel function of the 2nd kind, of x, order 1	`besk1(x)`
	Modified Bessel function of the 2nd kind, of x, order n, for integers $n \geq 0$	`beskn(n,x)`
Error Functions	Error function of x	`erf(x)`
	Complimentary error function of x	`erfc(x)`
	Inverse error function of x	`ierf(x)`
	Inverse of the complimentary error function of $x, -1 < x < 1$	`ierfc(x)`
Fresnel Functions	Sine Fresnel integral function of x	`fress(x)`
	Cosine Fresnel integral function of x	`fresc(x)`
Elliptic Integral Functions	Complete elliptic integral of the 1st kind, K $-1 < x < 1$	`ellk(x)`
	Complete elliptic integral of the 2nd kind, E $-1 < x < 1$	`elle(x)`
	Incomplete elliptic integral of the 1st kind, F $-1 < x < 1, -\frac{\pi}{2} < \phi < \frac{\pi}{2}$	`iellf(x,`ϕ`)`
	Incomplete elliptic integral of the 2nd kind, E $-1 < x < 1, -\frac{\pi}{2} < \phi < \frac{\pi}{2}$	`ielle(x,`ϕ`)`
Other	Square root of $x, x \geq 0$	`sqrt(x)`
	Cube root of x	`cbrt(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)`
	Smoothed hat function (1 if $|x| < 0.5(1 - xs)$; 0 if $|x| > 0.5(1 + xs)$; or $0.5(1 + \cos(\pi(|x| - 0.5)/xs))$ otherwise	`hatsmth(x,xs)`
	Sawtooth function of x, ($x - floor(x)$ varying from 0 to 1)	`saw(x)`
	Gaussian function of x	`gauss(x)`
	Round to nearest whole number (e.g. 3.7 becomes 4.0)	`anint(…)`
	Truncate argument to nearest whole number (e.g. 3.45 becomes 3.0, -2.1 becomes -2.0)	`aint(…)`
	Return greatest integer less than or equal to argument (e.g. 3.2 becomes 3.0, -2.1 becomes -3.0)	`floor()`
	Return smallest integer greater than or equal to argument (e.g. 3.2 becomes 4.0, -2.1 becomes -2.0)	`ceiling()`
	Gamma function of x	`gamma(x)`
	Logarithm of the gamma function of $x, x > 0$	`lgamma(x)`

	Random number in the range $[0, x)$ Note: The result of this function is treated as always time- and space-varying	**rand(x)**

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 the 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 ($\leq$)	**<=**
Greater than ($>$)	**>**
Greater than or equal to ($\geq$)	**>=**
Equal to ($=$)	**= or ==**
Not equal to ($\neq$)	**!=**

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 scalar field **s**, the electric potential field **e**, 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

Function simplification by math parser:
Platforms: All
Symptoms: A mathematical expression, as part of a user defined variable or command input, may be read by Viper incorrectly. Simplifications such as performing addition, multiplication, removal of brackets etc. may lead to two operators being placed next to each other, such as '+-' or '--', which the parser may not be able to

simplify. An error message notifying that the function has not been simplified correctly will be provided.

Workaround: Always check **output.txt** files for math parser error messages. Rearrange terms and add brackets as necessary such that the function can be read correctly. Particularly, rather than '$A - B$' try '$A + (-B)$', where A and B are expressions which the math parser may also need to simplify. To observe the entire simplification process, use the verbose Viper executable, if available.

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

$$\rho\boldsymbol{g} - \boldsymbol{\nabla}p + \boldsymbol{\nabla} \cdot \boldsymbol{\tau}_{ij} = \rho\frac{D\boldsymbol{u}}{Dt}$$

where $\boldsymbol{g}$ is the gravity acceleration vector, p is a scalar pressure field, ∇ is the gradient operator, $\boldsymbol{\tau}_{ij}$ is the viscous stress tensor, $\boldsymbol{u}$ is a velocity vector, and t is time.

The velocity time derivative is sometimes referred to as the substantial derivative, which is defined

$$\frac{D\boldsymbol{u}}{Dt} = \frac{\partial\boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \boldsymbol{\nabla})\boldsymbol{u}$$

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

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

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

$$\tau = \mu\frac{du}{dy}$$

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

$$\frac{\partial\boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \boldsymbol{\nabla})\boldsymbol{u} = -\frac{1}{\rho}\boldsymbol{\nabla}p + \nu\nabla^2\boldsymbol{u} \tag{1a}$$

$$\nabla \cdot \boldsymbol{u} = 0 \qquad\qquad\qquad (1b)$$

where we introduce a kinematic viscosity

$$\nu = \frac{\mu}{\rho}.$$

Finally, equation (1a) can be used to reveal the single most important parameter describing the viscous behaviour of Newtonian fluids, the Reynolds number. If the length, velocity, time and kinematic pressure are respectively scaled by D, U_∞, D/U_∞ and ρU_∞^2, respectively, where D is a reference length scale and U_∞ is a reference speed, the momentum equation can be rewritten as

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \boldsymbol{\nabla})\boldsymbol{u} = -\boldsymbol{\nabla} p + \frac{1}{Re}\nabla^2 \boldsymbol{u}$$

where all quantities are now non-dimensional, and where the Reynolds number is defined as

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

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 $2n$-3, where n is the number of quadrature points. Illustrations of the nodal polynomial expansion basis employed by Viper are shown below.

$\phi_0(\xi) =$

$\phi_1(\xi) =$

$\phi_2(\xi) =$

$\phi_3(\xi) =$

$\phi_4(\xi) =$

$\phi_5(\xi) =$
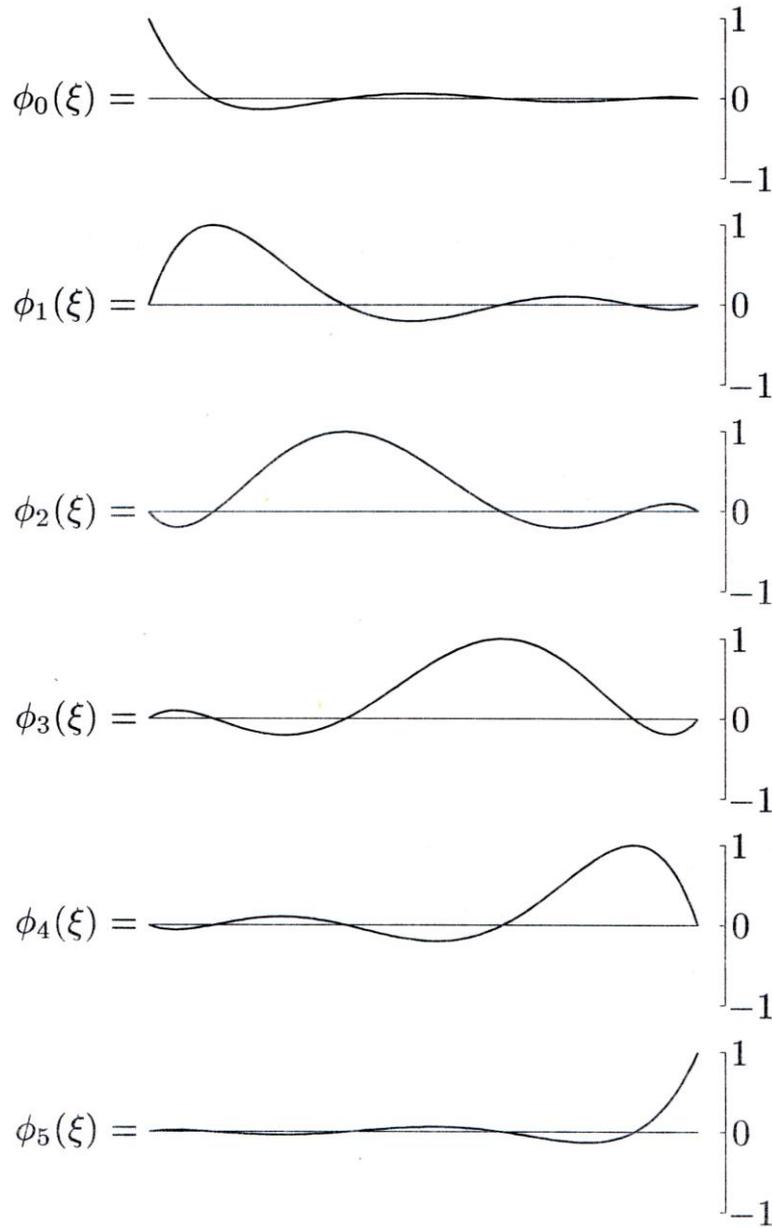
One-dimensional nodal expansion modes for a polynomial of degree 6 (from Karniadakis & Sherwin 2005).

$$\phi_{pq}(\xi_1, \xi_2) = h_p(\xi_1)h_q(\xi_2)$$
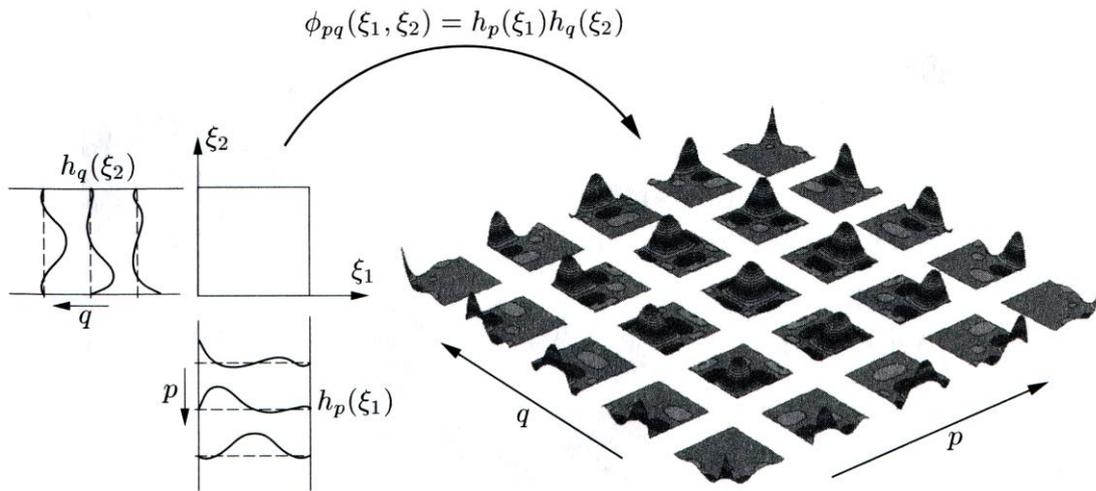
Construction of a two-dimensional nodal expansion basis from the product of two one-dimensional expansions of degree 5 (from Karniadakis & Sherwin 2005).

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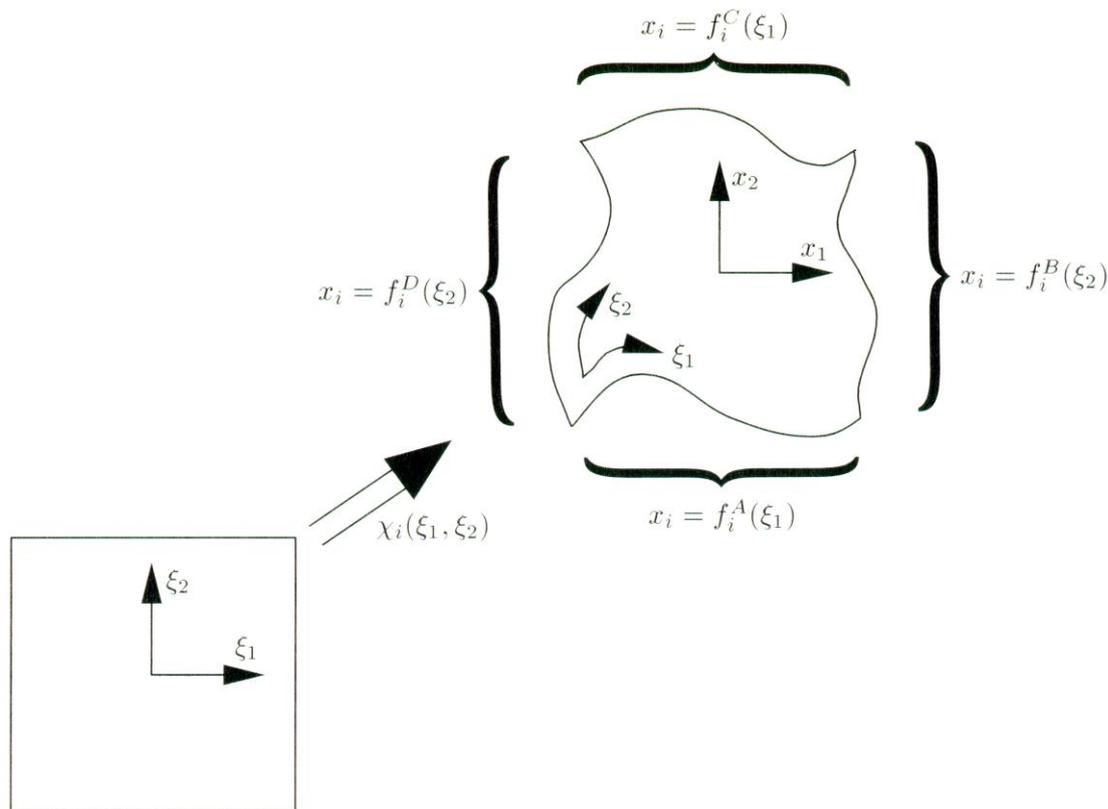
Mapping of a bi-unit square onto a general curvilinear quadrilateral element (from Karniadakis & Sherwin 2005). An analogous mapping onto a bi-unit cube is conducted for three dimensional hexahedral elements.

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.

12
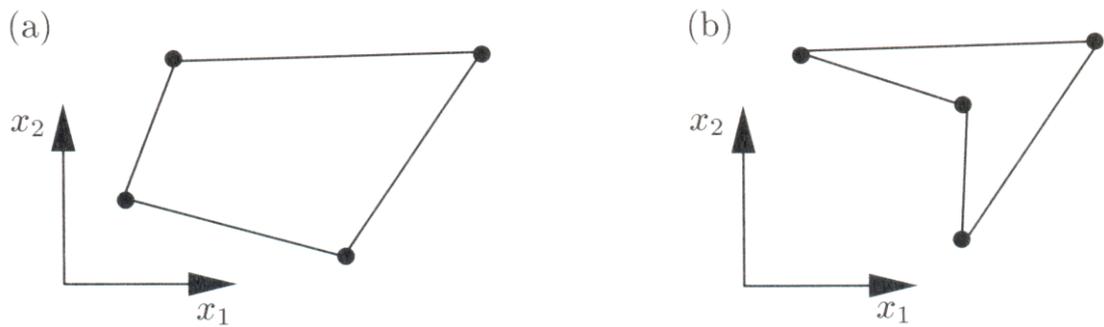
Examples of (a) valid and (b) invalid quadrilateral elements (from Karniadakis & Sherwin 2005).

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).
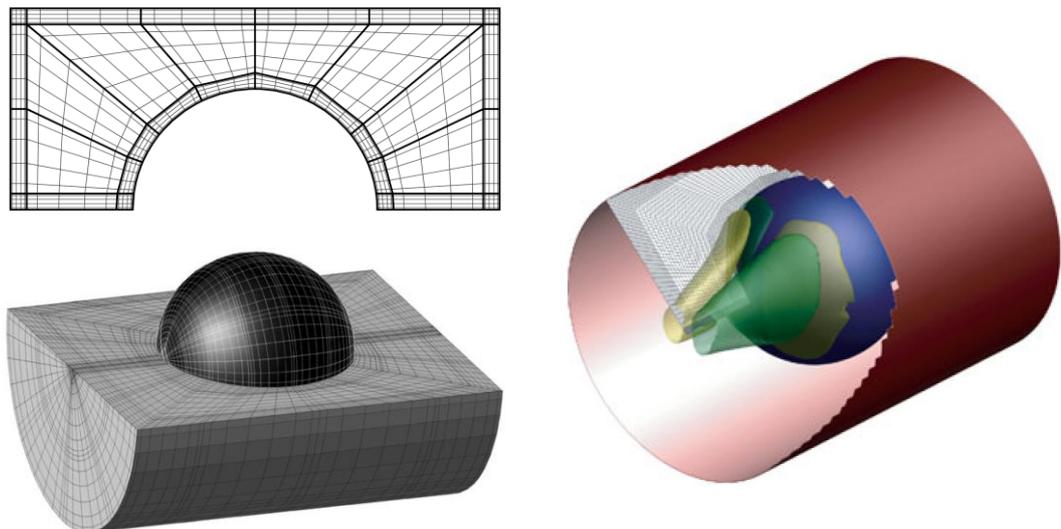
Left: Meshes employed for two- (top) and three- (bottom) dimensional computations of the axisymmetric and three-dimensional pressure-driven flows past spheres moving through a tube, respectively (Sheard & Ryan 2007). The upper half of the three-dimensional mesh has been removed to reveal the meshed surface 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

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

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

$$L\boldsymbol{u} = L_1\boldsymbol{u} + L_2\boldsymbol{u} + \cdots + L_m\boldsymbol{u} \, ,$$

then the solution that updates the variable $\boldsymbol{u}$ from time step n to $n + 1$ can be calculated by summing the contribution of each operator on $\boldsymbol{u}$ 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 $\boldsymbol{u}$ values at previous times required to find $\boldsymbol{u}^{n+1}$.

For the incompressible Navier—Stokes equations, Karniadakis, Israeli & Orszag (1991) propose a three-step time splitting scheme

$$\frac{\widehat{\boldsymbol{u}} - \sum_{q=0}^{J-1} \alpha_q \boldsymbol{u}^{n-q}}{\Delta t} = \sum_{q=0}^{J-1} \beta_q \mathbf{N}(\boldsymbol{u}^{n-q}) + \boldsymbol{F}(\boldsymbol{x}, t) + \boldsymbol{G}(\boldsymbol{x}, t)^{\mathrm{T}} \boldsymbol{I} \boldsymbol{u}^{n-q} \qquad (1c)$$

$$\frac{\widehat{\widehat{\boldsymbol{u}}} - \widehat{\boldsymbol{u}}}{\Delta t} = -\boldsymbol{\nabla} p^{n+1} \qquad (1d)$$

$$\frac{\gamma \boldsymbol{u}^{n+1} - \widehat{\widehat{\boldsymbol{u}}}}{\Delta t} = \frac{1}{Re} \nabla^2 \boldsymbol{u}^{n+1} \, , \qquad (1e)$$

where $\boldsymbol{F}(\boldsymbol{x}, t)$ and $\boldsymbol{G}(\boldsymbol{x}, t)$ are the coefficients for constant and linear forcing terms, $\boldsymbol{I}$ is the identity matrix, $\mathbf{N}(\boldsymbol{u})$ 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 pressure, p. We first take the divergence of both sides, and enforce the incompressibility constraint on the intermediate velocity field $\widehat{\widehat{\boldsymbol{u}}}$ as

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

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

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

The intermediate velocity field $\widehat{u}$ is calculated during the first substep, so this equation can be solved as a Poisson equation for the pressure p, with appropriate high-order Neumann boundary conditions for pressure 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 $\widehat{\widehat{u}}$ (Eq. 1d). The third substep involves solving a set of Helmholtz equations (Eq. 1e) for each of the velocity components, to determine the final velocity field u^{n+1}. Boundary conditions for the velocity field are imposed during this substep.

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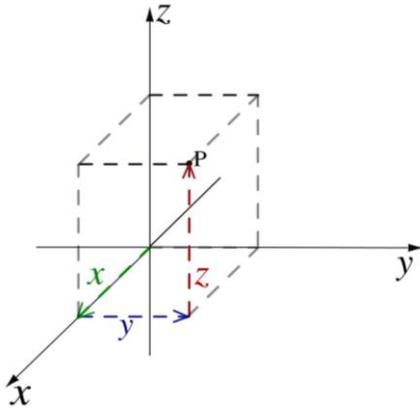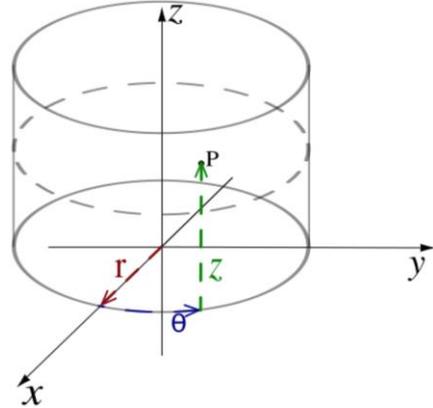
In three dimensions, the derivative operators acting on a scalar field in Cartesian coordinates are written as

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

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

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

and the divergence operator is written

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

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 $((\boldsymbol{u}\cdot\nabla)\boldsymbol{u})$, the rotation form $((\nabla\times\boldsymbol{u})\boldsymbol{u})$, and the skew-symmetric form $(\frac{1}{2}(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} + \frac{1}{2}\nabla(\boldsymbol{u}\boldsymbol{u}))$. 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

Note that pre-March 2013 Viper used to employ each of 3 possible forms of the advection operator (previously chosen using the **advect** command): convective, rotational, and skew-symmetric (though the rotation form is replaced by the convection form in cylindrical coordinates). Blackburn & Sherwin (2004) showed that the convection form produced results that converged slightly more rapidly than the skew-symmetric form with increasing spatial resolution. Furthermore, practice has demonstrated that similar convergence is achieved for each form, and the speed decrease for the rotational and skew-symmetric forms are therefore difficult to justify. Therefore, the convective form is used throughout the code from builds 12 March 2013 onwards.

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.

