M3DP USER'S GUIDE

Joshua A. Breslau ⟨jbreslau@pppl.gov⟩

January 15, 2002

Contents

1 The M3D Form of the Resistive MHD Equations **4**
- 1.1 Operators . . . 4
- 1.2 The Scalar Variables . . . 5
  - 1.2.1 The Magnetic Field and Currents . . . 5
  - 1.2.2 The Velocity Components . . . 7
- 1.3 The Scalar Equations . . . 7
  - 1.3.1 Electromagnetic quantities . . . 7
  - 1.3.2 Velocities . . . 8
  - 1.3.3 Other Quantities . . . 10

2 Walkthrough **12**
- 2.1 Running the Example . . . 12
- 2.2 Reading the Data . . . 13
- 2.3 Initializing the Mesh . . . 15
  - 2.3.1 Communicator Initialization . . . 15
  - 2.3.2 Index Initialization . . . 16
  - 2.3.3 Geometry Initialization . . . 21
  - 2.3.4 Finite Element Initialization . . . 23
- 2.4 Initializing the Physics . . . 28
  - 2.4.1 Mesh Allocation . . . 29
  - 2.4.2 Parameter Initialization . . . 30
  - 2.4.3 Field Initialization . . . 31
  - 2.4.4 Solver Initialization . . . 33
- 2.5 Preparing to Iterate . . . 35
  - 2.5.1 Toroidal Derivatives . . . 37
  - 2.5.2 The F Equation . . . 41
  - 2.5.3 The ψ Equation . . . 46
  - 2.5.4 Initial Velocity Perturbation . . . 47
  - 2.5.5 The Φ Equation . . . 49
  - 2.5.6 Deviations from Equilibrium . . . 56
  - 2.5.7 The Multiple-degree-of-freedom Matrix . . . 56
- 2.6 Iterating . . . 57
  - 2.6.1 Reading the Parameter File . . . 57
  - 2.6.2 Fortran Initializations . . . 58

A The `wxy` File **62**

Index **66**

Chapter 1

The M3D Form of the Resistive MHD Equations

1.1 Operators

All equations will be solved in toroidal geometry, so we use cylindrical coordinates (R, z, φ). Note that this differs from the more common convention (used, e.g., by the **VectorAnalysis** package of *Mathematica* and in most reference books) of ordering them (r, θ, z). It is essential to keep in mind the proper definitions of the various differential operators in this coordinate system.

For a scalar field f, the gradient is defined as

$$\nabla f \equiv \frac{\partial f}{\partial R}\hat{R} + \frac{\partial f}{\partial z}\hat{z} + \frac{1}{R}\frac{\partial f}{\partial \varphi}\hat{\varphi}.$$

We also introduce a "perpendicular gradient" operator,

$$\nabla_\perp f \equiv \frac{\partial f}{\partial R}\hat{R} + \frac{\partial f}{\partial z}\hat{z}.$$

The divergence of a vector $\mathbf{A}$ in cylindrical coordinates is

$$\nabla \cdot \mathbf{A} \equiv \frac{1}{R}\frac{\partial}{\partial R}(RA_R) + \frac{\partial A_z}{\partial z} + \frac{1}{R}\frac{\partial A_\varphi}{\partial \varphi}.$$

The Laplacian of a scalar is therefore

$$\nabla^2 f \equiv \nabla \cdot \nabla f = \frac{1}{R}\frac{\partial}{\partial R}\left(R\frac{\partial f}{\partial R}\right) + \frac{\partial^2 f}{\partial z^2} + \frac{1}{R^2}\frac{\partial^2 f}{\partial \varphi^2}.$$

The divergence of the perpendicular gradient is the new operator

$$\Delta^\dagger f \equiv \nabla \cdot \nabla_\perp f = \frac{1}{R}\frac{\partial}{\partial R}\left(R\frac{\partial f}{\partial R}\right) + \frac{\partial^2 f}{\partial z^2} \;=\; \frac{\partial^2 f}{\partial R^2} + \frac{1}{R}\frac{\partial f}{\partial R} + \frac{\partial^2 f}{\partial z^2}.$$

A new, divergence-like operator that will come in handy is

$$\nabla_\perp \cdot \mathbf{A} \;\equiv\; R\nabla \cdot \frac{\mathbf{A}_\perp}{R} \;=\; \frac{\partial A_R}{\partial R} + \frac{\partial A_z}{\partial z},$$

where $\mathbf{A}_\perp \equiv A_R\hat{R} + A_z\hat{z}$. If we apply this operator to the gradient or perpendicular gradient, we find that

$$\nabla_\perp^2 f \equiv \nabla_\perp \cdot \nabla_{(\perp)} f = R\nabla \cdot \left(\frac{1}{R}\nabla_\perp f\right) \;=\; \frac{\partial^2 f}{\partial R^2} + \frac{\partial^2 f}{\partial z^2}.$$

Another second-order differential operator that will arise in the course of the derivations below is

$$\Delta^\star f \equiv R^2\nabla \cdot \left(\frac{1}{R^2}\nabla_\perp f\right) \;=\; \frac{\partial^2 f}{\partial R^2} - \frac{1}{R}\frac{\partial f}{\partial R} + \frac{\partial^2 f}{\partial z^2}.$$