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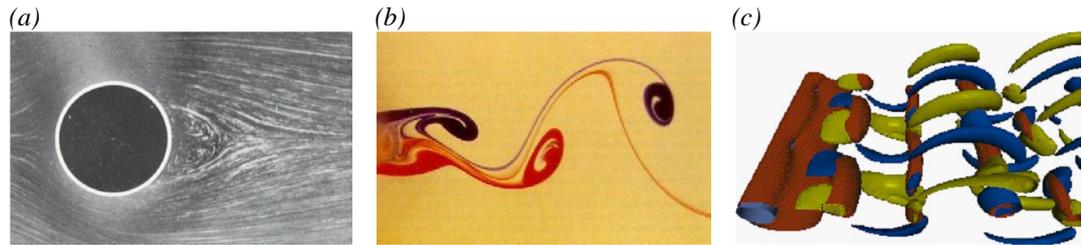
Instabilities developing in the wake of a circular cylinder. (a) The steady two-dimensional wake below $Re = 46$ (Van Dyke 1982), (b) the periodic two-dimensional Kármán vortex street above $Re = 46$, and (c) the three-dimensional "Mode A" wake above $Re \approx 190$ (Thompson, Hourigan & Sheridan 1996).

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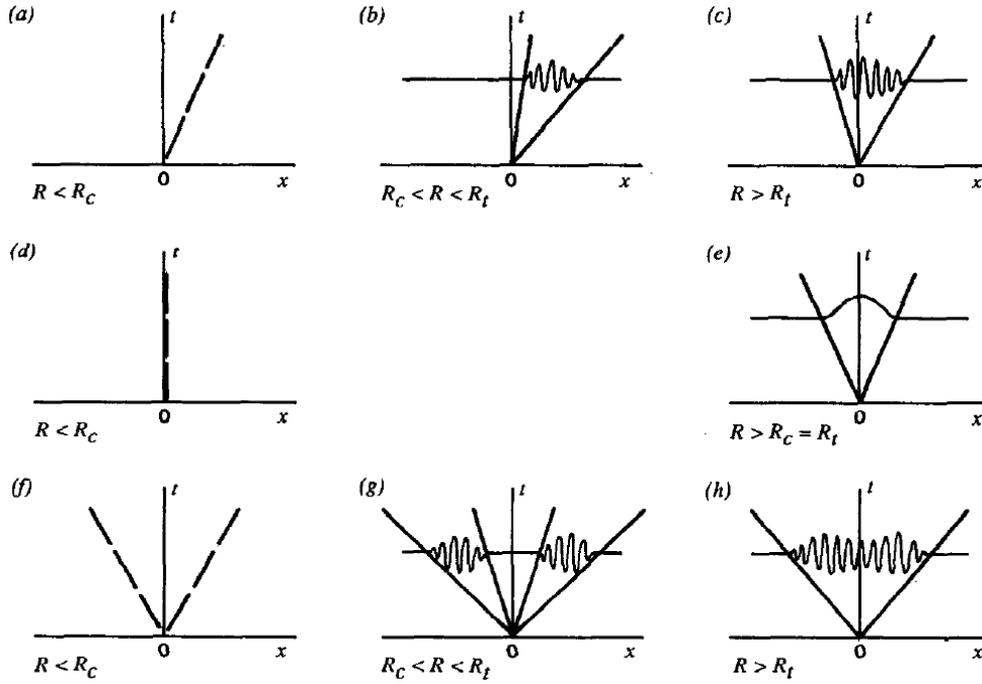
Instability responses. (a-c) Single travelling wave: (a) stable, (b) convectively unstable, (c) absolutely unstable. (d-e) Stationary mode: (d) stable, (e) absolutely unstable. (f-h) Conterpropagating travelling waves: (f) stable, (g) convectively unstable, (h) absolutely unstable. Figure reproduced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

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 $(\boldsymbol{u}, p)$ into a two-dimensional base flow $(\overline{\boldsymbol{u}}, \overline{p})$ and a three-dimensional disturbance $(\boldsymbol{u}', p')$,

$$\boldsymbol{u} = \overline{\boldsymbol{u}} + \boldsymbol{u}',$$
$$p = \overline{p} + p'.$$

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

$$\frac{\partial \boldsymbol{u}'}{\partial t} + (\overline{\boldsymbol{u}} \cdot \boldsymbol{\nabla})\boldsymbol{u} + (\boldsymbol{u} \cdot \boldsymbol{\nabla})\overline{\boldsymbol{u}} = -\boldsymbol{\nabla}p' + \frac{1}{Re}\nabla^2\boldsymbol{u}' \qquad (2a)$$

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

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,

18

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

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

$$\boldsymbol{u}'(x, y, z, t) = \begin{vmatrix} \hat{u}(x, y, t) e^{i\beta z} \\ \hat{v}(x, y, t) e^{i\beta z} \\ \hat{w}(x, y, t) e^{i\beta z} \end{vmatrix} ,$$

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

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 $\mathbf{A}$ as

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

The eigenvalues of $\mathbf{A}$ correspond to the Floquet multipliers of the system, $\mu = e^{\sigma T}$, where σ is the growth rate of the instability. The stability of the base flow $(\overline{\boldsymbol{u}}, \overline{p})$ 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 $\mathbf{A}$, though due to the size of the systems typically under investigation, $\mathbf{A}$ 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 (use command **buoyancy**). For computations employing this facility, the scalar field acts as a normalised temperature field, and the diffusion coefficient represents a thermal diffusion coefficient.

The Boussinesq approximation provides a means of coupling the momentum and scalar transport equations. An additional body force term is appended to the momentum equations. This term linearly relates the differences in relative temperatures to a buoyancy ratio, which appears as the buoyancy term

$$\boldsymbol{e}_y g \frac{\rho}{\rho_0},$$

where g is the acceleration due to gravity (in the $+\boldsymbol{e}_y$ direction, which is specified using the **buoyancy** command), ρ is the (local) fluid's density, and ρ_0 is a reference fluid density. This density ratio is related to the difference in relative temperatures via

$$\frac{\rho}{\rho_0} = 1 - \alpha(\hat{\theta} - \hat{\theta}_0)$$

where α is the volumetric thermal expansion coefficient of the fluid, $\hat{\theta}$ is a (dimensional) relative temperature, and $\hat{\theta}_0$ is a (dimensional) reference temperature.

The Boussinesq approximation provides an excellent means of adding additional accuracy to natural convection flows, by simulating the effects of buoyant

rising due to lower density fluid (and the accompanying circulation this induces). However, as compressible solvers incur significantly higher computational costs, the density differences simulated must remain small (all other terms in the equations solved treat the fluid as incompressible). This directly requires small temperature gradients for the Boussinesq approximation to remain valid. For more information on the validity of the Boussinesq approximation see Gray and Giorgini (1976), with two other key assumptions being that all other fluid properties (such as viscosity) remain independent of temperature and that viscous dissipation is negligible.

Regardless of the use of the Boussinesq approximation, the scalar fields are always calculated using the extrapolated velocity and scalar fields, after the advection operation (first substep). Then the resulting scalar field at the future time (n+1; both substeps are performed for the scalar equation) is input into the gravity term in the momentum equation.

Advection-Diffusion

The transport of a passive scalar field *s* on an evolving flow field $\boldsymbol{u}$ is described by

$$\frac{D\phi}{Dt} = \frac{\partial \phi}{\partial t} + (\boldsymbol{u} \cdot \boldsymbol{\nabla})\phi = \nu_s \nabla^2 \phi \tag{3}$$

where ν_s is the coefficient of diffusion for the scalar field. 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.

The same general form of time integration scheme is used (Karniadakis, Israeli & Orszag (1991)), however, as only advection and diffusion terms are present (there is no pressure field) a two-step scheme is employed. The velocity field after the first substep (of the momentum equations) is determined. Then the first substep of the scalar evolution equation is solved. This involves calculating both the scalar advection, and any u-velocity scalar forcing terms (due to `gvar_scalar_uvel_forcing`)

$$\frac{\hat{\phi} - \sum_{q=0}^{J-1} \alpha_q \phi^{n-q}}{\Delta t} = \sum_{q=0}^{J-1} \beta_q (\boldsymbol{u^{n+1}} \cdot \boldsymbol{\nabla})\phi + < \textbf{coeff} > u^{n+1} \,,$$

where `<coeff>` is the coefficient for scalar forcing (which is separate to the diffusion coefficient, and is defined with `gvar_scalar_uvel_forcing`).

The second substep involves solving a Helmholtz equation for the diffusion term,

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

which is when boundary conditions on the scalar field are imposed. Once the scalar field has been determined, the appropriate term in the momentum equation(s) are updated, if present (due to simulating the Boussinesq approximation; see `buoyancy`).

For second-order accuracy in time ($J = 2$), the required coefficients are:

Coefficient	Value
γ	3/2
α_0	2
α_1	$-1/2$
β_0	2
β_1	-1

Table: Second-order backwards-multistep scheme coefficients.

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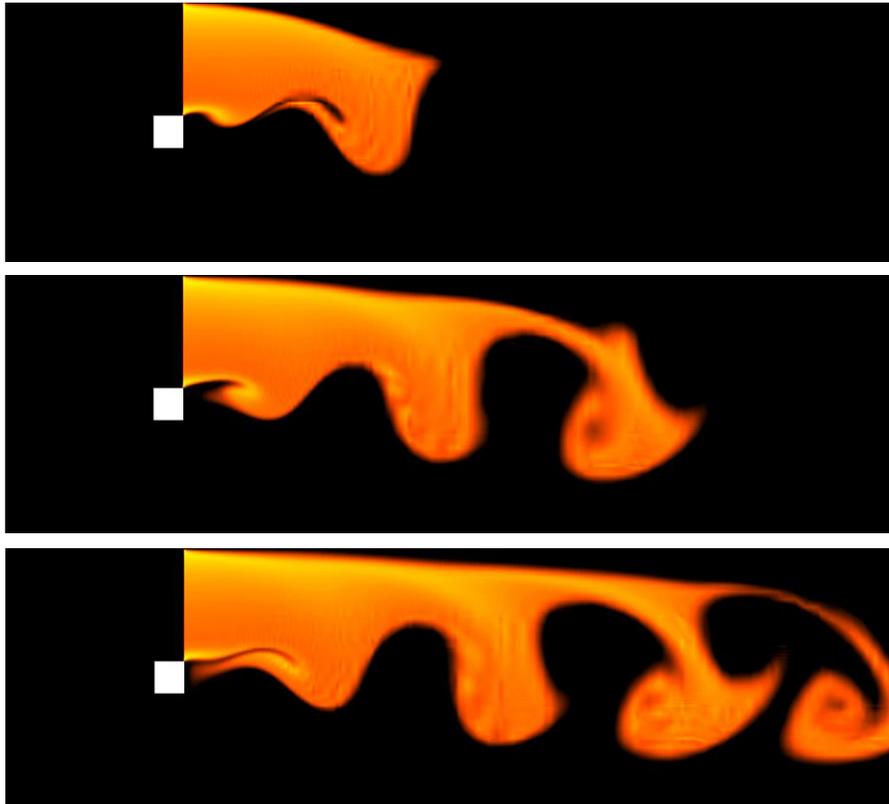
Contours of scalar field concentration, demonstrating fluid mixing behind a square cylinder at $Re = 90$ in a channel with blockage ratio 1/8.

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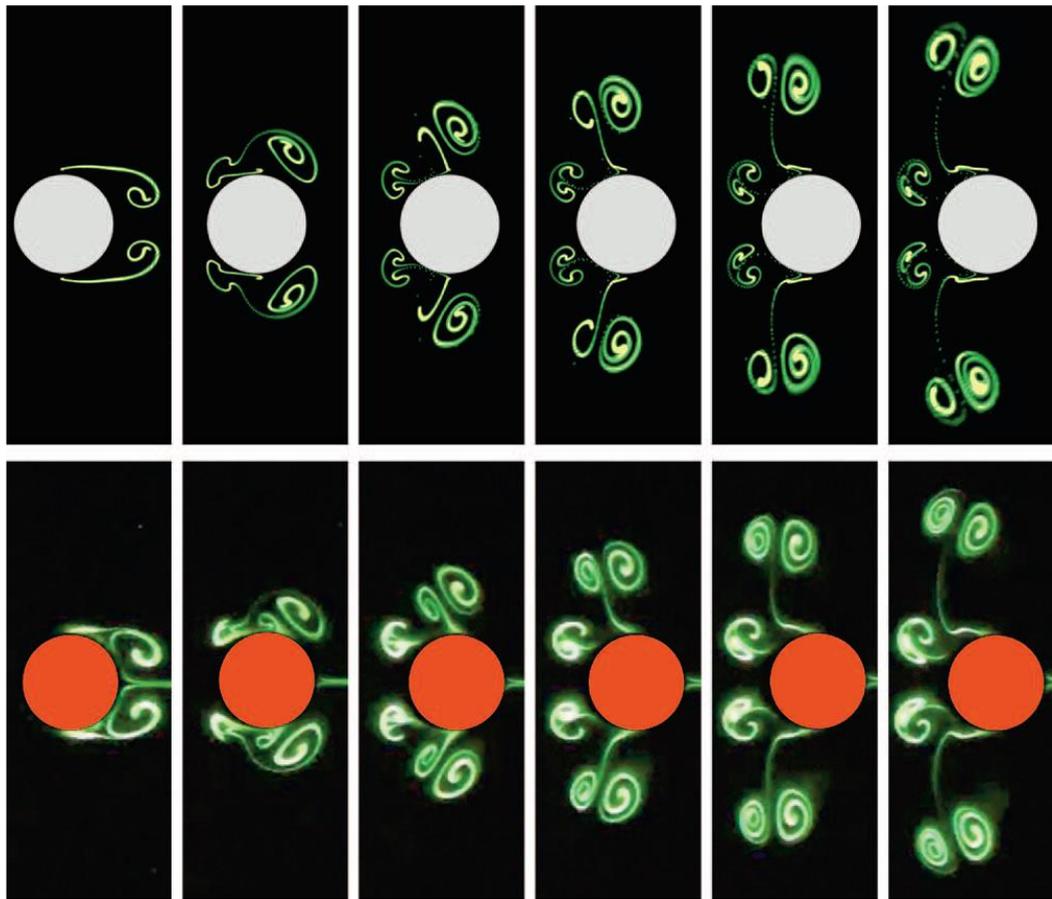
A time sequence (from left to right) comparing simulated particle tracking computations (top) and experimental dye visualization (bottom) for an arresting cylinder at $Re = 500$ with a translation distance of two cylinder diameters (Sheard, Leweke, Thompson & Hourigan 2007).

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.

Forcing Terms

The Navier-Stokes equations can be augmented with the addition of a variety of forcing terms, allowing for the modelling of various flows (these modifications are also able to be used in concert with other modifications, such as the use of the Boussinesq approximation). These can take the form of either constant forcing terms, defined with **gvar_forcing_f[u,v,w,s]**, or linear forcing terms, defined with **gvar_forcing_g[u,v,w,s]**, where the options represent the equation which will be modified (u-, v- or w-velocity component momentum equation, or the scalar field equation). The linear forcing terms are always linear in the respective component for the equation (the u-momentum equation can have a term linear in u-velocity appended). The only exception is the hard coded **gvar_scalar_uvel_forcing** which appends a term to the scalar equation which is linear in the u-velocity component. In either case a coefficient for the term can be specified, which can be a function of all native or used defined variables in the **viper.cfg** file, and is zero by default. Some common examples follow:

For periodic flows (infinite length ducts, channels or boundary layers), the pressure can be decomposed into a driving background pressure gradient, and a fluctuating component,

$$p = -\frac{2}{Re}x + p',$$

where the coefficient $(-2/Re)$ is unique to the boundary layer problem this example is from (and which is negative as the pressure gradient decrease in the direction of increasing flow velocity, which in this case is the positive $\boldsymbol{e}_x$ direction). Hence, the momentum equation is rewritten as,

$$\frac{\partial \boldsymbol{u}}{\partial t} = -(\boldsymbol{u}\cdot\boldsymbol{\nabla})\boldsymbol{u} - \nabla p' + \frac{2}{Re}\boldsymbol{e}_x + \frac{1}{Re}\nabla^2\boldsymbol{u}.$$

The implementation within the configuration file is shown below, nothing that the forcing must be applied only to the u-velocity component equation, for a background pressure gradient in the positive $\boldsymbol{e}_x$ direction:

```
gvar_usrvar Re 200              (Reynolds number)
gvar_rkv 'Re'
gvar_forcing_fu '2/Re'
```

Note that the Reynolds number only represents the reciprocal kinematic viscosity if the characteristic length and velocity scales are non-dimensionalized to a maximum of 1. This is highlighted here as the characteristic velocity will likely vary as the simulation evolves, particularly if shear and other diffusive effects act upon the velocity profile. Hence, although the original pressure gradient defined is constant, its effect on the flow will not allow it maintain a constant characteristic velocity. This effect can be quite large if the effects of buoyancy also modify the velocity field (natural convection). In such cases, the **forceflow** command may be preferable, as this adjusts the forcing such that the flowrate remains constant, and hence the characteristic (reference) velocity is held constant. If this disparity seems concerning, the flow fields which evolve (both

when varying with time, and when steady state) are very similar, it is just that in one case the forcing varies and the flow rate is held constant (**forceflow**) and in the other the forcing is held constant and the flow rate varies (**gvar_forcing_fu**).

A common use for the linear (gradient) terms (**gvar_forcing_g[u,v,w,s]**) are to represent linear friction (Rayleigh friction) or Coriolis forces. This particular example refers to the simulation of a magnetohydrodynamic flow, discussed in the following section, where the effects of Hartmann braking are simulated by linear friction.

The momentum equation for a quasi-2D magnetohydrodynamic flow (in which the pressure has not been decomposed, and hence which will be simulated using **forceflow**) is:

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \nabla)\boldsymbol{u} = -\nabla p + \frac{1}{Re}\nabla^2 \boldsymbol{u} - \frac{H}{Re}\boldsymbol{u}$$

The final term, with coefficient $(-H/Re)$ for a confined duct flow, represents the linear friction term. This will be implemented in the configuration file as,

```
gvar_usrvar Re 200          (Reynolds number)
gvar_usrvar H  100          (Hartmann friction parameter)
gvar_rkv 'Re'
gvar_forcing_Gu '-H/Re'
gvar_forcing_Gv '-H/Re'
```

In this case two forcing terms are needed (it is a two-dimensional simulation), as the friction term acts on both the u- and v-velocity components, as denoted by $\boldsymbol{u} = (u, v)$. Even though the reader may yet to be introduced to magnetohydrodynamics, the ease with which the forcing terms can be implemented shows how broad reaching their impact can be regarding flow modelling.

A final example considers the additional, hard coded forcing term, **gvar_scalar_uvel_forcing** which allows for an additional advection term to be placed in the scalar advection-diffusion equation (although only in the x direction). This is beneficial when simulating a heat flux distributed along the x-direction. Note that an energy balance over a control volume is required to determine the effect of the thermal gradient along the duct.

The dimensional scalar advection-diffusion equation is written as:

$$\frac{\partial \hat{\theta}}{\partial \hat{t}} + (\hat{\boldsymbol{u}} \cdot \hat{\nabla})\hat{\theta} = \kappa \hat{\nabla}^2 \hat{\theta},$$

where the temperature (similar to the pressure) is decomposed into periodic fluctuation and a background horizontal thermal gradient,

$$\hat{\theta} = \hat{\theta}' + A\hat{x},$$

where the background gradient,

$$A = \frac{\kappa(\mathrm{d}\hat{\theta}/\mathrm{d}\hat{y})_{\mathrm{w}}}{\hat{Q}},$$

will need to be determined for the exact problem based on an energy balance, but should have a form similar to that provided above. On substitution into the advection term of the scalar transport equation, their will be two components, the conventional advection of the fluctuating temperature,

$$(\hat{\boldsymbol{u}} \cdot \hat{\nabla})\hat{\theta}'$$

and the advection of the thermal gradient

$$(\hat{\boldsymbol{u}} \cdot \hat{\nabla})A\hat{x} = \hat{\boldsymbol{u}}A$$

Hence, after non-dimensionalization, there will be a gradient in temperature which is advected, or forced, by the u-velocity component, which requires the use of **gvar_scalar_uvel_forcing**. This merely requires the coefficient to be specified in the configuration file.

Hopefully, the breadth of the uses of the forcing terms has been appropriately described, as they allow for a wide variety of problems to be modelled.

Magnetohydrodynamics (and the SM82 Model)

The motion of an electrically conducting fluid in the presence of a magnetic field is considered to be a magnetohydrodynamic problem, which assumes that the velocity and magnetic fields are coupled. In general, the approximation of a low magnetic Reynolds number is used to decouple the velocity and magnetic fields, with the magnetic Reynolds number defined as:

$$Re_m = \sigma\mu uL = \frac{uL}{\lambda},$$

where σ is the electrical conductivity of the fluid, μ the permissivity of free space, and $\lambda = (\sigma\mu)^{-1}$ the magnetic diffusivity (u and L are characteristic fluid length and velocity scales). The magnetic Reynolds number represents the rate of advection of the magnetic field (if it is frozen into the fluid) to the rate of diffusion of the magnetic field. The formation of the term, from an order of magnitude analysis of the advection-diffusion equation of the magnetic field, $\boldsymbol{B}$, can be found in Davidson (2001). If the magnetic Reynolds number is low, the magnetic field is dominated by diffusion, and the velocity and magnetic field equations are decoupled (the magnetic field influences the velocity field, but the velocity field does not influence the imposed magnetic field). A full discussion of the electromagnetic MHD equations, and the full quasi-static approximation briefly discussed above, can be found in Davidson (2001).