Finally, for completeness, we define the curl:

$$\nabla \times \mathbf{A} \equiv \left(\frac{\partial A_\varphi}{\partial z} - \frac{1}{R}\frac{\partial A_z}{\partial \varphi}\right)\hat{R} + \frac{1}{R}\left[\frac{\partial A_R}{\partial \varphi} - \frac{\partial}{\partial R}(RA_\varphi)\right]\hat{z} + \left(\frac{\partial A_z}{\partial R} - \frac{\partial A_R}{\partial z}\right)\hat{\varphi}.$$

1.2 The Scalar Variables

1.2.1 The Magnetic Field and Currents

We begin with the magnetic vector potential $\mathbf{A}$, where

$$\mathbf{B} = \nabla \times \mathbf{A}. \tag{1.1}$$

Since Faraday's law tells us that

$$\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E}, \tag{1.2}$$

it follows that

$$\frac{\partial \mathbf{A}}{\partial t} = -\mathbf{E} + \nabla\Phi \tag{1.3}$$

for any scalar field Φ. To specify $\mathbf{A}$, it is convenient to choose the gauge

$$\nabla_\perp \cdot \mathbf{A} \;=\; 0 \tag{1.4}$$

When we apply the $\nabla_\perp\cdot$ operator to (1.3), we then find that

$$\nabla_\perp^2 \Phi \;=\; \nabla_\perp \cdot \mathbf{E}. \tag{1.5}$$

It is evident that the gauge (1.4) places no restrictions on the toroidal component of $\mathbf{A}$, A_φ; the most general form of $\mathbf{A}$ satisfying (1.4) is therefore

$$\mathbf{A} \;=\; \nabla_\perp f \times \hat{\varphi} \;+\; \psi\nabla\varphi \tag{1.6}$$

where f and ψ are general scalar fields. Taking the curl of (1.6), we find that

$$\mathbf{B} \;=\; \nabla\psi \times \nabla\varphi \;+\; \frac{1}{R}\nabla_\perp\left(\frac{\partial f}{\partial\varphi}\right) \;-\; \nabla_\perp^2 f\hat{\varphi}. \tag{1.7}$$

If we define

$$F \;\equiv\; \frac{\partial f}{\partial\varphi} \tag{1.8}$$

and

$$I \;=\; -\frac{R}{R_0}\nabla_\perp^2 f \;=\; \frac{R}{R_0}B_\varphi \tag{1.9}$$

then the magnetic field can be written

$$\mathbf{B} \;=\; \nabla\psi \times \nabla\varphi \;+\; \frac{1}{R}\nabla_\perp F \;+\; R_0\, I\, \nabla\varphi. \tag{1.10}$$

Definition (1.9) is convenient because in the presence of a strong toroidal field $B_{\varphi_0} \equiv 1$ produced by TF coils, I can be written

$$I = 1 + \epsilon\tilde{I} \tag{1.11}$$

where ϵ is the inverse aspect ratio $a/R_0 \to 1/R_0$ when lengths are normalized to a, so that

$$\tilde{B}_\varphi = \epsilon\frac{R_0}{R}\tilde{I} = \frac{\tilde{I}}{R}$$

is the time-evolving component of B_φ due to poloidal currents in the plasma.

We can now express the current density in terms of these new scalars by applying Ampere's law,

$$\mathbf{J} = \nabla \times \mathbf{B} \tag{1.12}$$

to (1.10):

$$\mathbf{J} \;=\; -\Delta^\star\psi\nabla\varphi + \frac{1}{R^2}\nabla_\perp\psi' - \frac{1}{R}\nabla F' \times \nabla\varphi - \frac{1}{R}\frac{\partial F}{\partial z}\nabla\varphi + \nabla\tilde{I} \times \nabla\varphi \tag{1.13}$$

where the primes denote partial derivatives with respect to φ. If we represent the toroidal current density in terms of a new quantity

$$C \equiv -RJ_\varphi = \Delta^\star\psi + \frac{1}{R}\frac{\partial F}{\partial z} \tag{1.14}$$

then the current density can be written somewhat more compactly as

$$\mathbf{J} \;=\; \left(\nabla\tilde{I} - \frac{1}{R}\nabla_\perp F'\right) \times \nabla\varphi + \frac{1}{R^2}\nabla_\perp\psi' - C\nabla\varphi. \tag{1.15}$$

1.2.2 The Velocity Components

We expect the strong toroidal field to keep poloidal flows nearly incompressible. Perfect conservation of toroidal flux across a poloidal plane could be expressed as

$$\frac{\partial B_\varphi}{\partial t} + \nabla_\perp \cdot (B_\varphi \mathbf{v}_\perp) = 0. \tag{1.16}$$

For $\epsilon \ll 1$, the toroidal flux is dominated by the unchanging $R_0 B_{\varphi_0}/R$ component; thus to zeroth order in ϵ,

$$\nabla_\perp \cdot \frac{\mathbf{v}_\perp}{R} = 0.$$

In analogy with the derivation of the expression for the vector potential (1.6) arising from the gauge condition (1.4), we could then write

$$\frac{\mathbf{v}}{R} = \epsilon \nabla_\perp U \times \hat{\varphi} \ + \ v_\varphi \nabla \varphi$$

for some scalar functions U and v_φ; however to allow, more generally for *some* compressibility, we add an extra, compressible term to get

$$\mathbf{v} = R^2 \epsilon \nabla_\perp U \times \nabla \varphi \ + \ \nabla_\perp \chi \ + \ v_\varphi \hat{\varphi} \tag{1.17}$$

1.3 The Scalar Equations

Having expressed the vector quantities in terms of scalars, we now need equations for the time evolution of magnetic field quantities ψ and $\tilde{I}$ (F is related to $\tilde{I}$ by the elliptic equation $\nabla_\perp^2 F = -\tilde{I}'/R$); velocity quantities U, χ, and v_φ; and density ρ and pressure p.

1.3.1 Electromagnetic quantities

We begin with the poloidal flux function, ψ. From (1.6), it is clear that $\psi = RA_\varphi$; hence we can dot (1.3) with $R\hat{\varphi}$ to get

$$\frac{\partial \psi}{\partial t} = -RE_\varphi + \Phi' \tag{1.18}$$

This, of course, requires an expression for $\mathbf{E}$ (we can then invert (1.5) to find Φ). We use the resistive MHD Ohm's law,

$$\mathbf{E} + \mathbf{v} \times \mathbf{B} = \eta \mathbf{J} \tag{1.19}$$

where the resistivity $\eta(\mathbf{r})$ is a scalar. Dotting (1.19) with $-R\hat{\varphi}$, we find that

$$\begin{aligned} \frac{\partial \psi}{\partial t} \ &= \ \epsilon R (\nabla U \times \nabla \psi)_\varphi + \epsilon R \nabla_\perp F \cdot \nabla U - \nabla_\perp \chi \cdot \nabla \psi \\ &+ (\nabla_\perp \chi \times \nabla_\perp F)_\varphi + \eta C + \frac{\partial \Phi}{\partial \varphi} \end{aligned} \tag{1.20}$$

Because the application of a Laplacian operator on the sort of non-uniform mesh used by M3DP tends to introduce considerable numerical noise, while the inversion of such an operator does not, the present version of the code evolves not ψ but $\Delta^\star\psi$.

Next, we turn our attention to the fluctuating toroidal flux variable, $\tilde{I}$. Since $\partial\tilde{I}/\partial t = (1/\epsilon)\partial I/\partial t$, we can find it by dotting (1.2) with $R\hat{\varphi}$ and substituting in (1.10): $\partial\tilde{I}/\partial t = -R(\nabla\times\mathbf{E})_\varphi$, or, using (1.19),

$$\begin{aligned}\frac{\partial\tilde{I}}{\partial t} &= -\epsilon R^2\left(\nabla_\perp\tilde{I}\times\nabla_\perp U\right)\cdot\nabla\varphi - \left(\frac{1}{\epsilon}+\tilde{I}\right)\Delta^\star\chi - \nabla_\perp\tilde{I}\cdot\nabla_\perp\chi \\ &+ R^2\left[\nabla_\perp\left(\frac{v_\varphi}{R}\right)\times\nabla_\perp\psi\right]\cdot\nabla\varphi - \frac{v_\varphi}{R}\frac{\partial\tilde{I}}{\partial\varphi} + R\nabla_\perp F\cdot\nabla_\perp\left(\frac{v_\varphi}{R}\right) \\ &+ \nabla_\perp\eta\cdot\left(\nabla_\perp\tilde{I} - \frac{1}{R}\nabla_\perp F'\right) - (\nabla_\perp\eta\times\nabla_\perp\psi')\cdot\nabla\varphi \\ &+ \eta\left[\Delta^\star\tilde{I} - \frac{1}{R}\nabla_\perp^2 F' + \frac{2}{R^2}\left(\frac{\partial\psi'}{\partial z}+\frac{\partial F'}{\partial R}\right)\right].\end{aligned} \tag{1.21}$$

(1.20) and (1.21) are the two scalar equations for the time evolution of the magnetic field. They depend on auxiliary variables F and Φ, the first of which bears a simple relation to $\tilde{I}$. The equation for Φ, found by expanding (1.5) using (1.19), is

$$\begin{aligned}\nabla_\perp^2\Phi &= \epsilon\nabla_\perp\tilde{I}\cdot\nabla_\perp U + I\nabla_\perp^2 U - \frac{v_\varphi}{R}\Delta^\star\psi + \frac{1}{\epsilon R^2}\frac{\partial\chi}{\partial z} \\ &+ \left[\nabla_\perp\chi\times\nabla_\perp\left(\frac{\tilde{I}}{R}\right) - \nabla_\perp F\times\nabla_\perp\left(\frac{v_\varphi}{R}\right)\right]\cdot\hat{\varphi} - \frac{1}{R}\nabla_\perp v_\varphi\cdot\nabla\psi \\ &+ \frac{\eta}{R^2}\left[\frac{1}{R}\left(\frac{\partial F'}{\partial z}-\frac{\partial\psi'}{\partial R}\right) - \frac{\partial\tilde{I}}{\partial z} + \frac{\partial C}{\partial\varphi}\right] \\ &+ \left[\nabla_\perp\eta\times\left(\nabla_\perp\tilde{I} - \frac{1}{R}\nabla_\perp F'\right)\right]\cdot\nabla\varphi + \frac{1}{R^2}\nabla_\perp\eta\cdot\nabla_\perp\psi'.\end{aligned} \tag{1.22}$$