The magnetic field influences the velocity field through the Lorentz force, given by $\boldsymbol{j} \times \boldsymbol{B}$, where $\boldsymbol{j}$ is the current density. This may be induced by the change in magnetic flux as a material surface of the fluid moves through the magnetic field (from Faraday's law of induction), or if it is externally applied by a voltage difference (or both). Although there are significantly more complexities, the presence of the Lorentz force is the key difference between MHD and OHD flows. The effect of the Lorentz force is

to reduce velocity differentials (higher velocity fluids have larger current densities), and creates much thinner boundary layers. These scale based on the Hartmann number, which is the ratio of the square of the strength of electromagnetic to viscous forces,

$$Ha = \left(\frac{\sigma B^2 L^2}{\rho \nu}\right)^{1/2}$$

where B represents the strength of the (imposed) magnetic field. An important component of the Hartmann number is the magnetic damping time $\tau^{-1} = \sigma B^2/\rho$ which represents the rate at which momentum diffuses along magnetic field lines (see, for example, Davidson (1995), Sommeria and Moreau (1982), Pothérat (2007)). The strength of the magnetic field also strongly defines the thickness of the boundary layers, which on walls perpendicular to the magnetic field scale as Ha^{-1} and on walls parallel to the field as $Ha^{-1/2}$. When considering finite geometries the Hartmann friction parameter $H = n(L^2/a^2)Ha$ may be more appropriate, where n is the number of walls perpendicular to the field, a the distance between two Hartmann walls and L the characteristic length scale (see Pothérat (2007)).

Finally, an interaction parameter is defined, which represents the ratio of electromagnetic to inertial forces,

$$N = \frac{Ha^2}{Re}.$$

These three key parameters are of great importance to the validity of the SM82 model (Sommeria and Moreau (1982)). If diffusion of momentum (along magnetic field lines) occurs much more rapidly than transfer of momentum due to viscosity, then flow structures will be elongated along field lines. This requires electromagnetic forces which are much stronger than both inertia or viscosity, hence $N \gg 1$ and $Ha \gg 1$. A $Re \gg 1$ also helps supress velocity variations between transverse planes. Finally, under the quasi-static approximation (time steady magnetic fields, and $Re_m \ll 1$) the flow can be assumed to be quasi two-dimensional (averaging the flow along the field lines). The flow simulated is truly two dimensional, hence to account for the difference (as the Hartmann boundary layers break the two-dimensionality) an additional linear Hartmann braking term must be appended to the momentum equation, which takes the form

$$-\frac{H}{Re}\boldsymbol{u}.$$

This linear friction term (which is formally accurate to the first order in N) can easily be simulated using linear forcing terms defined in the **`viper.cfg`** file (**`gvar_forcing_gu`** and **`gvar_forcing_gv`**).

Quasi-static MHD

Unlike the SM82 model, the quasi-static solver computes the electric potential field, and hence requires electric potential boundary conditions to be specified. Note that commands that can output information on other fields (such as the scalar field), are not necessarily equipped to output information about the electric potential field (hence use the **`current`** command). Furthermore, the quasi-static MHD equations can only simulate an electric field in one dimension, as noted hereafter (see Appendix A for derivations, that indicate which magnetic field directions have been hard coded).

The quasi-static equations to be solved are:

$$\frac{\partial \boldsymbol{u}}{\partial t} = \mathbf{N}(\boldsymbol{u}) - \boldsymbol{\nabla} p + \frac{1}{Re}\nabla^2 \boldsymbol{u} + N(\boldsymbol{j} \times \boldsymbol{e_B})$$

$$\boldsymbol{\nabla} \cdot \boldsymbol{u} = 0$$

where $\mathbf{N}(\boldsymbol{u}) = -(\boldsymbol{u} \cdot \boldsymbol{\nabla})\boldsymbol{u}$ is the non-linear advection operator, $\boldsymbol{j}$ is the induced current density, N the interaction parameter and $\boldsymbol{e_B}$ is a unit vector in the direction of the magnetic field. Ohm's law defines the current density as

$$\boldsymbol{j} = -\boldsymbol{\nabla}\phi + \boldsymbol{u} \times \boldsymbol{e_B}$$

where ϕ represents the electric potential field. The MHD approximations (discussed in Davidson (2001)), require solenoidal (closed loop) currents, hence requiring

$$\boldsymbol{\nabla} \cdot \boldsymbol{j} = 0 \therefore \nabla^2 \phi = \boldsymbol{\nabla} \cdot (\boldsymbol{u} \times \boldsymbol{e_B}) \therefore \nabla^2 \phi = \boldsymbol{e_B} \cdot (\boldsymbol{\nabla} \times \boldsymbol{u})$$

where the last modification uses a vector identity and requires a uniform magnetic field.

Following a similar approach to Karniadakis, Israeli & Orszag (1991), to integrate from time n to time $n+1$, the equations are cast at the future time, the time derivative term is replaced by a backwards differencing relation, and an appropriate-order extrapolation of the non-linear term to the future time is used. The momentum equation then becomes

$$\frac{\gamma_0 \boldsymbol{u}^{n+1} - \sum_{q=0}^{J_i-1} \alpha_q \boldsymbol{u}^{n-q}}{\Delta t}$$
$$= \sum_{q=0}^{J_e-1} \beta_q \mathbf{N}(\boldsymbol{u}^{n-q}) - \boldsymbol{\nabla} p^{n+1} + \frac{1}{Re}\nabla^2 \boldsymbol{u}^{n+1} + N(\boldsymbol{j}^{n+1} \times \boldsymbol{e_B}).$$

The same coefficients are used for time integration as the velocity field. The solution of the momentum equation is divided into three sub-steps, almost identically to the integration of the velocity field, although with additional terms present:

$$\frac{\boldsymbol{u}^* - \sum_{q=0}^{J_i-1} \alpha_q \boldsymbol{u}^{n-q}}{\Delta t} = \sum_{q=0}^{J_e-1} \beta_q \mathbf{N}(\boldsymbol{u}^{n-q}) + N(\boldsymbol{j}^{n+1} \times \boldsymbol{e_B}),$$
$$\frac{\boldsymbol{u}^{**} - \boldsymbol{u}^*}{\Delta t} = -\boldsymbol{\nabla} p^{n+1},$$
$$\frac{\gamma_0 \boldsymbol{u}^{n+1} - \boldsymbol{u}^{**}}{\Delta t} = \frac{1}{Re}\nabla^2 \boldsymbol{u}^{n+1}.$$

Poisson equations are solved for the electric potential field and the pressure. The sequence of calculations is therefore:

1. Extrapolate velocity field to $n+1$ time:

$$\widetilde{\boldsymbol{u}}^{n+1} = \sum_{q=0}^{J_e-1} \beta_q \boldsymbol{u}^{n-q},$$

2. Obtain electric potential field from solution of Poisson equation (note the electric potential field boundary conditions are imposed during this calculation):

$$\nabla^2 \tilde{\phi}^{n+1} = \boldsymbol{\nabla} \cdot (\tilde{\boldsymbol{u}}^{n+1} \times \boldsymbol{e}_B),$$

3. Calculate current density:

$$\tilde{\boldsymbol{j}}^{n+1} = - \boldsymbol{\nabla}\tilde{\phi}^{n+1} + \tilde{\boldsymbol{u}}^{n+1} \times \boldsymbol{e}_B,$$

4. Evaluate first intermediate velocity field:

$$\boldsymbol{u}^* = \sum_{q=0}^{J_i-1} \alpha_q \boldsymbol{u}^{n-q} + \Delta t\big(\mathbf{N}(\tilde{\boldsymbol{u}}^{n+1}) + N(\tilde{\boldsymbol{j}}^{n+1} \times \boldsymbol{e}_B)\big),$$

5. Obtain pressure from solution of Poisson equation (this is constructed by taking the divergence of the pressure sub-step, and enforcing the divergence-free constraint on the second intermediate velocity field; the pressure boundary conditions are imposed during this calculation):

$$\nabla^2 p^{n+1} = (\boldsymbol{\nabla} \cdot \boldsymbol{u}^*)/\Delta t,$$

6. Evaluate third intermediate velocity field:

$$\boldsymbol{u}^{**} = \boldsymbol{u}^* - \Delta t \boldsymbol{\nabla} p^{n+1},$$

7. Obtain the final velocity field from the Helmholtz equations (the velocity boundary conditions are imposed during this calculation):

$$\nabla^2 \boldsymbol{u}^{n+1} - \frac{\gamma_0 Re}{\Delta t} \boldsymbol{u}^{n+1} = -\frac{Re}{\Delta t} \boldsymbol{u}^{**}.$$

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

Viper is parallelized using the Message Passing Interface (MPI). With MPI, separate copies of the program are run on each processor, with each being allocated its own block of memory. MPI supplies routines that facilitate communication of data between each processor, synchronization, etc.

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 overhead due to duplication of data structures (mesh connectivity, derivative matrices, etc.) on each MPI process. There is also time lost when processes sit idle waiting for others to reach a collective MPI communication routine, as well as for the communications between processes. 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 this overhead.

To gain the most benefit from parallel computations, care is required to ensure that an appropriate number of MPI processes are used. For instance, if a simulation contains 5 flow fields, and the user chooses to run the simulation over two MPI processes, then one process will compute two fields and sit idle while the other process carries out the necessary calculations for its third field. 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 MPI processes 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 MPI processes for their computations.

Parallel base flow simulations

Viper is written to parallelize simulations by distributing multiple fields across multiple processes. Single-field computations (i.e. two-dimensional quadrilateral and three-dimensional hexahedral simulations) obtain no speedup if run across multiple processes; these simulations are most efficiently carried out on a single processor.

Parallel linear stability and optimal growth analysis computations

These simulations require the simultaneous computation of a two-dimensional base flow field and one or more perturbation fields. Each individual perturbation field must be evolved iteratively over a specified time interval. The total compute time required for a single field is problem and parameter dependent, but typically varies between tens to hundreds of hours. There is a one-way coupling only in this algorithm, where the perturbation fields depend on the base flow velocity fields from previous time steps, but not vice versa. Hence, the master process evolving a base flow solution is required to communicate this solution via a single MPI_BCAST communication at every time step to the processes on which the perturbation fields are being evolved. This algorithm therefore exhibits a strong parallel scalability.

It is possible to acquire, in a single large parallel job, a comprehensive spectrum of instability growth rates as a function of perturbation wavenumber. Further efficiency gains are possible where available RAM permits by clustering multiple perturbation fields onto each MPI process; this reduces the message-passing overhead and improves the efficiency of the project-wide utilisation of resources. A short time integration test depicts the scalability of this algorithm. The reference case involved time integration of a base flow and 1 or more perturbation fields evolved on a single processor, and the compute time was compared with that from a set of simulations where the job was distributed across multiple processors. The figure below plots the resulting compute time, demonstrating a negligible increase in compute time in the distributed case from 1 to 4 fields (processors), an approximately 30% increase in time between 4 and 16 fields (processors), and constant compute time from 16 to beyond 1000 fields (processors). To compare, the times from the 1-CPU reference case divided by the

number of fields (processors) are also plotted. Memory restrictions limited this reference case to 64 fields.
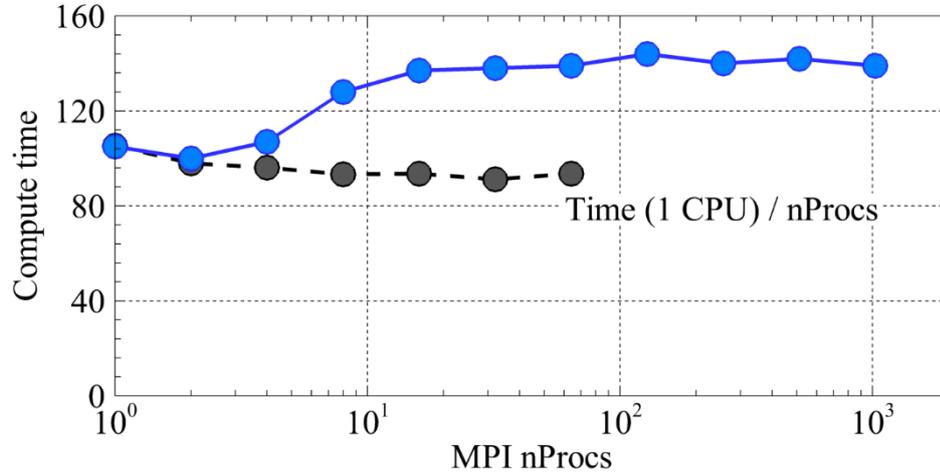
Time integration of a two-dimensional base flow on MPI process 0 plus nProcs-1 test cases over processes 1 to nProcs-1. The compute time for this test case is denoted by the blue unbroken line, while for reference the average time per field from 1-CPU simulations are plotted with the black dashed line.

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 MPI processes that should be used when computing linear stability analysis computations is $N_p + 1$. To avoid idle MPI processes, users should compute with either this number of MPI processes, 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 MPI process. Less efficient speedup would be achieved for computations using 7, 6, 5, or 3 MPI processes.

Parallel spectral-element/Fourier computations

The spectral-element/Fourier algorithm computes a three-dimensional solution where a Fourier series represents the variation in the flow in the out-of-plane direction. MPI is therefore useful for the efficient parallel computation of three-dimensional flows in domains with geometric homogeneity in one dimension.

This algorithm predominantly evolves the flow in Fourier space (pressure and diffusion substeps), permitting much of the work at every time step to be conducted in isolation on Fourier modes distributed across MPI processes. However, within each time step, the flow must be transformed back to physical space for calculation of a nonlinear advection term, before being transformed back to Fourier space. This is accomplished via a pair of MPI_ALLTOALLV() calls bracketing inverse and forward Discrete Fourier transforms. This erodes the scalability of the routine when compared to the more lightly coupled linearised solver.

---

[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.

A short time-evolution test of a representative 3D flow simulations containing between 2 and 256 Fourier modes distributed over the same number of processors. Figure 2 shows that the compute time increases 1.89 times from the 2-field/2-processor case to the 256-field/256-processor case. The reference case computed on a single processor demonstrates a consistent average compute time up to the maximum 64 fields that could be accommodated in a single-CPU job. The code facilitates further efficiency gains where RAM limitations permit by clustering multiple Fourier modes onto each MPI process, which further reduces the message-passing overhead.
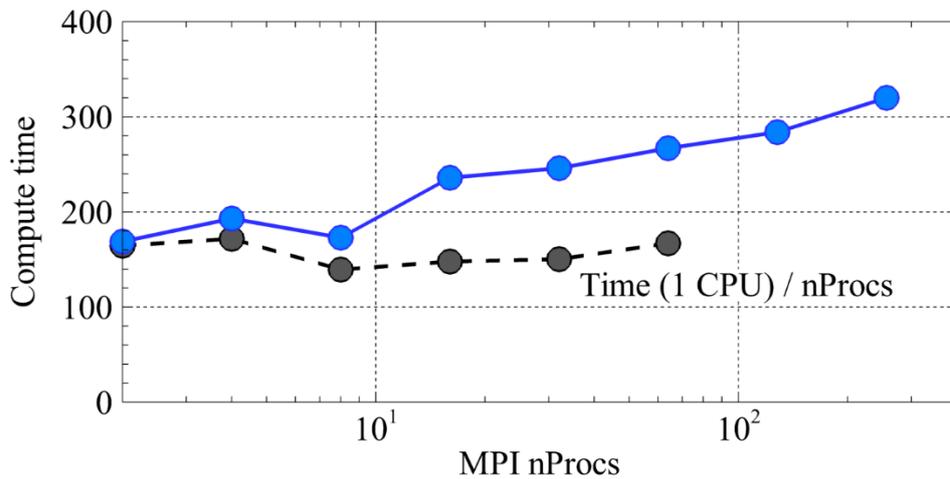
Time integration of a three-dimensional flow computed with between 2 and 256 Fourier modes distributed across the same number of processors. The compute time for this test case is denoted by the blue unbroken line, while for reference the average time per field from 1-CPU simulations are plotted with the black dashed line.

Dividing the compute times for the distributed cases by the corresponding times for the 1-CPU cases provides a measure of the speedup obtained by parallelisation. The figure below shows the measured speedups up to the available 64 processors for the aforementioned linearised solver and spectral element-Fourier 3D solver. Power law fits to the data demonstrate that the speedups scale with the -0.873rd and -0.862nd power of the number of processors, respectively. An ideal (linear) speedup would scale with the -1 power.
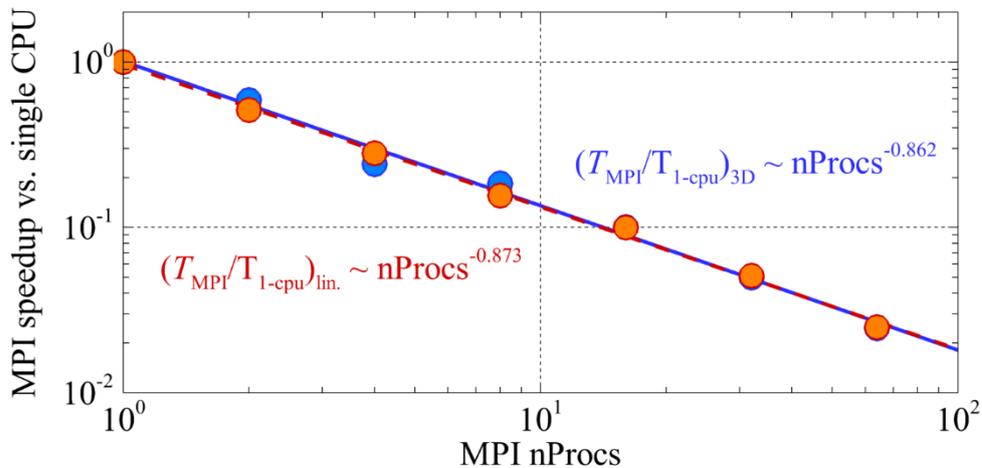
Speedup plotted against number of processors for the linearised (blue) and Fourier 3D (red) solvers described earlier. Power law fits to the data are shown on the figure.

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 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 MPI process. As with linear stability analysis calculations, poorer performance will result if the number of MPI processes was not a factor of 16.

Running Viper

While the Viper executable can run on a single processor directly, to run an MPI job across multiple processors, users must use the command "mpirun". For example, to execute Viper over 4 processors, the following command would be called:

```
mpirun -n 4 ../viper.x < macro.txt > output.txt
```

Here **mpirun** is used to distribute the program **viper.x** (which here is located in the parent directory (**../**) over 4 processors (specified with the option **-n 4**), with input taken from the text file **macro.txt**, and output being written to **output.txt**.

If the command **mpirun** is not recognized under your NCI login, you will need to load the relevant module. Users should add a statement similar to the following statement to their **.login** file in their home directory:

```
module load openmpi/1.8.8
```
(or)
```
module load openmpi/1.10.7-mlx
```

The exact modules necessary for using Viper are detailed in the Login Setup section of Chapter 5, based on the system on which they are running Viper. Contact either A/Prof. Gregory Sheard, or the appropriate system administrators, for assistance in which current openMPI modules should be loaded. Note that there is currently no Windows version of Viper available.

Getting the most out of Viper

The parallelization of Viper is implemented to scale up both the simulation of linearized perturbation fields (such as for linear stability analysis, Floquet stability analysis, and transient growth analysis), and the spectral element-Fourier 3D solver. In each case, perturbation fields or Fourier modes are distributed over the available processors.

Viper is designed to handle jobs where the number of fields or Fourier modes does not match the number of processors, but to get the most out of a parallel run, jobs would ideally have the same number of fields assigned to each processor (so each is doing a similar amount of work). Therefore, the number of fields should be set to be an integer multiple of the number of processors.

For linear stability analysis, the total number of fields is **1 + the number of perturbation fields** (don't forget the base flow!), while for spectral-element-Fourier 3D runs, the number of fields **is equal to the specified number of Fourier modes** (set using the "**-k**" option in the **fourier** command).

Be aware that there is a communication overhead in MPI jobs due to the time taken to communicate data between processors, which could be significant for high-resolution Viper runs with a large number of fields. Therefore, users may not necessarily find that the fastest execution time will occur when the maximum number of processors (i.e. equal to the number of fields) is employed. Users are encouraged to experiment with the number of processors for their specific jobs to determine the most efficient setup.

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.
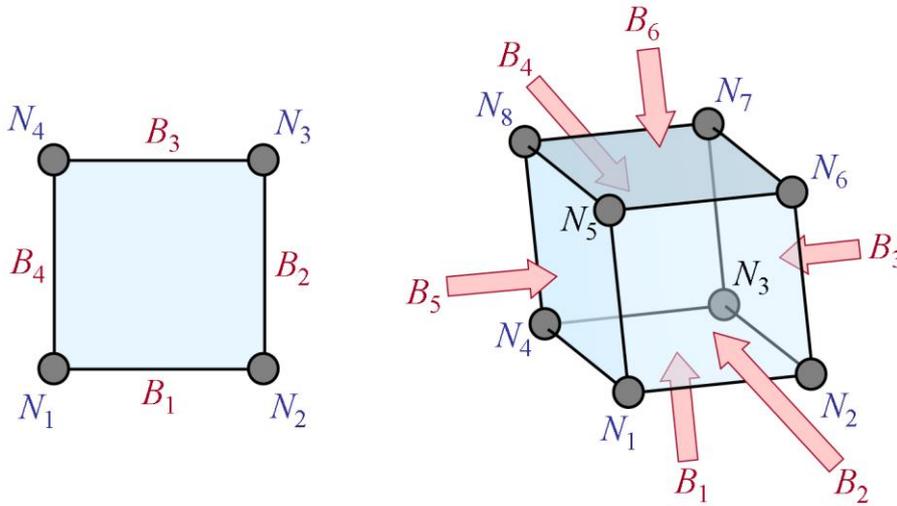
An example mesh file is shown below:

e.g.

```
441
 -5.00000000000000000E-01    4.00000000000000000E+00        1
 -4.95647999999999775E-01    4.00000000000000000E+00        2
 -4.88553000000000150E-01    4.00000000000000000E+00        3
 -4.77073999999999981E-01    4.00000000000000000E+00        4
 -4.58714000000000106E-01    4.00000000000000000E+00        5
 -4.29896000000000004E-01    4.00000000000000000E+00        6
.........
  4.88553000000000150E-01    5.00000000000000000E+00        439
  4.95647999999999775E-01    5.00000000000000000E+00        440
  5.00000000000000000E-01    5.00000000000000000E+00        441
400
1  1   2   23   22   3   0   0   4   1
2  2   3   24   23   3   0   0   0   1
........
397   416   417   438   437   0   0   1   0   1
398   417   418   439   438   0   0   1   0   1
399   418   419   440   439   0   0   1   0   1
400   419   420   441   440   0   2   1   0   1
```

The first row defines how many nodes there are, in this case 441. Each row thereafter defines the x and y coordinates of each node, and its corresponding number. After the location of each of the 441 nodes has been defined, then define how many elements there are, in this case 400. The next rows then define the information about each element (although the element number is now in the first column, rather than the last column as for nodes).