1.3.2 Velocities

The basic MHD momentum equation that will determine the behavior of the components of the velocity vector is

$$\rho\left(\frac{\partial\mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}\right) = \mathbf{J}\times\mathbf{B} - \nabla p + \mu\nabla^2\mathbf{v}, \tag{1.23}$$

where $\mu(\mathbf{r})$ is a scalar viscosity. It will be useful to define a new density variable

$$d \equiv R^2\rho; \tag{1.24}$$

in terms of d, the momentum equation is

$$\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v} \;=\; \frac{R^2}{d}\mathbf{J}\times\mathbf{B} - \frac{R^2}{d}\nabla p + \mu\frac{R^2}{d}\nabla^2\mathbf{v}. \tag{1.25}$$

The first scalar velocity component to investigate is the stream function U. We can extract it by operating on $\mathbf{v}$ with $-\nabla\varphi\cdot\nabla\times$:

$$-\frac{1}{\epsilon}\nabla\varphi\cdot(\nabla\times\mathbf{v}) = \Delta^\dagger U.$$

If we apply this same operator to (1.25), we find that

$$\frac{\partial}{\partial t}\Delta^\dagger U \;=\; \ldots$$

$$-\mathbf{B}\cdot\nabla\left(\frac{RJ_\varphi}{d}\right) + \mathbf{J}\cdot\nabla\left(\frac{1/\epsilon+\tilde{I}}{d}\right) + \frac{2}{\epsilon d}\frac{\partial p}{\partial z} - \frac{R}{\epsilon d^2}\hat{\varphi}\cdot(\nabla d\times\nabla p)$$

$$-\frac{1}{\epsilon}\nabla\varphi\cdot\nabla\times\left(\mu\frac{R^2}{d}\nabla^2\mathbf{v}\right) \tag{1.26}$$

The evolution of χ will be tracked in terms of its perpendicular gradient. First, it is clear from (1.17) that

$$\frac{\partial\chi}{\partial R} = v_R - \epsilon R\frac{\partial U}{\partial z}$$

so, making use of (1.25),

$$\frac{\partial}{\partial t}\left(\frac{\partial\chi}{\partial R}\right) \;=\; -\epsilon R\frac{\partial}{\partial z}\left(\frac{\partial U}{\partial t}\right) + \hat{R}\cdot\frac{\partial\mathbf{v}}{\partial t} \;=$$

$$-\epsilon R\frac{\partial}{\partial z}\left(\frac{\partial U}{\partial t}\right) - \mathbf{v}_\perp\cdot\nabla_\perp\left(\frac{\partial\chi}{\partial R} + \epsilon R\frac{\partial U}{\partial z}\right) - \epsilon v_\varphi\frac{\partial U'}{\partial z} - \frac{v_\varphi}{R}\frac{\partial\chi'}{\partial R} + \frac{v_\varphi^2}{R} - \frac{R^2}{d}\frac{\partial p}{\partial R}$$

$$+\frac{1}{d}\left(\frac{1}{\epsilon}+\tilde{I}\right)\left[\frac{1}{R}\left(\frac{\partial F'}{\partial R} + \frac{\partial\psi'}{\partial z}\right) - \frac{\partial\tilde{I}}{\partial R}\right] + \frac{C}{d}\left(\frac{\partial F}{\partial z} - \frac{\partial\psi}{\partial R}\right) + \hat{R}\cdot\mu\frac{R^2}{d}\nabla^2\mathbf{v} \tag{1.27a}$$

where

$$\mathbf{v}_\perp = \left(\epsilon R\frac{\partial U}{\partial z} + \frac{\partial\chi}{\partial R}\right)\hat{R} + \left(-\epsilon R\frac{\partial U}{\partial R} + \frac{\partial\chi}{\partial z}\right)\hat{z}$$

and

$$\hat{R}\cdot\nabla^2\mathbf{v} = \frac{\partial^2 v_R}{\partial R^2} + \frac{1}{R}\frac{\partial v_R}{\partial R} - \frac{v_R}{R^2} + \frac{\partial^2 v_R}{\partial z^2} + \frac{1}{R^2}\frac{\partial^2 v_R}{\partial\varphi^2} - \frac{2}{R^2}\frac{\partial v_\varphi}{\partial\varphi}.$$

Similarly,

$$\frac{\partial\chi}{\partial z} = \epsilon R\frac{\partial U}{\partial R} + v_z,$$

so

$$\frac{\partial}{\partial t}\left(\frac{\partial \chi}{\partial z}\right) = \epsilon R \frac{\partial}{\partial R}\left(\frac{\partial U}{\partial t}\right) + \hat{z}\cdot\frac{\partial \mathbf{v}}{\partial t} =$$

$$\epsilon R \frac{\partial}{\partial R}\left(\frac{\partial U}{\partial t}\right) - \mathbf{v}_\perp \cdot \nabla_\perp \left(\frac{\partial \chi}{\partial z} - \epsilon R \frac{\partial U}{\partial R}\right) + \epsilon v_\varphi \frac{\partial U'}{\partial R} - \frac{v_\varphi}{R}\frac{\partial \chi'}{\partial z} - \frac{R^2}{d}\frac{\partial p}{\partial z}$$

$$+\frac{1}{d}\left(\frac{1}{\epsilon} + \tilde{I}\right)\left[\frac{1}{R}\left(\frac{\partial F'}{\partial z} - \frac{\partial \psi'}{\partial R}\right) - \frac{\partial \tilde{I}}{\partial z}\right] - \frac{C}{d}\left(\frac{\partial F}{\partial R} + \frac{\partial \psi}{\partial z}\right) + \hat{z}\cdot\mu\frac{R^2}{d}\nabla^2\mathbf{v} \tag{1.27b}$$

where

$$\hat{z}\cdot\nabla^2\mathbf{v} = \nabla^2 v_z = \frac{\partial^2 v_z}{\partial R^2} + \frac{1}{R}\frac{\partial v_z}{\partial R} + \frac{\partial^2 v_z}{\partial z^2}.$$