36

The first number (1) is the element number, the second number is the bottom left corner of the element specified by node number (1), then the bottom right node (2), then the top right node (23), then the top left node (22). This counter-clockwise order must always be used. The next four numbers are used to specify the **btag** values interpreted by the **viper.cfg** file. Hence, for the element boundary between the bottom left and right nodes (1 and 2), a **btag** number of 3 is assigned. Then nothing for the right side and top sides of the element (specified by a zero number), then a **btag** number of 4 is assigned to the left hand side of the element (between nodes 22 and 1). This goes on for each element, where the last element (400), has a bottom left corner of node (419), a bottom right corner of node (420), a top right corner node of (441) and a top left corner node of (440). It also has a **btag** of 2 assigned to the right side (between nodes 420 and 441), and a **btag** of 1 assigned to the top side (between nodes 441 and 440). The number in last column number is redundant (the 1 at the end of each row). However, it must still be specified (i.e. a 1 must be here for every element).

Note that regardless if the mesh is converted from a Gambit file, it is likely that the mesh file will contain CRLF line headers. These will need to be removed using **dos2unix <mesh_file>** (including extension) on the mesh file, when operating on a linux system.

Converting from Gambit

The Gambit mesh generation package can be used to generate meshes for use in Viper. Conversion utilities available from The Sheard Lab website (http://sheardlab.org/) 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 (where the queue script is).

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.

Note that most commands in which a floating point value was specified are now capable of accepting either previously user defined variables, or a mathematical function, of either global or user variable type. If a command still only accepts floating point values, please contact Dr. Gregory Sheard such that this can be updated to the new convention.

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, pressure, scalar and electric potential field boundary conditions on a boundary through the **<var>** string, which can be set to "**vel**", "**p**" "**s**" or "**e**" (case insensitive), for velocity, pressure, the scalar field (often temperature) and the electric potential field, respectively.

In addition, if **<var>** takes the value "**s**", then a scalar field will automatically be initialized, which will then be computed using a backwards-differentiation advection-diffusion scheme similar to that used for the velocity field. Users should also then specify the coefficient of scalar diffusion using **gvar_scalar_diff**.

Note that velocity boundary conditions can alternatively be specified per component, by setting **<var>** to **u**, **v**, or **w**. This is useful for prescribing exact stress-free boundary conditions on horizontal or vertical boundaries (by setting the normal velocity component to a zero Dirichlet condition, and the tangential component(s) to zero Neumann condition(s); a zero Neumann condition is the natural default if no explicit boundary condition is specified.

Note that a positive value specified for a Neumann boundary conditions refers to an outward normal vector.

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.

e.g. 3:

```
\> btag 5 u 1 '0.0'
```

Specifies that boundary number 5 will be prescribed a fixed Dirichlet velocity condition with $u = 0.0$ on the boundary.

e.g. 4:

```
\> btag 3 w 2 '0.0'
```

Specifies that the out-of-plane velocity component on boundary number 3 will be prescribed a Neumann velocity condition with a gradient value of 0.0.

e.g. 5:

```
\> btag 4 p 2 0.5
```

Specifies that boundary number 4 will be prescribed a Neumann condition with an outward normal gradient of the pressure field, having a value of $dp/dn = 0.5$ at the boundary.

e.g. 6:

```
\> btag 2 s 4
```

Specifies that boundary number 2 will be prescribed a periodic condition, whereby the values of the scalar field at this boundary will be equal to values at a corresponding boundary with identical element distribution in y (and z if 3D).

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_forcing_fu

Syntax: **gvar_forcing_fu <function>**
Function: **Add a forcing term to the u-velocity momentum equation.**
Description:

This command adds a forcing term of the form $+F$ to the u-component of the Navier-Stokes momentum equations. The forcing term F (**<function>**) may be a constant, a function of time only, a spatially varying steady-state function, or a time-dependent spatially varying function, and in each case may be expressed in terms of user-defined variables (**gvar_usrvar**) in addition to **t**, **x**, **y**, **z** and **RKV**.
Note: By default, the forcing term is zero.

e.g.:

```
\> gvar_forcing_fu 'x-y+t-3.47'
```

gvar_forcing_fv

Syntax: **gvar_forcing_fv <function>**
Function: **Add a forcing term to the v-velocity momentum equation.**
Description:
This command adds a forcing term of the form $+F$ to the v-component of the Navier-Stokes momentum equations. The forcing term F (**<function>**) may be a constant, a function of time only, a spatially varying steady-state function, or a time-dependent spatially varying function, and in each case may be expressed in terms of user-defined variables (**gvar_usrvar**) in addition to **t**, **x**, **y**, **z** and **RKV**.
Note: By default, the forcing term is zero.
e.g.:

```
\> gvar_forcing_fv 'x-y+t-3.47'
```

gvar_forcing_fw

Syntax: **gvar_forcing_fw <function>**
Function: **Add a forcing term to the w-velocity momentum equation.**
Description:
This command adds a forcing term of the form $+F$ to the w-component of the Navier-Stokes momentum equations. The forcing term F (**<function>**) may be a constant, a function of time only, a spatially varying steady-state function, or a time-dependent spatially varying function, and in each case may be expressed in terms of user-defined variables (**gvar_usrvar**) in addition to **t**, **x**, **y**, **z** and **RKV**.
Note: By default, the forcing term is zero.
e.g.:

```
\> gvar_forcing_fw 'x-y+t-3.47'
```

gvar_forcing_fs

Syntax: **gvar_forcing_fs <function>**
Function: **Add a forcing term to the scalar advection-diffusion equation.**
Description:
This command adds a forcing term of the form $+F$ to the u-component of the scalar advection-diffusion equation. The forcing term F (**<function>**) may be a constant, a function of time only, a spatially varying steady-state function, or a time-dependent spatially varying function, and in each case may be expressed in terms of user-defined variables (**gvar_usrvar**) in addition to **t**, **x**, **y**, **z** and **RKV**.
Note: By default, the forcing term is zero.
e.g.:

```
\> gvar_forcing_fs 'x-y+t-3.47'
```

gvar_forcing_gu

Syntax: **gvar_forcing_gu <function>**
Function: **Add a linear forcing term to the u-velocity momentum equation.**
Description:

This command adds a forcing term of the form $+Gu$ to the solver, where G is a function defined using this command, and u is the u-velocity component to the u-component of the Navier-Stokes momentum equations (e.g. $\frac{du}{dt} = \cdots + Gu$). The forcing term G (**<function>**) may be a constant, a function of time only, a spatially varying steady-state function, or a time-dependent spatially varying function, and in each case may be expressed in terms of user-defined variables (**gvar_usrvar**) in addition to **t**, **x**, **y**, **z** and **RKV**.

Note: By default, the forcing term is zero.
e.g.:

\> gvar_forcing_gu 'x-y+t-3.47'

gvar_forcing_gv

Syntax: **gvar_forcing_gv <function>**
Function: **Add a linear forcing term to the u-velocity momentum equation.**
Description:

This command adds a forcing term of the form $+Gv$ to the solver, where G is a function defined using this command, and v is the v-velocity component, to the v-component of the Navier-Stokes momentum equations (e.g. $\frac{dv}{dt} = \cdots + Gv$). The forcing term G (**<function>**) may be a constant, a function of time only, a spatially varying steady-state function, or a time-dependent spatially varying function, and in each case may be expressed in terms of user-defined variables (**gvar_usrvar**) in addition to **t**, **x**, **y**, **z** and **RKV**.

Note: By default, the forcing term is zero.
e.g.:

\> gvar_forcing_gv 'x-y+t-3.47'

gvar_forcing_gw

Syntax: **gvar_forcing_gw <function>**
Function: **Add a linear forcing term to the u-velocity momentum equation.**
Description:

This command adds a forcing term of the form $+Gw$ to the solver, where G is a function defined using this command, and w is the w-velocity component, to the w-component of the Navier-Stokes momentum equations (e.g. $\frac{dw}{dt} = \cdots + Gw$). The forcing term G (**<function>**) may be a constant, a function of time only, a spatially varying steady-state function, or a time-dependent spatially varying function, and in each case may be expressed in terms of user-defined variables (**gvar_usrvar**) in addition to **t**, **x**, **y**, **z** and **RKV**.

Note: By default, the forcing term is zero.

e.g.:

```
\> gvar_forcing_gw 'x-y+t-3.47'
```

gvar_forcing_gs

Syntax: **gvar_forcing_gs <function>**
Function: **Add a linear forcing term to the u-velocity momentum equation.**
Description:
This command adds a forcing term of the form $+Gs$ to the solver, where G is a function defined using this command, and s is the scalar variable, to the scalar advection-diffusion equation (e.g. $\frac{ds}{dt} = \cdots + Gs$). The forcing term G (**<function>**) may be a constant, a function of time only, a spatially varying steady-state function, or a time-dependent spatially varying function, and in each case may be expressed in terms of user-defined variables (**gvar_usrvar**) in addition to **t**, **x**, **y**, **z** and **RKV**.
Note: By default, the forcing term is zero.
e.g.:

```
\> gvar_forcing_gs 'x-y+t-3.47'
```

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 a rounded curve or around a sharp corner. An example of this is the sharp trailing edge of an aerofoil.

gvar_mhd_coeff

Syntax: **`gvar_mhd_coeff <coeff>`**

Function: **Sets the prefactor for the quasi-static MHD term in the momentum equation.**

Description

The coefficient **`<coeff>`** must be a real non-negative value. For more information on the quasi-static solver, refer to Chaper 2: Quasi-Static MHD, or the command **`mhd`**.

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 previously defined 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 *N* (*p*-resolution).**

Description:

The element polynomial degree *N* is set to an integer **`<value>`**, where **`<value>`** must be equal to, or greater than, $N = 2$. The default value is $N = 4$. In 2D and 3D, the number of nodes per element is N^2 and N^3, respectively. 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. However, discontinuities will not necessarily be more accurately modelled by higher polynomial orders. Furthermore, very high polynomial orders (say greater than 20) may cause issues with modelling the advection operator, which is effectively of order $N(N - 1)$. Increasing the spatial resolution, or the use of a controlling factor, such as spectral vanishing viscosity (see **`svv`**) may be needed if divergence in simulations is noticed at very high polynomial orders.

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. The default value is 10.0.

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_scalar_uvel_forcing

Syntax: **`gvar_scalar_uvel_forcing <coeff>`**

Function: **Sets a scalar forcing multiplied by the u-velocity.**

Description:

This command adds a forcing term to the scalar advection- diffusion equation, multiplied by the x-direction (Cartesian) / axial z-direction (cylindrical) velocity component. The forcing term takes the form $-$**`<coeff>`** $* u$, which is appended to the scalar advection- diffusion equation (e.g. $\frac{ds}{dt} = \cdots -$**`<coeff>`** $* u$). The real parameter **`<coeff>`** (which can be specified as a function of user-defined variables) is the

prefactor of this term. Its sign is such that a positive (left-to-right) u-velocity and a positive-valued coefficient will lead to a decrease in the scalar field.
Note: This function is currently only implemented in 2D.

gvar_usrvar

Syntax: **`gvar_usrvar <var_name> <var_function>`**
Function: **Defines a user-defined variable as a function.**
Description:

The function is a mathematical expression, which can be a function of time **`t`**, spatial coordinates **`x`**, **`y`**, **`z`**, the reciprocal kinematic viscosity **`RKV`**, plus any previously created user-defined variables. A character string is required for each of the **`<var_name>`** and **`<var_function>`** parameters. **`<var_name>`** is the name of the new variable, which cannot be the same as an existing or previously defined user-specified variable, and **`<var_function>`** is a string specifying the function evaluated when **`<var_name>`** appears in subsequent functions, that appear in either **`viper.cfg`** or **`macro.txt`** files.

mesh_file

Syntax: **`mesh_file <filename>`**
Function: **Defines the macro-element distribution.**
Description:

The **`mesh_file`** contains the node coordinates and macro-element information, before the interpolating polynomials are applied. It should be a data file (.msh) which is commonly converted from an **`.FDNEUT`** file. The macro-elements can be observed using tecplot by calling **`tecp`** with a polynomial order of $N = 2$. Any other polynomial order will then display the locations of the interpolating nodes. The format of the mesh file is described in Chapter 3, Accepted Mesh Formats, noting that all nodes and elements should be unique.

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.

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.x < macro
```

launches the Viper executable **viper.x**, 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	step 100
loop 3	save -s -f save.dat
macro macro3.txt	tecp -s -f tecplot.plt
endl	flowrate
stop	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):

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):

autocorrf	**nu_horiz_2d**
arnoldi	**nu_xsect_2d**
avg_one_dir	**pert_ke_evol**
current	**reconload**
energies	**reconstore**
energyf	**sample**
flowrate	**samplef**
flux	**save**
forceflow	**stab**
forces	**svd**
get_min_max	**tecp**
int	**tg**
intf	**time_avg**
L2	**tony_psi**
line	**track**
moments	**womersley**

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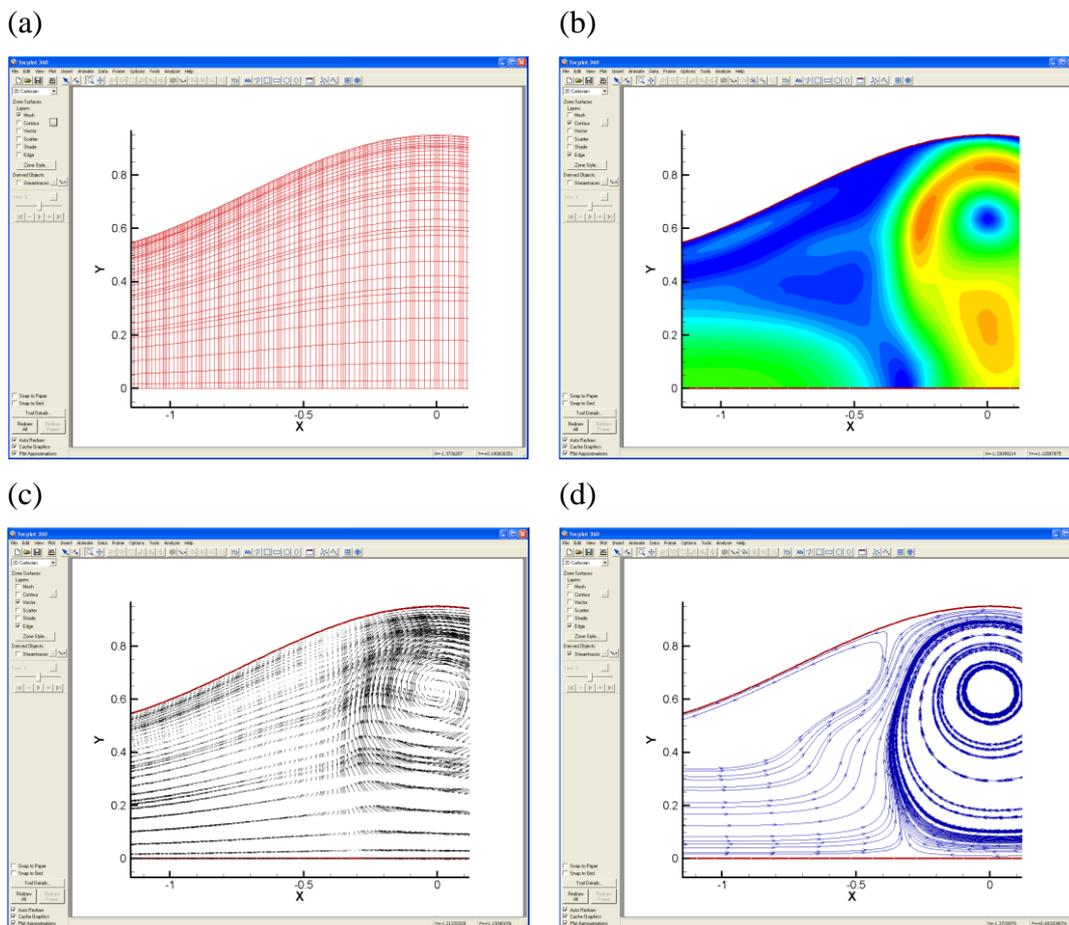
Visualization of a portion of the computational domain of a two-dimensional simulation. (a) The mesh, (b) flooded contours of instantaneous velocity magnitude, (c) velocity vectors, and (d) velocity streamlines.

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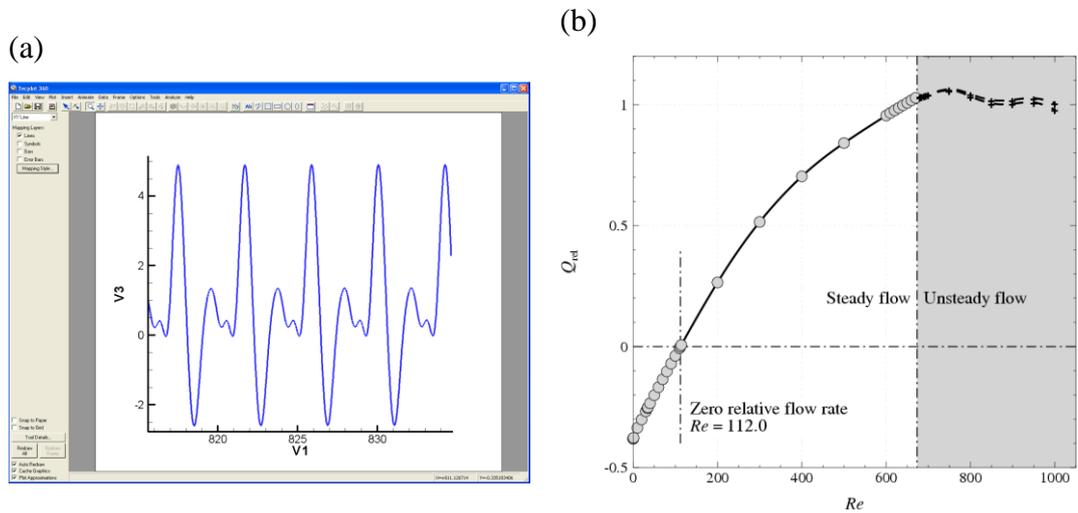
Graphing data with Tecplot: (a) A screenshot showing a time-dependent data set loaded into Tecplot with default plotting options. (b) A plot from Sheard & Ryan (2007) generated using Tecplot.

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
```

The full list of commands are provided below, 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.

Note that anywhere a floating point value could be specified in a command, a mathematical function or previously user defined variables should also be able to be input (so long as it evaluates to a floating point value). If a command still only accepts floating point values (and provides an error if a variable is provided instead), please contact Dr. Gregory Sheard such that this can be updated to the new convention.

Advect

Syntax: **`advect <option>`**
Function: **Toggle advection substep on/off during time integration.**
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 currently implements only the convective form of the advection term. The **`advect`** command can 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. The default setting of this feature is ON.

Toggling is performed as followed:

    `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>`** $\geq 2 +$ **`<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 in **`<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 Fu, the autocorrelation s calculated first by taking the product of Fu 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`**.

Avg_one_dir

Syntax: **`avg_one_dir [-k <field> -dir <dir> -var <var>]`**
Function: **Average a specified field along a specified direction.**
Description:
This routine averages a single variable, along a single specified direction over the field given by option "**`-k`**". This routine works best if the mesh has mesh lines parallel to the direction to be used for averaging.
The following options are available:

> **`-k <field>`**
> > Used to specify an integer perturbation field number. A legitimate value must be specified, hence the field number must be less than or equal to the number of floquet modes present (**`k <= Nfloq_modes`**). To average all fields, set "-k" value to a negative number.

> **`-dir <dir>`**
> > Used to specify a single direction to average along, either x or y (e.g. to average along the x direction: **`-dir x`**), but not z (see the comment below).

> **`-var <var>`**
> > Used to specify a single variable to average along, either one of the velocity components, u, v, or w, or the scalar field, s. (e.g. for the scalar field: **`-var s`**).

This function requires the solution to be initialised, and cannot be used unless the simulation is two-dimensional. It cannot be invoked in a spectral-element-Fourier 3D simulation.

See also: `init`.

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`.

Axirotate

Syntax: `axirotate <omega>`

Function: **Computes cylindrical coordinates computations in a frame rotating about the axis.**

Description:

This command activates extra terms required to compute flows in a rotating reference frame if using cylindrical coordinates (i.e. z-r-θ, see `axi`). These include Coriolis corrections to the base flow and perturbation fields, and centripetal corrections to the base flow. The user must supply an angular velocity for the rotating frame, `<omega>`.

Notes:

A positive-signed `<omega>` equates to rotation out of the page above the symmetry axis.

This feature only makes sense when swirling flow (`wvel`) is activated. Therefore calling this command also activates `wvel`. All boundary conditions must be expressed relative to the rotating frame. i.e., if a wall is rotating with the flow, it should be expressed as a Dirichlet velocity boundary with zero velocity (i.e. no movement relative to the rotating frame).