To find out how v_φ evolves in time, we simply dot (1.25) with $\hat{\varphi}$:

$$\frac{\partial v_\varphi}{\partial t} = \epsilon R\left(\nabla_\perp U \times \nabla_\perp v_\varphi\right)\cdot\hat{\varphi} - \nabla_\perp\chi\cdot\nabla_\perp v_\varphi - \frac{v_\varphi}{R}\frac{\partial v_\varphi}{\partial \varphi} - \epsilon v_\varphi \frac{\partial U}{\partial z} - \frac{v_\varphi}{R}\frac{\partial \chi}{\partial R}$$

$$+\frac{1}{d}\left(\nabla_\perp \tilde{I}\times\nabla_\perp\psi\right)\cdot\hat{\varphi} + \frac{1}{d}\left(\nabla_\perp\tilde{I}\cdot\nabla_\perp F\right)$$

$$+\frac{1}{Rd}\left[\frac{\partial}{\partial\varphi}\left(\nabla_\perp\psi\times\nabla_\perp F\right)\right]\cdot\hat{\varphi} - \frac{1}{2Rd}\frac{\partial}{\partial\varphi}\left(|\nabla_\perp\psi|^2 + |\nabla_\perp F|^2\right)$$

$$-\frac{R}{d}\frac{\partial p}{\partial\varphi} + \mu\frac{R^2}{d}\left[\nabla^2 v_\varphi - \frac{v_\varphi}{R^2} + \frac{2}{R^2}\frac{\partial}{\partial\varphi}\left(\epsilon R\frac{\partial U}{\partial z} + \frac{\partial\chi}{\partial R}\right)\right] \tag{1.28}$$

1.3.3 Other Quantities

Density evolution is governed by the mass conservation (continuity) equation

$$\frac{\partial\rho}{\partial t} + \nabla\cdot(\rho\mathbf{v}) = 0, \tag{1.29}$$

or, in terms of d,

$$\frac{\partial d}{\partial t} = -R^2\nabla\cdot\left(\frac{d\mathbf{v}}{R^2}\right) \tag{1.30}$$

(1.30) can be expanded to give

$$\frac{\partial d}{\partial t} = -d\left(\Delta^\star\chi + \frac{1}{R}\frac{\partial v_\varphi}{\partial\varphi}\right) - \epsilon R^2\left(\nabla d\times\nabla_\perp U\right)\cdot\nabla\varphi - \nabla_\perp\chi\cdot\nabla d - \frac{v_\varphi}{R}\frac{\partial d}{\partial\varphi}. \tag{1.31}$$

Finally, in order to close the system of equations, we need an energy-conserving equation describing the evolution of the pressure in terms of the other variables. The form used by M3D is

$$\frac{\partial p}{\partial t} + \mathbf{v}\cdot\nabla p = -\gamma p\nabla\cdot\mathbf{v} + \rho\nabla\cdot\overleftrightarrow{\kappa}\cdot\nabla\frac{p}{\rho} \tag{1.32}$$

where γ is the ratio of specific heats $= 5/3$ for an adiabatic system, and

$$\overset{\leftrightarrow}{\kappa} \equiv \overset{\leftrightarrow}{\kappa}^{emu} \frac{1}{av_0}$$

is the thermal conductivity tensor. Expanding (1.32) in terms of the scalar components of $\mathbf{v}$ results in

$$\begin{aligned} \frac{\partial p}{\partial t} &= \epsilon R \left(\nabla_\perp U \times \nabla_\perp p\right) \cdot \hat{\varphi} - \nabla_\perp \chi \cdot \nabla_\perp p - \frac{v_\varphi}{R} \frac{\partial p}{\partial \varphi} \\ &-\gamma p \left(2\epsilon \frac{\partial U}{\partial z} + \Delta^\dagger \chi + \frac{1}{R} \frac{\partial v_\varphi}{\partial \varphi}\right) + \frac{d}{R^2} \nabla \cdot \overset{\leftrightarrow}{\kappa} \cdot \nabla \left(\frac{R^2 p}{d}\right) \end{aligned} \tag{1.33}$$

Chapter 2

Walkthrough

2.1 Running the Example

Let's walk through the execution of the first example on the T3E. For easier browsing, "level shifts" between functions are indicated in the margins.

```
/.../m3dp/mhd/driver% cp wout.test2 vmec4.dat
/.../m3dp/mhd/driver% cp wxylin wxy
```

The first of these shell commands creates a duplicate of the 455 kB ASCII Vmec output file to be used as input to **m3dp**. The second prepares an 87-line ASCII m3d input file that specifies some run parameters. Input parameters for **parm3d** will be read from the local configuration file **config.dat12**, which consists of the following:

```
8
4
30
1
4.712385
```