This function requires the simulation be two-dimensional, but will activate `axi` to toggle to the appropriate co-ordinate system for any two-dimensional simulation.

See also: **axi**, **wvel**.

Buoyancy

Syntax: **buoyancy [-g <gx> <gy> -a <exp_coeff> -c <froude>]**

Function: **Activate Boussinesq buoyancy term.**

Description:

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. Use the **-g** option to supply x- and y-components of a gravity vector. These will be automatically rescaled into a unit vector. The momentum equations are modified by adding a gravity term:

$$-\rho' \boldsymbol{g}$$

where $\boldsymbol{g}$ is a unit vector in the direction of gravity, the direction of which is set using the **-g** option. The **-g** option takes vector components in x and y (specified with **<gx>** and **<gy>**, respectively), which are rescaled by Viper into a unit vector. A z component cannot be specified, and is zero by default. ρ' is the fluctuating component of density due to temperature variations, and is expressed as:

$$\rho' =< \textbf{exp\_coeff} >* S$$

The coefficient in front of the temperature, S, i.e. **<exp_coeff>**, is supplied by the user with the **-a** option, which may be expressed as a function including user-specified variables/functions. The direction of buoyancy is such that a positive **<exp_coeff>** will cause colder fluid to fall in the direction of gravity, and hotter fluid to rise opposite to the direction of gravity. 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.

The **-c** option specifies a Froude number (which represents the ratio of inertia to gravity, e.g. $Fr = L\omega^2/g$ or $Fr = V^2/Lg$). If this option is specified, this Froude number is multiplied by the **-a** coefficient, and the resulting coefficient is used to modify the advection terms to account for centrifugal buoyancy effects in the flow. To consider only centrifugal buoyancy effects, set the gravity vector to zero.

Examples of use of the **buoyancy** command:

HOT (high-S) fluid will RISE (left, negative x); buoyancy coefficient is 100.0:
> **buoyancy -g 1.0 0.0 -a 100.0**

HOT (high-S) fluid will RISE (upward, positive y); buoyancy coefficient is 30.0:
> **buoyancy -g 0.0 -2.3 -a '3*10'**

HOT (high-S) fluid will move radially inward due to centrifugal effect:
> **buoyancy -g 0.0 0.0 -a '3*10' -c 100**

HOT (high-S) fluid will move radially inward and rise due to centrifugal and gravity effects
> **buoyancy -g 1.0 0.0 -a '3*10' -c 100**

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.).

The centrifugal buoyancy term will only be activated if a positive Froude number is specified.

Current

Syntax: `current [-f <filename> -k <field>]`

Function: **Reports on divergence of electric current in quasi-steady MHD simulations.**

Description:

Calculates the divergence of the electric current field in a quasi-static MHD simulation, and outputs to a text file the integral of the square of the divergence over the computational domain. These calculations can only be performed if the solution has been initialised.

Notes:

For spectral element-Fourier domains, the integral is evaluated over the domain volume. For Cartesian 2D base flows and perturbaion fields having zero spanwise wavenumber, the result is computed on the 2D plane (i.e. a value expressed per unit span). For axisymmetric base flows and perturbation fields having zero azimuthal wavenumber, the integral result is calculated over the full 2*pi radian azimuthal domain size. For linearised perturbation fields having non-zero wavenumber, the integral is calculated using the corresponding azimuthal/spanwise span of the domain.

The following options are available:

`-f <filename>`

Used to specify a filename **`<filename>`** (including extension) to write the computed integral to. If omitted, the default filename is **`int_sqr_div_and_mag_current.dat`**.

`-k <field>`

Used to specify an integer perturbation field number, when a linearised perturbation field is active, to calculate the integral on. A legitimate value must be specified, hence the field number must be less than or equal to the number of floquet modes present (**`k <= Nfloq_modes`**). The default is $k = 0$, corresponding to the base flow.

Note: The `current` requires the quasi-static MHD solver be active. To activate the electric potential field, define the appropriate electric potential field boundaries in the `viper.cfg` file (see `btag`).

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`**.

Energies

Syntax: **`energies [-f <filename>]`**

Function: **Output volume-integrated kinetic and potential energies.**

Description:

In a simulation with buoyancy, this command computes and outputs to a file the total kinetic energy, the total vertical buoyancy flux, and the background and available potential energies. This routine can be used in an SE-Fourier 3D simulation, but the results will be the total energies calculated on only the fundamental (zero-wavenumber) mode of the 3D solution. These computations can only be performed if the solution has been initialized

The following options are available:

> **`-f <filename>`**
>
>> Used to specify a filename **`<filename>`** (including extension) to write the computed integral to. If omitted, the default filename is **`energies_2d_plane.dat`**.

Note: The `energies` command can only be employed in 2D or SE-Fourier 3D simulations. It must also have a temperature field, buoyancy activated, and a non-zero gravity vector.

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

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

$$\oint_\Omega \|\hat{\boldsymbol{u}}_k\|^2 \, d\Omega \,,$$

where the kth Fourier mode coefficients of the velocity field are given by $\hat{\boldsymbol{u}}_k$, and Ω is the computational domain in the spectral-element plane (either *x-y* or *z-r*). This computation can only be performed if the solution has been initialised.

The following options are available:

> **`-f <filename>`**
>
>> Used to specify a filename **`<filename>`** (including extension) to write the computed integral to. If omitted, the default filename is **`energyf.dat`**.

See also: **autocorrf**, **energies,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**.

Filt_s_adv

Syntax: **filt_s_adv [-u <function>]**
Function: **Activate a filter for the advection of a scalar field.**
Description:
Used to establish a filter kernel that multiplies the RHS of the scalar advection calculation. When **<function>** evaluates to 0, no advection occurs, when **<function>** evaluates to 1, the advection is unchanged. The following options are available:

> **-u <function>**
>> Used to specify a filename **<function>** of x, y, or t (this is currently only implemented in two-dimensional simulations, but is evaluated at each timestep). The default is **<function>** $= 1$.

Flowrate

Syntax: **flowrate [<filename>]**
Function: **Output flow rate through each boundary.**
Description:
The flow rates through each boundary are calculated. This calculation can only be performed if the solution has been initialised.
The following options are available:

> **<filename>**
>> Used to specify a filename **<filename>** (including extension) to write the flowrate through each boundary to. If omitted, the default filename is **flowrate.dat**.

Fixscalar

Syntax: **fixscalar [<value>]**
Function: **Constrain a domain integral of the scalar field to a specified value.**
Description:
Adds a constant correction during every time step to shift the domain integral of the scalar field to a value specified by **<value>**. An example of where this is useful is when the scalar field represents temperature, and only flux boundary conditions are imposed, leading to a perpetual heating or cooling in the enclosure. Subsequent calls to **fixscalar** toggle it on or off.
The following options are available:

`<value>`

> Used to specify a filename **`<value>`** to define the integral of the scalar field. If omitted a default of **`<value>`** = 0 is used.

Flux

Syntax: **`flux [-s] [-vel] [-f <filename>]`**
Function: **Output the flux through each boundary.**
Description:
The flux through each boundary is integrated from the component of the dot product of the gradient of the specified field and the outward normal vector along each boundary. This command also calculates the integrated absolute value of the flux through each boundary. These are output in a second set of data columns after the flux data columns. The following options are available:

`-s`

> Used to request that the output flux is that of the scalar field. This option cannot be requested while **`-vel`** is.

`-vel`

> Used to request that the output flux is that of the velocity field. Currently this is not yet implemented, please contact Dr. Gregory Sheard if you would like this feature to be implemented. This option cannot be requested while **`-s`** is.

`-f <filename>`

> Used to specify a filename **`<filename>`** (including extension) to write the flowrate through each boundary to. If omitted, the default filename is **`flux_scalar.dat`**.

See also: **`flowrate`**.

Forceflow

Syntax: **`forceflow [-b <bnum> -q <flowrate> -o <filename>]`**
Function: **Specify a flowrate to be achieved through a specific boundary.**
Description:
This command is used to specify a volume flowrate (per unit span) in a 2D simulation that is to be imposed through a specified boundary. This routine currently only works in 2D simulations, and is hard-coded to manipulate the u-velocity only. Hence, it finds application in duct or channel flow problems, typically with periodic boundaries, as an alternative to driving the flow with a constant forcing equivalent to an imposed horizontal pressure gradient. In those cases time-varying flow features can produce a time-dependent flow rate, making control of the Reynolds number more difficult. This routine works by checking the flow rate through the requested boundary each time step, and applying whatever forcing is required to the horizontal (u-) velocity field to preserve the target flow rate.

This command may be called before or after **`init`**, and multiple calls are permitted (e.g. for facilitating abrupt changes in flowrate, or switching off the flowrate forcing mid-simulation). The option **`-o`** is used to specify an output filename to output the current applied forcing. If the **`-o`** option is called, parameters supplied with the **`-b`** and **`-q`** options are ignored, and provided the solution is already initialized, the routine outputs to a text file named **`<filename>`** the most recently applied forcing.

Hence a typical usage of this command is to:

1) Call **forceflow** with **-b** and **-q** options specified (either before or after **init**.

2) Repeatedly call **forceflow** with only the **-o** option specified to output a forcing time history. The simulation must be initialised to use the **-o** option.

The following options are available:

-o <filename>

> Used to specify a filename **<filename>** (including extension) to write the flowrate through each boundary to. If omitted, the default filename is **forceflow_forcing.dat**.

-b

> Used to specify a boundary through which the desired flowrate should be kept constant (which should intersect the x-axis). The **-b** value must be a positive integer between 1 and the maximum number of boundaries.

-q

> Used to specify the desired flowrate. A function of user defined variables can be specified.

See also: **flowrate**.

Forces

Syntax: **forces <boundary> [<filename>]**

Function: **Calculate global forces imparted on a specific boundary.**

Description:

Calculates the global forces (pressure, viscous and total), in x, y (and z) imparted on a single boundary. If no boundary number is specified, or if the simulation has not been initialised, no calculations or output is performed.

The following options are available:

<boundary>

> Used to specify a single boundary **<boundary>** that has been defined in the **viper.cfg** file. If multiple calculations on different boundaries are needed, a separate call to forces must be used for each. They will all be appended to the same file unless a different optional **<filename>** specifier is used for each.

<filename>

> Used to specify a filename **<filename>** (including extension) to write the forcing acting on the boundary to, which will be appended at each timestep, or with each call to forces that uses the same **<filename>**. If omitted, the default filename is **forces.dat**.

Fourier

Syntax: **fourier [-f <filter_dist> -n <Nplanes> -k <Nmodes> -mode <mode_number> -span <span> -alias]**

Function: **Initialise a spectral-element/Fourier 3D 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.

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, no filter is applied, but if required, a ramp filter can be extablished varying from 100% at $r = 0$ to 0% at $r = $ **<filter_dist>**.
>
> **-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, hence **<Nplanes>** must be at least 2. A default of 4 planes is used. Alternately using the **−k** option may be simpler.
>>
>> Note: For best efficiency from parallel simulations, computations should be run on **<Nplanes>**/2 + 1 MPI processes, 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 MPI process.
>
> **-k <Nmodes>**
>
>> A positive integer is supplied to specify the number of non-zero Fourier modes employed in the computations. This option can be used in place of the clunky **−n** option. The number of Fourier modes should be a whole factor of the number of MPI processes available in a simulation. i.e., If **<Nmodes>**> = 16, this could be efficiently distributed over 1, 2, 4, 8 or 16 MPI processes in a parallel computation. If the **−alias** option is also used, there will be 50% additional modes used to compute the advection term, and if it is not used, then one extra mode is employed for advection (this provides a minimal level of antialiasing by default).
>
> **-mode <mode_number>**
>
>> The positive floating point value supplied as **<mode_number>** 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>** = $2\pi/$**<mode_number>**. If no span or mode number is supplied, the computation defaults to a span of 2π, corresponding to an out-of-plane mode number $m = 1$. If both are given, the computation will employ the most recently given value.
>
> **-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.

-alias

Apply two-thirds rule for anti-aliasing in Fourier space. Note that this will   increase the compute time as 50% more Fourier modes are included in calculations of the advection term.

Note: The `init` command must be called after `fourier`, to prepare for time integration. Furthermore, you cannot call `fourier` if Floquet analysis is active (do not call `pert` if using `fourier`). Furthermore, 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.

See also: **axi,pert,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**, **pert,rotate**

Getminmax

Syntax: **getminmax [-f <filename> -k <field> -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 ($1/\nu$),

dudx, **dudy**, **dudz**, **dvdx**, **dvdy**, **dvdz**, **dwdx**, **dwdy**, **dwdz** (spatial velocity gradients $\frac{du}{dx}, \frac{du}{dy}, \frac{du}{dz}, \frac{dv}{dx}, \frac{dv}{dy}, \frac{dv}{dz}, \frac{dw}{dx}, \frac{dw}{dy}, \frac{dw}{dz}$),

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**.

-k <field>

Used to specify an integer perturbation field number (i.e., 1, 2, ... , **<Nfloq_modes>**, when Floquet analysis is active) to search for maxima/minima. The default is **<field>** $= 0$, corresponding to the base flow.

-p <function>

A user-specified function **<function>** is provided to the routine. If omitted, the default is vorticity in the *x-y* plane ($\frac{dv}{dx} - \frac{du}{dy}$): "**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 $|curvature|^2 = 0.0$.

-x <level> <tol>

A cutoff 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.

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. This routine will also initialise particle tracking if required.

Int

Syntax: **int [-f <filename> -k <field> -u <function>]**
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 $\frac{du}{dx}, \frac{du}{dy}, \frac{du}{dz}, \frac{dv}{dx}, \frac{dv}{dy}, \frac{dv}{dz}, \frac{dw}{dx}, \frac{dw}{dy}, \frac{dw}{dz}$),

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>**. The solution must be initialized for the integral to be computed.

Note that for 3D hexahedral spectral element and spectral element-Fourier domains, the integral is evaluated over the domain volume. For Cartesian 2D base flows and perturbation fields having zero spanwise wavenumber, the integral result is computed on the 2D plane (i.e. a value expressed per unit span). For axisymmetric base flows and perturbation fields having zero azimuthal wavenumber, the integral result is calculated over the full 2*pi radian azimuthal domain size. For linearised perturbation fields having non-zero wavenumber, the integral is calculated using the corresponding azimuthal/spanwise span of the domain.

The following options are available:

> **-k <field>**
>
> > Used to specify an integer perturbation field number (i.e., 1, 2, ... , **Nfloq_modes**, when Floquet analysis is active) to calculate the

integral on. The default is **`<field>`** $= 0$, corresponding to the base flow.

    `-f <filename>`

        Used to specify a filename **`<filename>`** (including extension) to write the computed integral to. If omitted, the default filename is **`integral.dat`**.

    `-u <function>`

        Used to specify the function to be integrated. The default is **`<function>`** $= 0$.

See also: **`intf,l2`**.

Intf

Syntax: **`intf [-f <filename> -u <function>]`**

Function: **Integrates a user-specified function over Fourier modes in an SE-F 3D computation.**

Description:

A user-specified function **`<function>`** is input (using the same mathematical functions available during configuration), and the value of this function is integrated separately on each mode of a spectral element-Fourier 3D computation. The output is similar to that of the **`<energyf>`** command. 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 $\frac{du}{dx}, \frac{du}{dy}, \frac{du}{dz}, \frac{dv}{dx}, \frac{dv}{dy}, \frac{dv}{dz}, \frac{dw}{dx}, \frac{dw}{dy}, \frac{dw}{dz}$),

    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>`**. The solution must be initialized for the integral to be computed.

Note that for 3D hexahedral spectral element and spectral element-Fourier domains, the integral is evaluated over the domain volume. For Cartesian 2D base flows and perturbation fields having zero spanwise wavenumber, the integral result is computed on the 2D plane (i.e. a value expressed per unit span). For axisymmetric base flows and perturbation fields having zero azimuthal wavenumber, the integral result is calculated over the full 2*pi radian azimuthal domain size. For linearised perturbation fields having non-zero wavenumber, the integral is calculated using the corresponding azimuthal/spanwise span of the domain.

The following options are available:

    `-f <filename>`

        Used to specify a filename **`<filename>`** (including extension) to write the computed integral to. If omitted, the default filename is **`integralf.dat`**.

    `-u <function>`

        Used to specify the function to be integrated. The default is **`<function>`** $= 0$.

See also: **`int,l2`**.

Iterate

Syntax: `iterate [-tol <tol> -n <max_its>]`
Function: **Set up multiple iterations of base flow solution within each time step.**
Description:

By default, only one cycle through advection/pressure/diffusion is conducted each time step. However, in theory an improvement in accuracy (and possibly stability might be gained if the solution was iterated to improve the accuracy of the extrapolated estimate of the solution projected to the next time step which is required for the calculation of the non-linear (advection) term. To invoke this iteration procedure, this command must be called.

The following options are available:

 -tol <tol>

  Used to specify the convergence threshold for the simulations. The quantity monitored for convergence is the average change in the solution vectors each iteration. If **iterate** is and this option is not specified, the default is 1e-20. If a negative threshold is specified, iteration will always proceed until **<max_its>** has been reached.

 -n <max_its>

  Used to specify the maximum number of iterations performed per time step. If **iterate** is invoked and this option is not specified, the default is 5. If a **<max_its>** value of less than 1 is specified, the **<max_its>** value will be adjusted to 1.

See also: **step**.

L2

Syntax: `L2 [-f <filename> -k <field>]`
Function: **Compute the *L2* norm (integral of velocity magnitude throughout domain).**
Description:

An *L2* norm is computed by integrating the square of the magnitude of velocity in physical space, over the entire computational domain, consistent with the definition of Barkley, Blackburn & Sherwin (Int. J. Numer. Meth. Fluids 2008; 57:1435-1458). The integrand is thus defined as (for a three-dimensional computation):

$$\|\boldsymbol{u}\|^2 = u^2 + v^2 + w^2$$

Note that for 3D hexahedral spectral element and spectral element-Fourier domains, the integral is evaluated over the domain volume. For Cartesian 2D base flows and perturbaion fields having zero spanwise wavenumber, the integral result is computed on the 2D plane (i.e. a value expressed per unit span). For axisymmetric base flows and perturbation fields having zero azimuthal wavenumber, the integral result is calculated over the full 2π radian azimuthal domain size. For linearised perturbation fields having non-zero wavenumber, the integral is calculated using the corresponding azimuthal/spanwise span of the domain. The solution must be initialised for any computation to be performed.

The following options are available:

 -f <filename>

Used to specify a filename **`<filename>`** (including extension) to write the L_2 norm to. If omitted, the default filename is l2norm**`.dat`**.

`-k <field>`

Used to specify an integer perturbation field number (i.e., 1, 2, ... , **`Nfloq_modes`**, when Floquet analysis is active) to search for maxima/minima. The default is **`<field>`** = 0, corresponding to the base flow.

See also: **`int,intf`**.

Line

Syntax: **`line –p1 <x1> <y1> -p2 <x2> <y2> [-f <filename> -n <points> -u <fn_str> -avg]`**

Function: **Extract flow field data along a line between specified points.**

Description:

The line command extracts flow field values, or gradients along a line between two specified points, once the solution has been initialised. The **`line`** command can typically only be used in 2D simulations (as the line is specified on the 2D spectral-element plane). The exception is that if the **`–avg`** option is specified, **`line`** can also be used in spectral element-Fourier 3D simulations, where the average along the line and into the page is evaluated. The averaged value in SE-Fourier 3D simulations is evaluated using values from the fundamental Fourier mode only. It therefore only works for linear functions of the flow field variables (when using the **`-u`** option), i.e. specifying "**`line –u 'u+v' -avg`**" is okay, but " **`line –u 'u^2+v^2' -avg`** " is not.

The following options are available:

`–p1 <x1> <y1>`

Specifies the start point of the line for data extraction. This parameter must be specified for this command to function.

`–p2 <x2> <y2>`

Specifies the end point of the line for data extraction. This parameter must be specified for this command to function.

`-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 **`line.dat`** is used.

`-n <points>`

Used to specify the number of points (**`<points>`**) along the line at which the data is interpolated. If the **`-n`** option is not specified, the default is 10 points.

`-u <fn_str>`

Used to supply a user-specified function of variables **`t`**, **`x`**, **`y`**, **`u`**, **`v`**, **`w`**, scalar field (if active) and their gradients. If this option is used, only the value of this function is output for each point on the line, rather than all flow variables/gradients. Hence, this can be useful for reducing the output file size, if you want the value of only a single variable.

E.g. If the user wishes to interpolate the v-velocity only along a line, they could call:

```
\> line –u 'v'
```

-avg

> If this option is specified, only the average of the interpolated values along the line is output instead of at each interpolated point. This is useful for calculating and quickly outputting average values along boundaries or across cuts through the domain.

Load

Syntax: **load [-a <scale_factor> -f <filename> -k <floq_mode> -m]**
Function: **Load flow field vectors from file.**
Description:

Loads flow field vectors, as well as computation time **t**, from a user-specified file. This command loads flow field data from files created with the command **save**, and is used to resume a computation from a previously computed solution.

Note that load can be used after the solution has been initialised. Although uncommon, this allows for the user to replace static Dirichlet boundary conditions with whatever the velocities were in the restart file.

Update 03/06/2013: **load** no longer over-writes the **RKV** parameter value, or the time step **dt** value, with the values stored in the restart file. These must be set in the **viper.cfg** configuration file.

Update 12/05/2008: This command now no longer recognises the pre-04/11/2006 file format. For Spectral-element/Fourier simulations, either 2D or 3D SE/Fourier data may be input.

Update 04/11/2006: 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 restart.

The following options are available:

-a <scale_factor>

> Used to specify a scaling factor to apply to the field being loaded (default = 1.0). This is most useful when loading perturbation fields onto Fourier modes of a spectral element-Fourier 3D simulation.

-k <floq_mode>

> Used to specify an integer perturbation field number to load the saved file into. For linear stability analysis, perturbation fields range over (i.e., 1, 2, ... , **Nfloq_modes**). The default is **<floq_mode>** = 0, corresponding to the base flow. For SE-Fourier 3D jobs, individual Fourier modes are numbered 0 (fundamental 2D mode), and [1, 2, ... , $N_{\text{fourier\_planes}}/2$].

-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.

-r

Used to control whether the loaded field replaces whatever is in the flow field vectors (the default behaviour), or if the loaded field is added to the base flow (the behaviour if the option is used).

Note: The `init` command must be called before the use of the `-k load` option when loading saved perturbation fields.

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**.

Lsa

Syntax: **lsa [-prefix <string> -nev <integer> -ncv <integer> -tol <integer> -Nsteps <integer> -adjoint]**
Function: **Find leading eigenmodes of a linear time integration operator.**
Note: This is a driver routine. It automatically executes a loop, calls the Arnoldi command, and conducts the required time integration. The solution must have been initialised (`init`) and perturbation fields must be active (`pert`).
Description:

Linear stability analysis is used to find the amplification factors (Floquet multipliers) and corresponding perturbation fields (the eigenvalues and eigenvectors, respectively) of the operator describing the evolution of the perturbation field over a time interval, T. The following options are available:

-prefix <string>

Used to specify a filename prefix for eventual output of the LSA solver. The default is **lsa_**.

-nev <integer>

Used to specify the number of leading eigenmodes to be found by the Arnoldi solver. The default value (and minimum allowable value) is 1.

-ncv <integer>

Used to specify the length of the Arnoldi factorization used by the Arnoldi solver. The default value is 6, and the minimum allowable value is (**nev**+2)].

-tol <integer>

Used to specify the exponent of the convergence criterion used for the Arnoldi solver (i.e. 10^**`<integer>`**). The default value is -7, corresponding to 10^{-7}.

`-Nsteps <integer>`

Used to specify the number of time steps per Arnoldi iteration update. The time interval τ is calculated as $dt * N_{steps}$. The default value is 1000, and the minimum allowable value is 1.

`-adjoint`

If specified, the adjoint of the linearised equations will be integrated backwards in time rather than the default forward time integration of the linearised equations. The resulting computation is thus no longer a linear stability analysis, per se. Instead it becomes a calculation of the eigenmodes of the adjoint of the linearised evolution operator, which is useful in sensitivity analysis, etc.

See also: **`init,pert`**.

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 [-u <u_fn> -v <v_fn> -w <w_fn> -s <s_fn> -k <field>]`**
Function: **Applies a user-defined mask function to a specified 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, or a scalar field. Each field is applied a mask with a separate function, as:

$$field_{masked} = < \textbf{function} > * field_{original} .$$

If no mask function is specified, the default mask is 1.0 (no change to the field). Furthermore the mask will only be applied if the solution has been initialised.
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, e.g.

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

The following options are available:

-u \<u_fn>
> Used to specify a mask function for the u velocity field. The function can use intrinsic and user-specified variables, such as **x**, **y**, **t**, **RKV**, etc

-v \<v_fn>
> Used to specify a mask function for the v velocity field. The function can use intrinsic and user-specified variables, such as **x**, **y**, **t**, **RKV**, etc.

-w \<w_fn>
> Used to specify a mask function for the w velocity field. The function can use intrinsic and user-specified variables, such as **x**, **y**, **t**, **RKV**, etc.

-s \<s_fn>
> Used to specify a mask function for the scalar field. The function can use intrinsic and user-specified variables, such as **x**, **y**, **t**, **RKV**, etc.

-k \<field>
> Used to specify field the mask is applied to. By default, the mask is applied to the base flow (**k** = 0). Linearized perturbation fields are referenced using numbers 1, 2, 3, etc. To mask all fields, set the **-k** parameter to a negative value.

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.

Mhd

Syntax: **mhd coeff \<value>**
Function: **Used to invoke functions relating to the quasi-static MHD solver.**
Description:

Viper facilitates magnetohydrodynamic (MHD) simulations based on the quasi-static approximation. The quasi-static approximation is asymptotically exact for flows with magnetic Reynolds number $Re_m \ll 1$, and achieves high accuracy for $Re_m < O(1)$. Under the quasi-static model, the momentum equation is augmented by an additional term

$$N(-\nabla\phi + \boldsymbol{u} \times \boldsymbol{e_B}) \times \boldsymbol{e_B},$$

where N is an MHD prefactor, ϕ is the electric potential field, $\boldsymbol{u}$ is the velocity vector field, $\times$ denotes a vector cross product, and $\boldsymbol{e_B}$ is a unit vector in the direction of the magnetic field. From the requirement that the current density is divergence free, Ohm's law yields a Poisson equation for the electric potential field

$$\nabla^2\phi = \nabla \cdot (\boldsymbol{u} \times \boldsymbol{e_B})$$

where ∇^2 is the Laplacian operation, and $\nabla \cdot ()$ denotes the divergence operator.

This command is currently a duplicate of **gvar_mhd_coeff** and will overwrite the prefactor *N* set with **gvar_mhd_coeff**. By default the prefactor takes a zero value. The electric potential filed must be active for the coefficient to have any effect.

See also: **gvar_mhd_coeff**.

Moments

Syntax: **moments [-f <filename> -x <x> <y> -b <bndry_num>]**
Function: **Calculate moments about a boundary.**
Description:
This calculates the moments about a boundary with respect to a specified origin. Moments are calculated from the cross product $r \times dF$, where r is a moment arm vector from the user-specified centre about which the moment is calculated (**<x>**, **<y>**) to the boundary surface. dF is the integral contribution to the body force used by the **forces** command to calculate lift and drag. The resulting moments are counter-clockwise positive. The calculations are only performed if the solution has been initialised.

The following options are available:
> **-f <filename>**
>> Used to specify a filename **<filename>** (including extension) to write (append) the data to. If omitted, the default filename is **moments.dat**.
> **-x <x> <y>**
>> Used to specify the coordinates about which to calculate the moment. The moment arm r extends from coordinate (**<x>**, **<y>**) to the boundary surface. The default is the origin (0, 0).
> **-b <bndry_num>**
>> Used to specify the boundary number (as defined in the **viper.cfg** file) over which the moment is to be calculated. Typically the boundary number would correspond to a closed boundary such as the surface of a cylinder. The default is 0, corresponding to no specified boundary, which does not provide a useful output, but merely avoids the simulation crashing.

Note: The moments command can only be employed in 2D simulations (it does not work in 3D or SE-Fourier 3D simulations).

See also: **forces**.

Nu_horiz_2d

Syntax: **nu_horiz_2d -x <x1> <x2> [<x2> <x3> <x3> … <xn>] -y <y1> <y2> [-scheme <scheme> -k <order> -m –f <filename> -tol <tolerance> -m <depth>]**
Function: **Calculate the bulk temperature Nusselt number (2D channel flow, heated bottom wall).**
Description:
The Nusselt number is calcuated along the bottom edge of a rectangular region defined by the coordinates entered with the **–x** and **–y** options. The **–x** flag accepts a space-separated list of between 2 and 20 x-coordinates in ascending order, which define between 1 to 19 integration regions for Nusselt number. The **–y** flag specifies the lower

and upper y-coordinates for the vertical integration. These should span from a heated bottom boundary to the upper boundary. The Nusselt number is calculated as

$$Nu = \frac{1}{L}\int_{x_1}^{x_n} Nu_w dx = \left(\frac{1}{x_2-x_1}\int_{x_1}^{x_2} Nu_w dx + \frac{1}{x_3-x_2}\int_{x_2}^{x_3} Nu_w dx + \cdots \frac{1}{x_n-x_{n-1}}\int_{x_{n-1}}^{x_n} Nu_w dx\right),$$

where L is the horizontal length of' the heated plate. The local Nusselt number is defined as

$$Nu_w = \frac{h}{T_{bulk}-T_{wall}}\frac{dT}{dy}\bigg|_{wall},$$

where T_{wall} is the local temperature of the heated (bottom) wall, h is a characteristic length, $\frac{dT}{dy}\big|_{wall}$ is the vertical temperature gradient at the heated (bottom) wall, and T_{bulk} is the bulk temperature of the fluid. The bulk temperature is calculated as

$$T_{bulk} = \frac{\int_{y_1}^{y_2} uT\, dy}{\int_{y_1}^{y_2} u\, dy}.$$

The integrations are performed using one of several schemes, the default being the adaptive Simpson's rule. The calculations are only performed if the solution has been initialised.

The following options are available:

-x <x1> <x2> [<x2> <x3> <x3> … <xn>]

    Used to specify at least two x-locations for spatial averaging between. Additional points can be specified (up to 20, any more will be ignored), and must be in ascending order. If the pairs do not share a point, integration will be performed between them (e.g. if the pairs were listed **<x1> <x2> <x3> … <xn>** instead). Furthermore, the maximum length of a line in a configuration file is unlikely to permit 20 points being specified by one **nu_horiz_2d** call, particularly if they are quoted to many decimal places (which will give a line not terminated with a (') symbol error). Multiple **nu_horiz_2d** calls are recommend, which will append the same file if needed (or contact Dr. Gregory Sheard).

-y <y1> <y2>

    Used to specify the lower and upper y-coordinates for vertical Integration. These must be specified for the command to function.

-scheme <scheme>

    An integer flag between 1 and 4 is used to specify the quadrature scheme to be employed, which choices of:

    1: Adaptive trapezoidal scheme (this scheme tends to be slower than the adaptive Simpson's rule, but may be less erroneous at lower tolerances)

    2: Adaptive Simpson's rule (this tends to perform well, and is the default scheme)

    3:Adaptive Gauss-Kronrod scheme (this tends to be slower than the adaptive Simpson''s rule for a given accuracy)

    4: Automatic Simpson's rule (this scheme subdivides all intervals at each iteration and involves redundant function evaluations, and therefore tends to be slower than the adaptive Simpson's rule)

    If the **<scheme>** number does not equal 3, the **-k** option is ignored.

-k <order>

An integer flag between 1 and 6 is used to specify the order of the Gauss-Kronrod scheme to be used. The corresponding numbers of quadrature points are:

1: 7 Gauss points, 15 Gauss-Kronrod points
2: 10 Gauss points, 21 Gauss-Kronrod points
3: 15 Gauss points, 31 Gauss-Kronrod points
4: 20 Gauss points, 41 Gauss-Kronrod points
5: 25 Gauss points, 51 Gauss-Kronrod points
6: 30 Gauss points, 61 Gauss-Kronrod points

The default is 3: 15 Gauss points, 31 Gauss-Kronrod points. If the **<scheme>** number does not equal 3, the **−k** option is ignored.

-n

If included, this option switches on the outputting of local wall Nusselt number to a file named **nu_horiz_2d.dat**. Output includes t, x, wall temperature, bulk temperature, temperature gradient at wall, and the calculated Nu_x. Note that the data will likely not be ordered in x, and will only include data at points evaluated by the quadrature routine.

-f <filename>

Used to specify a filename **<filename>** (including extension) to write the data to. If the **-f** option is not specified, the default filename **nu_horiz_2d.dat** is used.

-tol <tolerance>

Used to specify the convergence threshold for both the bulk temperate and spatial averaging integrations required to evaluate the Nusselt number. The default is 1e-3. If a negative tolerance is specified, the default will be used instead.

-m <depth>

The is **<depth>** an integer parameter used to specify the maximum recursion depth for adaptive schemes. The defines a lower limit on the smallest size a single division of the x-domain (from x_1 to x_2) can becombe. Integration will halt if the division would be smaller than 2^**<depth>**. The default is arbitrarily large. In practice, a **<depth>** of 10 is sufficient for the calculation to not be limited by the **<depth>** for a complex heat flux distribution (although it could be a lot lower for simple functions, in which case it is unnecessary to use). The main reason for specifying a recursion depth is that in rare circumstances a heat flux distribution may be of the wrong shape to ever allow the requested tolerance to be reached. If this occurred, the simulation would integrate indefinitely until the walltime limit kills the task, which is a waste of computational resources. Sadly, no integration scheme is perfect.

Note: The nu_horiz_2d command can only be employed in 2D Cartesian simulations. It will not work in 2D cylindrical, 3D, or SE-Fourier 3D simulations. It also requires an active scalar field (see btag).

See also: **line**.

Nu_xsect_2d

Syntax: **nu_xsect_2d -b <boundary number> [-f <filename>]**

Function: **Calculate the bulk temperature Nusselt number for a 2D channel flow, heated bottom wall.**

Description:

Calculate the bulk temperature Nusselt number. The Nusselt number is integrated over the entire domain, with the heated boundary, which must be specified using the **-b** option. This routine is useful for calculating heat transfer for channel flows into the page. Integration of the bulk temperature is over a cross-section that is normal to the streamwise direction, and averaging is applied over the entire wall. Note that the x-direction is the streamwise direction (into the page), and the y-direction is normal to the bottom wall.

The Nusselt number is calculated as (this may or may not be divided by L)

$$Nu = \int_{x_1}^{x_2} Nu_w \, dx \, .$$

The local Nusselt number is defined as

$$Nu_w = \frac{L}{T_{bulk} - T_{wall}} \frac{dT}{dy}\Big|_{wall} ,$$

where T_{wall} is the local temperature of the heated (bottom) wall, L is the vertical length of the heated plate ($L = x_2 - x_1$), $\frac{dT}{dy}\Big|_{wall}$ is the vertical temperature gradient at the heated (bottom) wall, and T_{bulk} is the bulk temperature of the fluid. The bulk temperature is calculated as

$$T_{bulk} = \frac{\int_{y_1}^{y_2} \int_{x_1}^{x_2} uT \, dxdy}{\int_{y_1}^{y_2} \int_{x_1}^{x_2} u \, dxdy}$$

The integration is performed directly using Gauss-Legendre-Lobatto quadrature on the spectral elements (an iterative scheme is not used, unlike **nu_horiz_2d**). The calculations are only performed if the solution has been initialised.

    -b <boundary number>

    Used to specify the boundary that corresponds to the heated bottom wall. A positive integer boundary number must be entered.

    -f <filename>

        Used to specify a filename **<filename>** (including extension) to write the data to. If the **-f** option is not specified, the default filename **nu_xsect_2d.dat** is used.

Note: The nu_xsect_2d command can only be employed in 2D Cartesian simulations. It will not work in 2D cylindrical, 3D, or SE-Fourier 3D simulations. It also requires an active scalar field (see btag).

See also: **line**, **nu_horiz_2d**.

Onlyw

Syntax: **onlyw**
Function: **Toggles computation of w-velocity-only / all velocity components on/off.**
Description:
This command is implemented only in the 2D Cartesian and axisymmetric cylindrical solvers. During each time step, u- and v-velocity fields are reset to zero, forcing the solution to evolve only in the z- (θ-) direction, and suppressing any instabilities in the x-y (z-r) plane. By default all velocity components are computed.

Order

Syntax: **order [-vel <n>] [-s <n>]**
Function: **Change the order of time integration.**
Description:
By default, the velocity field (and the scalar field, if active) is computed to third order accuracy in time. This command allows the user to alter the order of time integration. In general, a higher order requires a smaller time step. See Chapter 2; Time Integration, for more information.
The following options are available:

>   **-vel <n>**
>>   Used to specify the order of the velocity field (valid options 1 to 3).
>   **-s <n>**
>>   Used to specify the order of the scalar field (valid options 1 to 3).

Overint

Syntax: **overint [-n <n>]**
Function: **Calculate the advection/convection operators at higher resolution.**
Description:
This routine is used to integrate the advection terms in the momentum equation and the convection term in the scalar advection-diffusion equation (if active) at a higher resolution. This is currently only implemented in the 2D solver, and must be called before **init**.
The advection and convection operators involve products of variables with gradients of other variables. Thus a higher resolution is required to properly resolve the result the operation. If the operators are calculated using the elemental polynomial basis (i.e. an element with $P \times P$ quadrature points), aliasing may introduce errors possibly leading to numerical instability (aliasing is where unresolved high-wavenumber/small-scale parts of the solution are mapped erroneously back onto the resolved modes). Aliasing tends to be more of a problem at higher Reynolds/Rayleigh numbers, where the physical viscosity/thermal diffusivity is insufficient to damp the small-scale errors introduced by aliasing. Over-integration tackles this problem by evaluating the advection operator at a higher resolution. The result is then interpolated back to the original elemental basis order.
The following options are available:

>   **-n <n>**
>>   An integer ($\text{<n>} = N$) specifying the number of elemental interpolation points (i.e. $N \times N$ points) upon which the advection term is evaluated

within each element. If this option is not specified, the "2/3rds rule" is invoked, whereby $N = \text{ceiling}(\frac{3P}{2})$. N must be greater than number of quadrature points originally set using **gvar_N** in the **viper.cfg** file.

Pbc

Syntax: **pbc**
Function: **Toggle the high-order Neumann pressure boundary condition off/on.**
Description:
By default, a high-order Neumann boundary condition is imposed on the pressure field on Dirichlet velocity boundaries. This follows from Karniadakis, Israeli & Orszag (1991), who showed that this was required to preserve the 3rd-order accuracy of time integration when using the backwards multistep scheme we employ. This command should be reserved for problem diagnosis: e.g. troublesome instabilities, etc. as the computation may reduce to 1$^{\text{st}}$ order accuracy in time.

Pert

Syntax: **pert <m1> [<m2> <m3> ... <mNfloq_modes>]**
Function: **Establishes perturbation fields for stability analysis.**
Description:
This command must be called prior to a call to **init**, as it is used to specify a number of spanwise (2D Cartesian) or azimuthal (2D axisymmetric) wavenumbers for linear stability analysis. Any number of fields can be specified, though the corresponding increase in memory resources required to compute the flows increases almost linearly with (**<Nfloq_modes>**+1). The spanwise/azimuthal wavelength is $L = \frac{2\pi}{m}$, where m is the wavenumber.

Linear stability analysis can be conducted with the driver command **lsa**. Transient growth analysis can be conducted using the direct time integration approach of Barkley, Blackburn & Sherwin by calling the driver routine **tg**. Transient growth analysis can be conducted using a reconstruction from eigenmodes of the linear evolution operator using the approach of Schmid & Henningson by calling the driver routine **svd**.

If a scalar field is active, calling **pert** will also invoke a scalar perturbation field, which may or may not be desired. This is facilitated for stability analysis of Boussinesq flows - it is unlikely to be useful elsewhere.

Note: floq has been renamed to pert. Furthermore, pert cannot be called if an SE/Fourier computation is initialized (do not call fourier if using pert). It also cannot be invoked in 3D, and init must be called after pert to initialize time stepping.

See also: **arnoldi**, **lsa**, **pert2**, **pert_ke_evol**, **svd**, **tg**.

Pert2

Syntax: **pert2 -fst <m1> -lst <mNfloq_modes> -n
<Nfloq_modes> [-log]**
Function: **Sets linearised perturbation fields.**
Description:

This command must be called prior to a call to **init**, as it is used to specify a number of spanwise (Cartesian) or azimuthal (cylindrical) wavenumbers for linear perturbation fields. The spanwise/azimuthal wavelength is $L = \frac{2\pi}{m}$, where m is the wavenumber.

Linear stability analysis can be conducted with the driver command **lsa**. Transient growth analysis can be conducted using the direct time integration approach of' Barkley, Blackburn & Sherwin by calling the driver routine **tg**. Transient growth analysis can be conducted using a reconstruction from eigenmodes of the linear evolution operator using the approach of Schmid & Henningson by calling the driver routine **svd**.

If a scalar field is active, calling **pert2** will also invoke a scalar perturbation field, which may or may not be desired. This is facilitated for stability analysis of Boussinesq flows - it is unlikely to be useful elsewhere.

The following options are available:

-fst <m1>
> Used to specify the first wavenumber in the sequence, which must be a non-negative integer. The default value is 0.0. No perturbation fields are established if this is not specified.

-lst <mNfloq_modes>
> Used to specify the last wavenumber in the sequence, which must be a non-negative integer. The default value is 0.0. No perturbation fields are established if this is not specified.

-n <Nfloq_modes>
> Used to specify the number of wavenumbers in the sequence. The number of floquet modes be a positive integer. The default value is 0, which will cause an error if left unmodified. No perturbation fields are established if this is not specified.

-log
> Invokes a logarithmic spread of wavenumbers rather than a linear spread. This option cannot be used with a zero wavenumber (as $\log(0) = -\infty$).

E.g. 1:

\> pert2 -fst 0.0 -lst 5.0 -n 3

Gives wavenumbers 0.0, 2.5, 5.0.

E.g. 2:

\> pert2 -fst 1.0 -lst 16.0 -n 5 -log

Gives wavenumbers 1.0, 2.0, 4.0, 8.0, 16.0.

Note: pert cannot be called if an SE/Fourier computation is initialized (do not call fourier if using pert). It also cannot be invoked in 3D, and init must be called after pert to initialize time stepping.

See also: **arnoldi**, **lsa**, **pert**, **pert_ke_evol**, **svd**, **tg**.

Pert_ke_evol

Syntax: **pert_ke_evol [-p <prefix> -k <field>]**
Function: **Outputs the out-of-plane averaged perturbation kinetic energy evolution terms.**
Description:
This command calculates the local minimum, local maximum and volume integrated values of the terms of the out-of-plane averaged linearised perturbation kinetic energy evolution equation. This equation is found by taking the dot product of the perturbation velocity vector with the momentum equation of the linearised perturbation field, then averaging in the out-of-plane direction. The calculations will only be perfomed if the solution has been initialised
The following options are available:

> **-p <prefix>**
>> Used to specify a string containing the filename prefix (three text files are output, **<prefix>_pert_KE_evol_terms_min.dat**, **<prefix>_pert_KE_evol_terms_max.dat**, **<prefix>_pert_KE_evol_terms_total.dat**   for   the minimum, maximum and total values for each term, respectively). If the **-p** option is not specified, the default prefix **pert_KE_evol.dat** is used.