We next type

```
/.../m3dp/mhd/driver% ./run.12
```

to execute the following local script:

```
mpprun -n 12 \
./parm3d.x -vmecfile vmec4.dat \
-configFile config.dat12 \
-ksp_rtol 1.E-9 -ksp_type bcgs \
-pc_type asm -pc_asm_blocks 3 -pc_asm_overlap 3 -sub_pc_type ilu \
-sub_pc_ilu_levels 5 -betaFraction 1.0 -gamma 1.666667 \
-viscosity 1.E-3 -resistivity 1.E-7 \
-pModePerturbation 1 -sGridPerturbation 0.30 \
```

```
-sWidthPerturbation 0.2 -deltaU 1.E-4 \
-its 5500 -samplingT 2000 -NumRestart 500000 -timestep 0.01 \
-iwriteCheckPoint 200 -rescale -EngUpBound 1.E-4
```

"`mpprun -n 12`" specifies immediate execution of the remaining script on 12 PEs. **parm3d.x** is the name of the local executable. The rest of the script consists of command-line options passed to **parm3d.x**, which will in turn pass them to the Petsc initialization routine.

When the script is executed, **parm3d.x** will start up on 12 processors. Like all C codes, it begins with the function `main()`. The `main()` function is located in file **mhd/driver/test.c** in the m3dp distribution; it is the only function in this file. The first statement in the main block is a call to Petsc library function `PetscInitialize()`, passing any command-line arguments to that routine. →`main()`

There are then two calls to `PLogStageRegister()`, one of the suite of Petsc library routines used for performance profiling and declared in **petsclog.h**. A few calls to Petsc routine `OptionsGetInt()` (declared in **options.h**) follow, setting four local `int` variables based on command-line input options. The variables are

name	default value	specified value
`Iterations`	1	5500
`Sampling_rate`	1	2000
`NumRestart`	100,000	500,000
`iwriteCheckPoint`	1000	200

2.2 Reading the Data

The next statement is a call to function `initProfileVmec()`, which resides in file **mhd/init/initProfileVmec.c**. This function reads in an initial state from an ASCII Vmec output file, and distributes it over a mesh divided among all processes of the local MPI communicator. It also constructs a finite element matrix and solver. Let us consider these operations one at a time. ↓`initProfileVmec()`

`initProfileVmec()` first assigns default values to four elements of the variable pointed to by its first argument, `dev`. This variable is a structure of type `Configuration`, defined in **mesh/cpu/meshcpu.h**.

`dev` Element	Assigned value
`iNumGridsPhi`	1
`iNumCPUPhi`	1
`iNumRadialGrids`	30
`iNumRadialCPU`	1

Next, `initProfileVmec()` places a call to MPI library routine `MPI_Comm_rank()` to determine the rank of the local processor within the `MPI_COMM_WORLD` communicator. It checks the command line argument list for the presence of a "-vmecfile" option, and, finding it, reads the value "vmec4.dat" into the string

variable `vmecfile`. It then passes this string to function `vmecread()`, which is found in file **mesh/vmec/vmecread.c**.

`vmecread()` first checks its rank within the communicator once again. If it is the root process (rank 0), it opens the Vmec data file for reading. It reads in the first three lines of this ASCII file, prints some of the information to `stderr`, and performs basic error checking. Five of the newly read values are then broadcast to all PEs with MPI calls: ↓`vmecread()`

Variable	Value
`Data.iNumFieldPeriod`	1
`Data.iNumRadialGrids`	99
`Data.iMaxPoloidalModeNumber`	19
`Data.iMaxToroidalModeNumber`	0
`Data.iAccumModeNumber`	19

These values are all evidently elements of variable `Data` of type `struct VmecData`, defined in **mesh/vmec/vmec.h**. This variable will store the data read in from the Vmec file. Each processor uses the function `iVector1D()` (defined in **utility/utility.c**) to allocate space for two zero-indexed one-dimensional arrays of type `int` and size `Data.iAccumModeNumber` = 19, assigning them to elements `iPoloidalMode` and `iToroidalMode` of its local instance of variable `Data`.

Space for `Data`'s 11 2D arrays of type `double` is then allocated by analogous calls to **utility.c** routine `dVector2D()`. Each of these arrays will have `iNumRadialGrids` = 99 rows and `Data.iAccumModeNumber` = 19 columns. Finally, four calls to `dVector1D()` allocate storage for four 1D radial arrays of `doubles`, each of length 99.

The root process then reads all the relevant data from the Vmec file into the newly allocated space and closes the file. It shares all the data with all other processes *via* MPI broadcast calls, and `vmecread()` returns, with `Data` as its return value.

Back at the `initProfileVmec()` level, the routine checks for and fails to find the "-originIssue" flag, prompting no action. It then places a call to `vmecscaling()`, located in **mesh/vmec/vmecscaling.c**, passing it a reference to the newly-read vmec data structure. ↑`initProfileVmec()`

`vmecscaling()` first searches for the Fourier coefficient for R corresponding to the $0, 0$ mode at the zeroth radial grid and assigns this value to `data->Rmajor`, the absolute major radius. If more than one $0, 0$ mode is found, an error is returned. The routine next determines the average minor radius a by averaging over many poloidal planes in one toroidal field period. The aspect ratio and average toroidal field B_ϕ at the magnetic axis are calculated based on this information. Conversion factors to "mars" units are computed, and the Fourier coefficients and field quantities are renormalized to the minor radius and toroidal field respectively. The return value is 1. ↓`vmecscaling()`