> **-k <field>**
>> An integer ranging from 1 to the number of active perturbation fields (**Nfloq_modes**) in the simulation from which the min/max/total values are to be calculated. The default field number is 1.

Note: pert_ke_evol can only be employed in 2D simulations. It also requires an active linearised perturbation field (a previous pert call).
See also: **arnoldi**, **lsa**, **pert**, **pert2**.

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: 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 [-l <level> -k <field>]**
Function: **Add a random perturbation to the velocity field.**
Description:
This command adds a small random perturbation to the velocity field of an initialized computation. This can help accelerate the development of instability or transient flow features. The random noise will be divergence free if added to the base flow, or if added to a 3D Fourier simulation. Without a call to **rand**, the user relies on noise at the limit of numerical precision to trigger the growth of instabilities. The solution must be initialised for a random perturbation to be added.
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.
The following options are available:

> **-l <level>**
>> Used to set the magnitude (**<level>**) of the added noise. A positive value must be specified. By default, **<level>** is 1e-4. It is distributed in physical space, not Fourier space, and hence, the random noise will be distributed differently as the macro-element distribution, or polynomial order, are varied.

> **-k <field>**
>> For spectral-element-Fourier three-dimensional simulations, this option allows only a specified Fourier mode (i.e. **<field>** = 1, 2, 3, etc.) to be perturbed, rather than all fields (which is the default behaviour). The **<field>** value must be no greater than the number of Fourier modes. For computations where linearized perturbation fields are being evolved, this option can be used to specify a single mode to be perturbed (**<field>** = 0 is the base flow, and positive integers (1, 2, 3, etc.) identify each linearized perturbation field). The **<field>** value must be no greater than the number of perturbation fields which are active.

See also: **fourier**, **pert**.

Reconload

Syntax: **reconload [-f <filename> -i <scheme> -p −t <tau>]**
Function: **Load velocity fields from a data file saved using reconstore.**
Description:
If the user has earlier stored snapshots of the flow field using **reconstore**, then this command is used to load the data from the file, and directs Viper to reconstruct the

velocity field from these snapshots instead of using the standard time integration scheme. The velocity, pressure, and scalar fields will be reconstructed, provided they were stored in the file. The user must ensure that the computation proceeding after **reconload** is consistent with that used when **reconstore** was called. For instance, a different mesh, polynomial order, setting for **wvel**, presence or otherwise of a scalar field, could all lead to unpredictable results.

The following options are available:

-f <filename>

Used to specify a filename **<filename>** (including extension) to write the data to. If omitted, the default filename is **reconstore.dat**.

-i <scheme>

Used to specify the interpolation scheme used. Available choices are fourier (default), polynomial, or akima (e.g. **−i fourier**, **−i poly**, **−i akima**). Akima interpolation is preferred to polynomial interpolation as it is provides a much smoother curve without the spurious wiggles plaguing polynomial and cubic spline interpolation schemes.

-p

Used to reconstruct the pressure field. By default, the pressure field is not reconstructed (saving compute time), as it is not needed for stability analysis. However, if the user wishes to reconstruct the pressure field (e.g. for generating plots of the pressure field), the **−p** option must be specified when this command is called.

-t <tau>

The file created using **reconstore** has no information about the time interval used to store the snapshots of the solution: the user specifies this with the **<tau>** parameter, which should be set to the full time interval over which the snapshot data was acquired (usually this would be the period of the solution). However, as **<tau>** is supplied with this command, which is called at the beginning of a computation used to perform a subsequent stability analysis (for instance), the user could set **<tau>** to be different from the original period, if they desired. This would rarely be required. Note: For Fourier interpolation, **<tau>** is the period of the flow field, whereas for polynomial interpolation, **<tau>** is the time between the first and last snapshot. Polynomial interpolation is only useful if the time of the computation remains within $0 < t < $ **<tau>**, to avoid ludicrous extrapolation errors.

Note: reconload should be called after init, and before time stepping commences.

See also: **reconstore**.

Reconstore

Syntax: **reconstore [-n <Nfields> -f <filename>]**
Function: **Stores velocity fields for later reconstruction by interpolation.**
Description:

Sometimes it is convenient to store a time-varying velocity field solution as a series of snapshots for later reconstruction using an interpolation scheme. This is particularly helpful during stability analysis, where it could be wasteful to continue to time integrate

a periodic base flow over the numerous periods required for the eigenvalue iterations to converge. This command is used to store snapshots of the flow field for later interpolation. The snapshots must be stored at equi-spaced time intervals. Reconstruction is achieved by calling **reconload**, for which a Fourier interpolation can be used if the stored flow is periodic, and either polynomial or Akima spline interpolation is available for transient fields. If the user wishes to store a periodic solution (for reconstruction using Fourier interpolation), they must ensure that the solution at the beginning of the period is only stored ONCE, and not again at the end of the period. For reconstruction using polynomial and Akima interpolation, snapshots of the first and last fields must be explicitly stored, as they are not necessarily the same. On the first call to this command, the data structures are created, based on the active fields in the computation (e.g. u, v, w, p and/or s), and the current velocity field is saved as the first snapshot. On subsequent calls to **reconstore**, any supplied options (**-n** or **-f**) are ignored, and the velocity field at the present time is stored as subsequent snapshots.

The following options are available:

> **-n <Nfields>**
>> Used to specify the number of field snapshots to save.interpolation scheme used. If **reconstore** is called in an SE/Fourier 3D computation, only the (real) fundamental spanwise/azimuthal mode is stored: in other words, the spanwise-averaged velocity field is stored, not the three-dimensional solution. The default is **<Nfields>** = 1.

> **-f <filename>**
>> Used to specify a filename **<filename>** (including extension) to save all **<Nfields>** to (all fields are saved after the final snapshot is stored). If omitted, the default filename is **reconstore.dat**.

Note: reconstore should first be called after INIT, and at a time when the flow has been advanced to the point that the first field is to be stored.

See also: **reconload**.

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 unit time).

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

1) The solid-body rotation of the reference frame is subtracted from the frozen rotating base flow, U, supplied to the advection term for calculation of the perturbation field evolution (this puts the base flow in the rotating frame of reference consistent with the perturbation field).

2) The correction due to the Coriolis acceleration $-2\boldsymbol{\omega} \times \mathbf{v}_{rotating}$ 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> -k <field> -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 (du/dx, etc.), kinematic static pressure (p), and strain rate magnitude at a physical point on the mesh. 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. The command **`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.

    `-k <field>`

        Used to specify an integer perturbation field number (i.e., 1, 2, ... , **`Nfloq_modes`**, when Floquet analysis is active) to interpolate data from. The default is **`<field>`** $= 0$, corresponding to the base flow.

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

        Used to specify the (x,y) or (x,y,z) coordinates of a point in the computational domain at which to interpolate the flow values. Any coordinates not explicitly specified are taken to be equal to zero. The z-coordinate is used for hexahedral 3D runs, spectral element-Fourier 3D runs, and 3D perturbation fields.

Samplef

Syntax: **samplef [-f <filename> -x <x> <y>]**
Function: **Return the Fourier coefficients of the velocity field at a point.**
Description:
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 **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 **samplef.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. Any coordinates not explicitly specified are taken to be equal to zero.

Note: **samplef** can only be called during SE/Fourier computations.
See also: **autocorrf**, **energyf**, **fourier**.

Save

Syntax: **save [-f <filename> -hugh -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 mesh parameters **Nelem**, **Nglobal**, **Nqdpts** to a user-specified file. The computation can only be saved if the simulation has been initialised. The saved fields can then be reloaded using the **load** command to re-start a computation.
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.

> **-hugh**
>> Used to write out u, v, w and p fileds in ASCII format readable by SEMTEX, Prof Hugh Blackburn"s spectral element code (2D and perturbation fields only). Note: The **-k** option must be used to specify which perturbation field is to be written to the file.

-k <floq_mode>

> Used to specify an integer perturbation field number (i.e., 1, 2, ..., **<Nfloq_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 <operation>**
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 the same backwards-multistep time integration approach as used to solve the velocity field. If a scalar field is active and **pert** is called, a scalar perturbation field is established. 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 option 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.

See also: **gvar_scalar_diff**, **pert**.

Set

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

Change the value of a variable - supported variables are:

- **RKV**   Reciprocal kinematic viscosity
- **dt**   Time step
- **t**   Time

The value of the variable and other parameters are input as **`<param>`** values as required. If the time step is changed, the flow fields at previous time intervals will be interpolated to the new times. Set **`dt`** should be used after all **`load`** calls and before **`init`**. Note that if typing an equal signs, spaces must be placed on either side of the equals sign. For example, to set the reciprocal kinematic viscosity to 173.5, type:

`\> set RKV 173.5`

or

`\> set RKV = 173.5`

but not

`\> set RKV=173.5`

Note that as an alternative usage, users may change variables **`RKV`**, **`dt`**, or **`t`** by omitting the **`set`** command: i.e. to change the time step, users could type:

`\> dt 0.004`

or

`\> dt = 0.004`

Spreadscalar

Syntax: **`spreadscalar [-r <newrange> -p <pivotval>]`**
Function: **Rescale the scalar field to spread the range of values.**
Description:

In some heat transfer jobs, such as duct flows with periodic boundaries for inflow/outflow, the scalar (temperature) boundary conditions might be set up to specify a hot temperature on one wall, while the other boundaries are insulated. Over time, the scalar field will diffuse towards a constant value equal to the hot wall temperature throughout the domain. In these scenarios, the actual temperature values are arbitrary; the focus is instead on the normalised wall heat transfer rates. This command combats the tendency of temperatures to asymptote to a constant value in these situations by rescaling the range of the field. It can be called at any time during time integration (i.e. after **`init`**).
The following options are available:

    `-r <newrange>`

        Used to specify the range of the scalar field after rescaling. The command will first record the difference between the maximum and minimum values of the field. A scale factor is then calculated that when applied to the scalar field will produce a difference between maximum and minimum values equal to **`<newrange>`**. This must be a positive value. If this option is not specified, no rescaling is performed.

-p <pivotval>

> Used to specify a value about which to rescale. For example, if a duct wall has a specified hot temperature value of, say, 1.0, then setting **<pivotval>** = 1.0 will scale around this value so the duct wall temperature is unaffected. If this option is not provided, a default value **<pivotval>** = 0.0 is used.

Note: spreadscalar requires a scalar field to be active. It also is yet to be implemented in SE-Fourier 3D computations.

Stab

Syntax: **stab [<filename>]**
Function: **Calculate Floquet multipliers for each linear instability mode.**
Description:

If Floquet linear stability analysis is being performed (call **pert** 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 the change in the magnitude of each perturbation field with their previous values, and evaluating growth rates based on the previous time at which a **stab** command was called. 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 (when **stab** was called), then

$$|\mu| = \frac{N(t + T)}{N(t)}\,,$$

providing the flow has evolved for a sufficient number of periods to isolate only the fastest-growing mode at the given wavelength. The calculations can only be performed if the solution has been initialised, in a simulation with active perturbation fields.

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.

The following options are available:

> **<filename>**
>
> > The period between **stab** calls and the resulting Floquet multiplier estimates are written to a specified file **<filename>** (including extension). If not specified, the default filename **floq_mult.dat** is used.

See also: **arnoldi**, **pert**.

Step

Syntax: **`step [<num_steps>]`**
Function: **Performs `<num_steps>` time integration steps.**
Description:

If **`<num_steps>`** is not specified, a single time step is completed, otherwise **`<num_steps>`** steps are taken. If zero were specified no time integration is performed. For backwards time integration using the adjoint of the linearised Navier-Stokes equations, supply a negative value to **`<num_steps>`**, i.e. **`step -5`** would evolve a linearised perturbation field 5 steps backwards in time. Note that this will only work with a frozen (see **`freeze`**) or a reconstructed (see **`reconload`**/**`reconstore`**) two-dimensional or axisymmetric base flow. Note that time stepping is only performed if the solution has been initialised. If time stepping has been halted by a stop criterion, calling step again will restart the process. If particle tracking is in use, time stepping will occur in increments of Ntracksteps.

See also: **`freeze`**, **`reconload`**, **`reconstore`**, **`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 [<min_du>]`**
Function: **Sets a stopping criterion on time stepping.**
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. To avoid stopping a set of timesteps early, set **`<min_du>`** 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`**.

Svd

Syntax: **`svd [-fields [u][v][w][s] -prefix <string> -nev <integer> -ncv <integer> -Nsteps <integer> -save -tecp — times <integer> <real> <real> -tol <integer> -vizmat]`**

Function: **Find leading singular value and right singular vector of a linear time integration operator.**

Note: This is a driver routine. It automatically executes a loop, calls the Arnoldi command, and conducts the required time integration. The solution must have been initialised (`init`) and perturbation fields must be active (`pert`).

Description:

Linear stability analysis (activated using **`pert`**, and conducted using **`arnoldi`**) returns the eigenmodes of a linear operator matrix $[A]$. This command computes an approximation of the leading singular value and corresponding right singular vector of this matrix. These correspond to transient growth properties $G(\tau)$ and the corresponding initial vector field producing this peak growth, where tau is the time interval used in the time itegration of the perturbation field. The matrix $[A]$ is partially reconstructed using eigenvectors and eigenvalues found using the Arnoldi method. The approximation improves as the number **`nev`** of requested eigenvalues increases. If only one eigenmode is requested, the result will correspond to the leading linear instability mode. The code outputs a file "**`<prefix>eigenvalues.dat`**", containing the positive spectrum of eigenvalues returned from the linear stability analysis (these are used to estimate the transient growth of the system). The code also outputs a file ""**`<prefix>sqr_singular_values.dat`**", containing the squared singular values at each requested time: the square of the singular value equates to $G(\tau)$, or the amplification of the optimal initial condition for a given time interval.

The following options are available:

`-fields [u] [v] [w] [s]`

        Used to specify which fields are included in the energy norm to be optimized. By default, the norm is a kinetic energy, featuring $u^2 + v^2 + w^2$. If a scalar field is active, s^2 is also included in the norm by default. However, users might only want to optimize a norm containing energy in the horizontal component of velocity, $E = u^2$, say (using **`-fields u`**), or the scalar field only, $E = s^2$, say (using **`-fields  s`**). This option facilitates these capabilities.

`-prefix <string>`

        Used to specify a string containing the filename prefix for eventual output of the SVD solver. If the **`-prefix`** is not specified, the default prefix **`svd_`** is used.

`-nev <integer>`

        Used to specify the number of leading eigenmodes (number of eigenvalues) to be found by the Arnoldi solver prior to the SVD solution phase. These eigenmodes are used to construct an aproximation to the linear operator matrix A. The default value (and minimum allowable value) is 1.

`-ncv <integer>`

        Used to specify the length of the Arnoldi factorization used by the Arnoldi solver prior to the SVD solution phase. The default value is 6, and the minimum allowable value is (**`nev`**+2)].

`-Nsteps <integer>`

Used to specify the number of time steps per Arnoldi iteration update. The time interval τ is calculated as $dt * N_{steps}$. The default value is 1000, and the minimum allowable value is 1.

-save

If included, the solver will output Viper restart files of the right singular vectors (optimal initial conditions) found for each requested τ value.

-tecp

If included, the solver will output Tecplot files of the right singular vectors (optimal initial conditions) found for each requested τ value.

-times \<integer> \<real> \<real>

Used to specify the spread of τ values at which $G(\tau)$ is to be approximated. The first number is an integer: the magnitude specifies the number of τ values, and the sign specifies the spread of values (positive for linear intervals, negative for an exponential spread - intervals increase at larger τ values). The next two floating point numbers provide the start and end τ values for the spread of τ values to be analysed.

-tol \<integer>

Used to specify the exponent of the convergence criterion used for the Arnoldi solver (i.e. 10**^\<integer>**). The default value is -7, corresponding to 10^{-7}.

-vizmat

If included, images will be output in the .pgm format showing the structure of the matrices formed in the calculation of the transient growth.

Svv

Syntax: **svv [-epsi \<epsi> -p \<Pcut> -f \<filter>]**
Function: **Activate spectral vanishing viscosity (SVV) filtering of velocity / scalar fields.**
Description:
Spectral vanishing viscosity is an approach for stabilising high Reynolds/Rayleigh number spectral element simulations by progressively applying a greater amount of artificial viscosity to higher-wavenumber spatial modes of the solution to help dampen spurious oscillations that can arise due to numerical instability or quadrature errors, etc. For further details see Kirby & Sherwin (Comput. Methods Appl. Mech. Eng., 2006), Malm et al. (J. Sci. Comput., 2013).
The following options are available:

-epsi \<epsi>

Used to specify the value of **\<epsi>**, which is a non-negative real number that defines the strength of the filter. The default value is 0.0 (no filter).

-p \<Pcut>

Used to specify the value of **\<Pcut>**, which is a positive integer less than the \<number of quadrature points> – 2, and specifies the mode numbers beyond which the filter is applied. For instance, if a simulation has spectral elements with 15 x 15 quadrature points, and **\<Pcut>** = 10, the filter will only be applied to mode 11 and higher in either direction.

Polynomials of order $> O(x^{<Pcut>+1})$ will be filtered. If this option is not specified, by default **`<Pcut>`** is set equal to the order of the elements so that no filter is applied.

`-f <filter>`

This option determines the selection of modes for filtering. If p and q express the orders of each tensor-product polynomial mode forming the modal basis over each quadrilateral spectral element, then two possibilities are available for application of the filter:

`<filter>` = 1: The filter is applied if p or q are greater than **`<Pcut>`**.
`<filter>` = 2: The filter is applied if $p + q$ is greater than **`<Pcut>`**. These correspond to equations (16) and (15), respectively, from Kirby & Sherwin (2006). By default, **`<filter>`** = 1 if this option is not specified. If a value other than 1 or 2, the value of **`<Pcut>`** will be set to 0.

Note: `svv` must be called before `init`. Furthermore `svv` can only be employed in two-dimensional simulations (not 3D or SE-Fourier 3D).

Tec_floq (Deleted)

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

This command has been deleted from Viper. The same effect (with more flexibility) can be achieved by loading base flow and required perturbation fields (**`load -k`**) into the appropriate Fourier modes of an SE-Fourier 3D simulation and use the regular **`tecp`** output from there. The amplitude of the perturbation field can be scaled up or down for visualization purposes using the options of the **`load`** command.

Tecp

Syntax: **tecp [-buoyancy –cartesian –cylindrical -e -f**
               <filename> -k <Floquet_mode> -m <mode_num> -n
               <plot_interp_pts> -nozero -rotate <deg> -s –t –u
               <function> –vars [<varlist>]]
Function: **Creates a Tecplot binary data file.**
Description:

Creates a Tecplot **`.plt`** binary (or **`.dat`** ASCII) data file containing various flow quantities specified by the user. If **`tecp`** is called before **`init`**, only the mesh (x, y, and z) coordinates are written to the Tecplot binary file, with the default file name **`tec_mesh.plt`**. If **`tecp`** is called after **`init`**, the mesh information and other requested variables are output, with the default filename **`tec_out.plt`** being used. The following options are available:

`-buoyancy`

This option requires a Boussinesq buoyancy-driven flow simulation. This option adds the base flow available potential energy density field to the output variables.

-cartesian

> For SE/Fourier 3D computations in cylindrical coordinates, this option causes the velocity components to be output in the Tecplot data file in a Cartesian sense: i.e., (u, v, w). This can be useful for vector plots in Tecplot. This is option can be abbreviated to **-ca**.

-cylindrical

> For SE/Fourier 3D computations in cylindrical coordinates, this option causes the velocity components to be output in the Tecplot data file in a cylindrical sense: i.e., (u_z, u_r, u_θ), or axial, radial and azimuthal components, respectively.' This is the default behaviour. This option can be abbreviated to **-cy**.

-e

> If the **-e** option is specified when outputting a linearised perturbation field, the various out-of-plane averaged perturbation kinetic energy evolution equation terms are added.

-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. If Floquet stability analysis is not being performed, the base flow is output.

-m <mode_num>

> In spectral-element/Fourier computations, this feature can be used to extract a single Fourier mode from the solution. The parameter **<mode_num>** is an integer, and expresses the number of the desired Fourier mode. That is, if you are computing a solution with 10 Fourier planes, this corresponds to 6 Fourier modes: the fundamental mode (0) plus 5 modes. Therefore **<mode_num>** may take a value from 0 to 5. If **<mode_num>** exceeds 5, it will default to 5, and if it is negative, this feature is ignored.
>
> As well as providing the capability of isolating the contribution of a single mode in a 3D SE/Fourier computation, this facility can be used to delete modes from a plot of an SE/Fourier computation. i.e., The data set in the Tecplot file generated using this option may be subtracted within Tecplot from a file containing all Fourier modes generated the same solution.

-n <plot_interp_pts>

> a) For 2D quadrilateral or 3D hexahedral simulations:
> 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.'

b) For 3D spectral-element/Fourier simulations:
Used to specify the plotting resolution in the spanwise/azimuthal direction. If omitted, the number of Fourier planes is used by default, but experience shows that to resolve detail in the highest mode of the simulation, a value at least 4 times the number of planes should be used.

-nozero

In spectral element-Fourier 3D computations, this removes the fundamental (or zero-wavenumber mode from the solution when generating the output. Note that the fundamental mode is removed AFTER all fields have been calculated.

-rotate <deg>

Used to rotate the output (2D or 3D hexahedral only) by angle **<deg>** clockwise around the x-y plane. E.g. "**-rotate 35.0** " will cause the Tecplot data file to display the mesh rotated clockwise by 35 degrees.