`initProfileVmec()` then checks the option list for a configuration file name. Each processor then opens the file, **config.dat12**, essentially simultaneously, ↑`initProfileVmec()`

and reads in the following elements of `dev`, overwriting the previously assigned values:

Element	Value
`iNumGridsPhi`	8
`iNumCPUPhi`	4
`iNumRadialGrids`	30
`iNumRadialCPU`	1
`dalpha`	4.712385

Other `dev` elements are computed based on known information. The toroidal machine period `dev.iMachinePeriod` is set to `vmecData.inumFieldPeriod` = 1. The number of CPUs per poloidal section, `iNumCPUPoloidal`, is set to three times the square of the number of CPUs in the radial direction $= 3(1)^2 = 3$. The total number of grids `iTotalGrids` is three-halves the product of of the number of toroidal grids and `iNumRadialGrids` $*$ (`iNumRadialGrids` $+1$), or $(3/2)(8)(30)(31) = 11,160$. The total number of CPUs needed, `iTotalCPU`, is the product of the numbers in the poloidal and toroidal directions, $(3)(4) = 12$, which is, fortunately, the number actually being used.

2.3 Initializing the Mesh

2.3.1 Communicator Initialization

Once all this configuration information has been established, `ConstructComm()` is called to build an appropriate communicator to be assigned to global `Comm *` pointer `phone`. The `Comm` data type is a structure defined in **mesh/cpu/comm.h**; ↓`ConstructComm()` it consists of pointers to two MPI communicators, and five integers that keep track of the communicator sizes and the local process's rank within each of them.

The function `ConstructComm()` can be found in **mesh/cpu/constructComm.c**. It first checks for (and finds) agreement between the total number of CPUs required and the actual number in the current global communicator. Next, it makes two calls to **utility/utility.c** routine `MPI_Comm_vector()`, which allocates enough memory to hold two one-dimensional arrays of MPI communicators of sufficient size to hold the four poloidal (one for each toroidal section) and three toroidal (one for each poloidal section) communicators; `phone`'s pointers are assigned to the appropriate arrays. ↕`MPI_Comm_vector()`

Then, `setComm()` is called. Located in **mesh/cpu/setComm.c**, this function calls MPI library routines to fill in the newly allocated arrays by creating new MPI intracommunicators based on groups (of processors) consisting of appropriate subsets of the 12-member MPI group associated with the global communicator. [For more information on MPI groups and communicators, see http://www.netlib.org/utk/papers/mpi-book/node120.html in the Netlib repository.] ↕`setComm()`

Finally, some bookkeeping is performed to determine each processor's rank within its various communicators, and `ConstructComm()` returns. The next call is to function `MapCPUtoPartition()` in **mesh/cpu/mapCPUtoPartition.c**. The first two arguments dereference the pointers to the variables `dev` and `phone` described above; the third is a pointer to global variable `cell` of type `struct NodeDomain`, defined in **mesh/cpu/nodedomain.h**. A `NodeDomain` is a set of six `doubles` specifying upper and lower limits of toroidal angle ϕ, poloidal angle θ, and minor radius r (actually Vmec coordinate $s = r^2$) that defines a local chunk of the torus. `MapCPUtoPartition()` simply sets these limits (which will, in general, be different for each processor) based on the available geometry and communicator information.

↑`initProfileVmec()`

↕`MapCPUtoPartition()`

2.3.2 Index Initialization

`initProfileVmec()` then calls `indexMapping()`, assigning the return value to global variable `map` of type `struct IndexMapping`. This fairly complicated structure, defined in **mesh/index/indexMapping.h**, includes application ordering and index set mapping contexts whose types are defined in the Petsc library.

Function `indexMapping()`, whose return type is `IndexMapping`, is located in file **mesh/index/indexMapping.c**. In its first statement block, it sets the number of ghost toroidal sections on either side of the local cell to two. With a sequence of calls to short routines of the form `findNumXxxxYyyy()` also in files in the **mesh/index** subdirectory, it sets additional values in `map`:

↓`indexMapping()`

`map` Element	Value (process 0)
`NumGhostPhiSectionLow`	2
`NumGhostPhiSectionHigh`	2
`NumPhiSections`	2
`NumLocalVertices`	465
`NumGhostVertices`	60
`NumLocalElements`	958
`NumBoundaryVertices`	30

Local 1D `int` arrays `petscOrdering`, `aoOrdering`, and `local` are allocated with three calls to `iVector1D()`. Their sizes are `map.NumLocalVertices` ≈ 465, `map.NumLocalVertices`, and `map.NumLocalVertices` + `map.NumGhostVertices` ≈ 525 respectively.

Another call to `iVector1D()` allocates `map.indexBdyVertice`, an integer array that will store the local indices of all `map.NumBoundaryVertices` ≈ 30 boundary vertices, in counter-clockwise order. Also allocated here are `map.local`, which stores the indices of all local vertices in the application ordering; `map.ghosts`, which holds the indices of the ghost vertices in the Petsc ordering; and the 2D integer array `map.elements`, which will hold triplets of indices corresponding to the three vertices of each local triangular element.