-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.

-u <function>

Used to supply a user-specified function of $t, x, y, z, u, v, w, s, p, SR$ and spatial derivatives of velocity and scalar fields to plot in the Tecplot binary file.

-vars <varlist>

This option replaces the **-o** and **-sr** options (which have been deleted), providing more control over which variables are included in Tecplot files. This is most useful where file sizes are a problem. This option is implemented for 2D, 3D, and SE/Fourier computations. The parameters **<varlist>** is a list of space-separated variable names taken from the following list:
- **vel**: Velocity components,
- **p**: Pressure,
- **vort**: Vorticity components,'
- **ddx**: Spatial velocity gradients ($du/dx, dw/dy, dv/dz$, etc.),
- **sr**: Strain rate magnitude (leading eigenvalue of strain tensor),
- **lambda2**: 2nd eigenvalue of tensor of velocity gradients suggested by Jeong & Hussain (1995) to identify vortex structures,
- **psi**: Streamfunction (2D simulations only).

By default **vel**, **p** and **vort** are provided without needing to specify a variable list. The fields **vel_mag** and **grad_u** are no longer available, as they can be calculated trivially within Tecplot from the **vel** or **ddx** fields, respectively. This is option can be abbreviated to **-v**.

Tg

Syntax: **tg   [-prefix   <string>   -nev   <integer>   -ncv <integer> -tol <integer> -Nsteps <integer>]**

Function: **Driver routine for transient growth analysis.**

Note: This is a driver routine. It automatically executes a loop, calls the Arnoldi command, and conducts the required time integration. The solution must have been initialised (init) and perturbation fields must be active (pert). The computation must also be two-dimensional and either freeze or reconload must be used with tg, else no computations can be performed.

Description:

Transient growth analysis is used to find the maximum possible amplification of energy of' a linear mode over a specified time interval, τ, and the corresponding optimal initial condition.

The following options are available:

>   **-prefix <string>**
>
>>   Used to specify a filename prefix for eventual output of the TG solver. The default is **tg_**.
>
>   **-nev <integer>**
>
>>   Used to specify the number of leading eigenmodes (number of eigenvalues) to be found by the Arnoldi solver. The default value (and minimum allowable value) is 1.
>
>   **-ncv <integer>**
>
>>   Used to specify the length of the Arnoldi factorization used by the Arnoldi solver. The default value is 6, and the minimum allowable value is (**nev**+2)].
>
>   **-tol <integer>**
>
>>   Used to specify the exponent of the convergence criterion used for the Arnoldi solver (i.e. 10^**<integer>**). The default value is -7, corresponding to 10^{-7}.
>
>   **-Nsteps <integer>**
>
>>   Used to specify the number of time steps per Arnoldi iteration update. The time interval τ is calculated as $dt * N_{steps}$. The default value is 1000, and the minimum allowable value is 1.

See also: **freeze**, **init**, **pert**, **reconload**, **transgrowth**.

Tic

Syntax: **tic**

Function: **Start stopwatch timer.**

Description:

tic starts the stopwatch timer. The internal system time is recorded, and elapsed time can be displayed by calling the **toc** command.

See also: **toc**.

Timeavg

Syntax: `timeavg [-save <filename> -tecp <filename> -u <function> -vars [<varlist>]]`

Function: **Driver routine for time averaging of flow solution.**

Note: This is a driver routine. The solution must have been initialised (`init`).

Description:

This command facilitates the recording of a time average of the flow solution. This command must be called after `init`. When it is first called, memory is allocated to store the time average of the flow solution, and the current field is stored (being the "time average" of a single field). Every subsequent call to this command updates the estimate of the time average, by scaling down the stored previous time average estimate by $N/(N + 1)$, where N is the number of fields already stored, then adding the new field scaled down by $1/(N + 1)$.

Note on usage: The time average is estimated by $\text{sum}(u_i)/N$, where N is the number of stored fields, and u_i is the $i'th$ stored field. It is therefore a discrete mean estimate, and will be closer to the theoretical exact mean for smaller time intervals between each `timeavg` call, and for larger N.

Example of usage - in this example the time-average estimate is updated every 10 time steps, but to save overall computation time, the time-averaged fields are output to file once every 500 calls (i.e. once every 5000 time steps). This is achieved using nested loops in a macro file or from the Viper command line:

```
\> loop 100
\>   loop 500
\>     step 10
\>     timeavg
\>   endl
\>   timeavg -save tavg_save.dat -tecp tavg_tecp.plt …
          -vars vel p vort ddx
\> endl
```

The following options are available:

`-save <filename>`

    Save the current estimate of the time-averaged flow fields to a Viper binary restart file with specified **`<filename>`** (preferred extension **`.dat`**).

`-tecp <filename>`

    Save the current estimate of the time-averaged flow fields to a Tecplot file with specified **`<filename>`** (preferred extension **`.plt`**).

`-u <function>`

    Used to supply a user-specified function of $t, x, y, z, u, v, w, s, p, SR$ and spatial derivatives of velocity and scalar fields to plot in the Tecplot binary file.

`-vars <varlist>`

    This option replaces the **`-o`** and **`-sr`** options (which have been deleted), providing more control over which variables are included in Tecplot files. This is most useful where file sizes are a problem. This option is implemented for 2D, 3D, and SE/Fourier computations. The parameters

`<varlist>` is a list of space-separated variable names taken from the following list:

- **`vel`**: Velocity components,
- **`p`**: Pressure,
- **`vort`**: Vorticity components,'
- **`ddx`**: Spatial velocity gradients ($du/dx, dw/dy, dv/dz$, etc.),
- **`sr`**: Strain rate magnitude (leading eigenvalue of strain tensor),
- **`lambda2`**: 2nd eigenvalue of tensor of velocity gradients suggested by Jeong & Hussain (1995) to identify vortex structures,
- **`psi`**: Streamfunction (2D simulations only).

By default **`vel`**, **`p`** and **`vort`** are provided without needing to specify a variable list. This option can be abbreviated to **`-v`**.

Toc

Syntax: **`toc`**
Function: **Display elapsed time from stopwatch.**
Description:
When **`toc`** is called, the elapsed time in seconds since **`tic`** was last called is output to screen. If **`tic`** has not been called, **`toc`** has no effect. Multiple **`toc`** calls may follow a single call to **`tic`**.
See also: **`tic`**.

Tony_psi

Syntax: **`tony_psi [-f <filename> -n <points> -r <r_min> <r_max> -w <omega> -z <z_val>]`**
Function: **Output streamfunction in a rotating frame from an SE-Fourier 3D run.**
Description:
This routine calculates the 2D streamfunction on a plane of constant z from an SE-Fourier 3D simulation in cylindrical coordinates. The streamfunction is computed relative to a rotating reference frame. This computation can only be performed if the solution has been initialised. It can only be employed in SE-Fourier 3D simulations in cylindrical (**`axi`**) coordinates.
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 **`tony_psi.dat`** is used.

> **`-n <points>`**
> > Used to specify the number of discrete points in the radial direction over which the interpolation of data for evaluation of the streamfunction is to take place. More points will improve the accuracy of the answer, but will incur a higher cost. **`<points>`** must be an integer >= 2. The default value is **`<points>`** = 100.

> **`-r <r_min> <r_max>`**

Used to specify the minimum and maximum radial coordinates between which the streamfunction is evaluated. By default, **`<r_min>`** = 0.0 and **`<r_max>`** = the maximum radial coordinate in the mesh (which may or may not be within the domain at the chosen z-value, so this option should not be omitted). **`<r_max>`** must be greater than **`<r_min>`** and both must lie within the domain.

`-w <omega>`

Used to specify any real value for the angular velocity of a rotating reference frame (taken relative to the reference frame of the simulation) upon which to compute the streamfunction. The azimuthal velocity will be altered by $u_\theta = u_{\theta,old} - $ **`<omega>`** $\cdot r$. By default, **`<omega>`** $= 0$ (i.e. no adjustment for a rotating reference frame).

`-z <z_val>`

Used to specify the z-coordinate of the r-θ plane on which the streamfunction is to be specified. By default, the mid-point of the range of z-values in the mesh is chosen.

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 4th-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.

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

`track diff <Sc>`

By default, particles transport with no diffusion, precisely following the flow. This command activates diffusion by means of a Gaussian-distributed random walk, whereby particle positions are adjusted by a Gaussian-distributed random number at each time step. The variance of the random number relates to the Schmidt number (supplied as **`<Sc>`**) through $\sigma^2 = 2\nu \cdot dt/Sc$ where σ^2 is the variance, and ν the kinematic viscosity. Smaller Schmidt numbers therefore represent more diffusion. If no Schmidt number is supplied no diffusion will be added to particle transport.

`track inject`

Tracer injection points are loaded from a text file named **`track_pts`**, which first gives the number of injection points, then lists the x,y, z-coordinates of each point (only two spatial coordinates per line are searched for in two-dimensional computations). One injection point is given per line, and a large number of points may be established concurrently. Particles are injected at each of these points every time

particle positions are updated (during time integration). Then particle tracking is initialized.

track inject_off

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

track inject_steps <Ninject_steps>

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

track load [-f <filename>]

Loads a binary restart file to a file **<filename>** (including extension) if the **-f** option is specified, or a default file **restart_ptcls.dat** if not. Note that to restart a particle transport simulation, the user also needs to save the velocity field using the **save** command, and then must use both **load** for the velocity field, the same **track** commands to initialize particle tracking, and then finally **track load** after **init** is called in the restarted simulation.

track sample [<filename>]

Saves velocity field information at each particle location to a text file **<filename>** (including extension). 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. If particle tracking has not been initialised (see **track seed** or **track load**) then no action is taken.

track save [-f <filename> -s]

Saves a binary restart file to a file **<filename>** (including extension) if the **-f** option is specified, or a default file **restart_ptcls.dat** if not. The **-s** option is used to create a numbered sequence of files instead of overwriting a single file. 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 **track save** call is made with the **-s** option. Note that to restart a particle transport simulation, the user also needs to save the velocity field using the **save** command, and then must use both **load** for the velocity field, the same **track** commands to initialize particle tracking, and then finally **track load** after **init** is called in the restarted simulation. If particle tracking has not been initialised (see **track seed** or **track load**) then no action is taken.

track saveold [<filename>]

Saves information (invoking the old ASCII particle output) about particles to a text file **<filename>** (including extension). 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. If particle tracking has not been initialised (see **track seed** or **track load**) then no action is taken.

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 inject`**, incorporating a rake of injection points.

`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 tecp [-f <filename> -ascii -s]`

Outputs particle data in Tecplot binary format (use extension **`.plt`**) by default, or in ASCII text format if the **`-ascii`** option is specified (use extension **`.dat`**). By default a filename **`tecp_ptcls.plt`** (or **`tecp_ptcls.dat`** for ASCII output) is used, or the user can specify their own filename using the **`-f`** option. Use the **`-s`** option to append a sequence number to the filename to store a sequence of files rather than overwriting the same file if multiple **`track tecp`** calls are made in a loop. **`track tecp`** will only create an output file if the both the computation and particle tracking have been initialised (see **`init`** and **`track seed`** or **`track load`**). The **`-s`** option is used to create a numbered sequence of files instead of overwriting a single file. 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 **`track tecp`** call is made with the **`-s`** option.

Transgrowth

Syntax: **`transgrowth <numsteps>`**

Function: **Perform forward and adjoint time integration for transient growth stability analysis.**

Description:

This command time integrates a linearised perturbation field, forward in time by a given number of steps, and then backward in time using the adjoint of the linearised equations. The user must call either **`stab`** or **`arnoldi`** (after each call to **`transgrowth`**), to respectively invoke either a simple power iteration method or the implicitly restarted Arnoldi method for finding the leading eigenvalue magnitude (transient growth amplification factor) and eigenvector (initial disturbance). This forms part of a manual version of the **`tg`** command. Its only practical utility is to use the simple power iteration method (**`stab`**), if the Arnoldi method consistently fails. It gives less information, providing only the magnitude of the leading eigenvalue rather than the full complex form of possibly several leading eigenvalues; though for transient growth, only the leading eigenvalue is of interest (as it must be positive and have only a real component).

Note: `transgrowth` can only be employ

Note: The solution must have been initialised (`init`) and perturbation fields must be active (`pert`). The computation must also be two-dimensional and either `freeze` or `reconload` must be used with `transgrowth`, else no computations can be performed.

See also: `freeze`, `init`, `pert`, `reconload`, `tg`.

Vismat

Syntax: `vismat`

Function: **Toggles 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 `p_laplace_matrix.pgm` and `X_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. X are velocity components u, v, w (if active) and scalar field s (if active), and Y is either pre_fact or post_fact. Images are written both before and after factorization. Please inform the developer if you would find a binary image file format preferable for output. **Note: This command must be called before `init`, otherwise no images will be produced.**

Womersley

Syntax: `womersley [-f <filename> -d <diameter> -u -w <omega>]`

Function: **Initialise the Womersley velocity profile.**

Description:

This command is called to impose a Womersley profile on a simulation. It is assumed that the profile will be imposed on the axial component of velocity (x-direction on a 2D mesh), and the profile is formulated for cylindrical coordinates (therefore the `axi` command should also be used). The user must supply a file containing Fourier coefficients of either a time-varying axial kinematic pressure gradient or an area-averaged velocity. The Womersley profile is computed using the analytical solution for flow in a pipe driven by a time-varying pressure gradient derived by J. R. Womersley (J. Physiol., vol. 127, 553-563, 1955). Once activated, the Womersley profile will be written onto any Dirichlet velocity boundary with a radial distance within the specified diameter of the pipe.

The following options are available:

    `-f <filename>`

        Used to specify the file from which the Fourier coefficient data is input from. The first line must contain an integer specifying the number of Fourier modes contained in the file (N). This must be followed by N rows, each containing two numbers (the real and imaginary components of each Fourier mode). It is assumed that the fundamental mode is the

first row, and positive frequency contributions follow in ascending order. Note that only positive frequency contributions may be included in this sequence. If this option is not specified, the default filename is **womersley_pgrad_coeffs.dat**.

-d <diameter>
Used to specify the diameter of the pipe in which the Womersley profile is being employed. If this option is not specified, the default diameter is 1 unit.

-u

The supplied Fourier coefficient data can represent either kinematic pressure gradient data (the default), or area-averaged velocity data. If this option is supplied, the profile will be calculated based on area-averaged velocity coefficients.

-w <omega>
Used to specify the angular velocity of the pressure gradient driving the Womersley profile. If this option is not specified, the default angular frequency is 1.

Wvel

Syntax: **wvel**

Function: **Toggles z/θ-component of velocity on or off in two-dimensional computations (default is OFF).**

Description:

By default, Viper only evolves in-plane velocity components in two-dimensional simulations (i.e., only u and v, but not w-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.

Note that the computations will still be two-dimensional – that is, there is still no variation (zero spatial gradients) in the third dimension. Furthermore, it can only be invoked for two-dimensional simulations. In addition, if axirotate has been specified wvel cannot be turned off (it will always remain active, as axirotate turns wvel on).

See also: **axi**, **axirotate**.

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 (2001) Nonlinear particle tracking for high-order elements. *J. Comput. Phys.* **172**(1), 356-386

Davidson, P. A. (1995) Magnetic damping of jets and vortices. *J. Fluid Mech.* **299**, 153-186

Davidson, P. A. (2001) *An Introduction to Magnetohydrodynamics*. Cambridge University Press

Gray, D. D. & Giorgini, A. (1976) The validity of the Boussinesq approximation for liquids and gases. *Int. J. Heat & Mass Tran.* **19**(5), 545-551

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.

Kirby, R.M. & Sherwin, S.J. (2006) Stabilisation of spectral/hp element methods through spectral vanishing viscosity: Application to fluid mechanics modelling. *Comput. Methods Appl. Mech. Eng.* **195**(23), 3128-3144

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.

Malm, J., Schlatter, P., Fischer, P.F. & Henningson, D.S. (2013) Stabilization of the spectral element method in convection dominated flows by recovery of skew-symmetry. *J. Sci. Comput.* **57**(2), 1-24

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

Pothérat, A. (2007) Quasi-two-dimensional perturbations in duct flows under transverse magnetic field. *Phys. Fluids*, **19**(7), 074104

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*.

Schmid, P.J., & Henningson, D.S. (2001) *Stability and Transition in Shear Flows*. Springer-Verlag New York.

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.

Sommeria J. & Moreau R. (1982) Why, how, and when, MHD turbulence becomes two-dimensional. *J. Fluid Mech.* **118**, 507-518

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.

Womersley, J.R. (1955) Method for the calculation of velocity, rate of flow and viscous drag in arteries when the pressure gradient is known. *J. Physiol.* **127**(3), 553-563

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

Appendix A

Derivation of the quasi-static MHD equations

The relevant equations are Ohm's law, and the divergence of Ohm's law under the assumption of solenoidal currents, noting also that the current density appears in the $N(\boldsymbol{j} \times \boldsymbol{e_B})$ term in the momentum equation:

$$\boldsymbol{j} = -\boldsymbol{\nabla}\phi + \boldsymbol{u} \times \boldsymbol{e_B}$$

$$\nabla^2\phi = \boldsymbol{\nabla} \cdot (\boldsymbol{u} \times \boldsymbol{e_B})$$

The Poisson equation for the electric potential with a magnetic field in the +y direction (Cartesian coordinate system):

$$\nabla^2\phi = \boldsymbol{\nabla} \cdot (\boldsymbol{u} \times \boldsymbol{e_B})$$

$$\nabla^2\phi = \boldsymbol{\nabla} \cdot \left(\left(u\boldsymbol{e_x} + v\boldsymbol{e_y} + w\boldsymbol{e_z} \right) \times \left(\boldsymbol{e_y} \right) \right)$$

$$\nabla^2\phi = \boldsymbol{\nabla} \cdot (-w\boldsymbol{e_x} + u\boldsymbol{e_z})$$

$$\nabla^2\phi = -\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}$$

$$\boldsymbol{j} = -\boldsymbol{\nabla}\phi + (\boldsymbol{u} \times \boldsymbol{e_B})$$

$$\boldsymbol{j} = -\left(\frac{\partial\phi}{\partial x}\boldsymbol{e_x} + \frac{\partial\phi}{\partial y}\boldsymbol{e_y} + \frac{\partial\phi}{\partial z}\boldsymbol{e_z} \right) + (-w\boldsymbol{e_x} + u\boldsymbol{e_z})$$

$$= \left(-\frac{\partial\phi}{\partial x} - w \right)\boldsymbol{e_x} - \frac{\partial\phi}{\partial y}\boldsymbol{e_y} + \left(-\frac{\partial\phi}{\partial z} + u \right)\boldsymbol{e_z}$$

$$N(\boldsymbol{j} \times \boldsymbol{e_B}) = N\left[\left(-\frac{\partial\phi}{\partial x} - w \right)\boldsymbol{e_x} - \frac{\partial\phi}{\partial y}\boldsymbol{e_y} + \left(-\frac{\partial\phi}{\partial z} + u \right)\boldsymbol{e_z} \right] \times \boldsymbol{e_y}$$

$$= N\left[\left(\frac{\partial\phi}{\partial z} - u \right)\boldsymbol{e_x} + \left(-\frac{\partial\phi}{\partial x} - w \right)\boldsymbol{e_z} \right]$$

The Poisson equation for the electric potential with a magnetic field in the axial +z direction (Cylindrical coordinate system):

$$\nabla^2\phi = \boldsymbol{\nabla} \cdot (\boldsymbol{u} \times \boldsymbol{e_B})$$

$$\nabla^2\phi = \boldsymbol{\nabla} \cdot \left(\left(u_r\boldsymbol{e_r} + u_\varphi\boldsymbol{e_\varphi} + u_z\boldsymbol{e_z} \right) \times \left(\boldsymbol{e_z} \right) \right)$$

$$\nabla^2\phi = \boldsymbol{\nabla} \cdot \left(u_\varphi\boldsymbol{e_r} - u_r\boldsymbol{e_\varphi} \right)$$

$$\nabla^2 \phi = \frac{1}{r} \frac{\partial}{\partial r} \left(r u_\varphi \right) - \frac{1}{r} \frac{\partial u_r}{\partial \varphi}$$

$$\boldsymbol{j} = -\boldsymbol{\nabla}\phi + (\boldsymbol{u} \times \boldsymbol{e}_B)$$

$$\boldsymbol{j} = -\left(\frac{\partial \phi}{\partial r} \boldsymbol{e_r} + \frac{1}{r} \frac{\partial \phi}{\partial \varphi} \boldsymbol{e_\varphi} + \frac{\partial \phi}{\partial z} \boldsymbol{e_z} \right) + \left(u_\varphi \boldsymbol{e_r} - u_r \boldsymbol{e_\varphi} \right)$$

$$= \left(-\frac{\partial \phi}{\partial r} + u_\varphi \right) \boldsymbol{e_r} + \left(-\frac{1}{r} \frac{\partial \phi}{\partial \varphi} - u_r \right) \boldsymbol{e_\varphi} - \frac{\partial \phi}{\partial z} \boldsymbol{e_z}$$

$$N(\boldsymbol{j} \times \boldsymbol{e}_B) = N \left[\left(-\frac{\partial \phi}{\partial r} + u_\varphi \right) \boldsymbol{e_r} + \left(-\frac{1}{r} \frac{\partial \phi}{\partial \varphi} - u_r \right) \boldsymbol{e_\varphi} - \frac{\partial \phi}{\partial z} \boldsymbol{e_z} \right] \times \boldsymbol{e_z}$$

$$= N \left[\left(-\frac{1}{r} \frac{\partial \phi}{\partial \varphi} - u_r \right) \boldsymbol{e_r} + \left(\frac{\partial \phi}{\partial r} - u_\varphi \right) \boldsymbol{e_\varphi} \right]$$