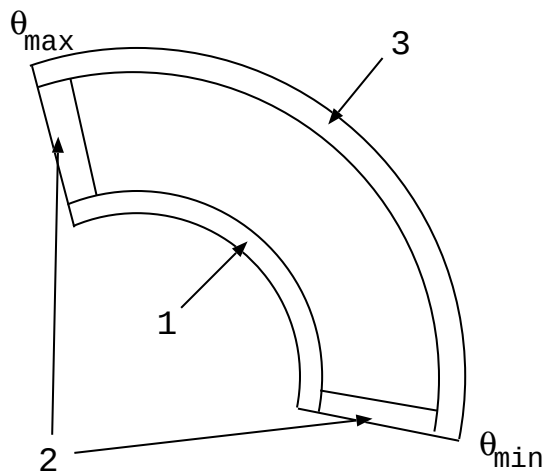

Figure 2.1: Schematic drawing of a local partition of a poloidal section of the torus, showing the order of assignment of ghost vertices.

This brings us to the first Petsc library call [aside from `PetscInitialize()` and various utility routines] encountered in the code. `VecCreateMPI()` is invoked here to create a parallel Petsc dummy vector distributed over all (three) CPUs within the calling process's poloidal communicator. The length of the local segment of the vector is `map.NumLocalVertices` ≈ 465; the length of the whole thing is left for Petsc to determine. A second Petsc call, `VecGetOwnershipRange()`, queries the new vector for its local lower and upper bounds, which are then used for an error check.

The bounds are then included in a call to `findOrdering()`, located in **mesh/index/findOrdering.c**. The purpose of this function is to initialize the `petscOrdering`, `aoOrdering`, and `map.ghosts` arrays in order to establish a correspondence between the application ordering of local and ghost vertices (the ordering corresponding to the grid topology) and the Petsc ordering (corresponding to the MPI topology), each of which assigns a unique index to each vertex across all processors. This added layer of complication has the enormous benefit of enabling Petsc to handle the mapping of ghost vertices to vertices residing on other processors automatically, making the parallelization transparent from the perspective of the author(s) of the routines that perform arithmetic operations on mesh variables. ↓`findOrdering()`

The first indices set in `findOrdering()` are those of the ghost grids bordering the cell at $r = r_{min}$ (Fig. 2.1), unless $r_{min} = 0$, in which case there are no ghost grids at this boundary. The ghosts are numbered using the application ordering: they start at

$$\frac{3(i-1)(i-2)}{2} + \frac{\theta_{min}}{\Delta\theta},$$

(where i is the radial index, starting at 1 for the innermost radial grid; θ_{min} is

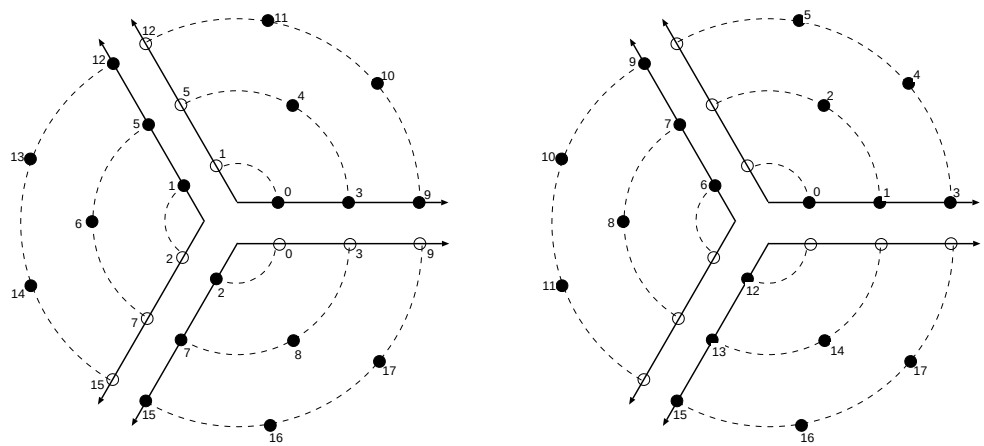

Figure 2.2: Application *vs* Petsc orderings of vertex indices in a poloidal plane for a sample run with three radial grids. Unfilled circles represent ghost vertices.

the low-θ boundary of the local cell; and $\Delta\theta = 2\pi/3i$ is the size in radians of the gap between successive θ grids at radius r_i) and proceed in increments of one with increasing θ. The result of this ordering is shown in Fig. 2.2a: vertices are numbered counter-clockwise around the poloidal plane and across processor boundaries, spiraling outward from the center.

The routine next enters a loop over radial grid locations i, from low to high. An inner loop over all θ within the mesh at r_i assigns indices of each vertex in the application ordering to successive elements of array `aoOrdering`, while assigning a separate set of indices (beginning with the lower bound of the local dummy vector and simply increasing by one with each assignment) to corresponding elements of `petscOrdering`. This ordering is thus characterized by a single continuous range of indices listing all vertices held by any one processor (Fig. 2.2b). Within the radial loop but outside the θ loop, additional ghost grid indices in the application ordering are added to `ghosts` from along the low and high θ boundaries of the cell.

Finally, back outside the radial loop, indices for ghost cells at the upper radial boundary are added to `ghosts`, and `findOrdering()` returns. ↑`indexMapping()`

`indexMapping()` next initializes `map.indexBdyVertice`. It has a -1th element of two less than the sum of the number of local and ghost vertices, $465 + 60 - 2 = 523$. Elements zero through `NumBoundaryVertices` $-1 = 29$ are numbered consecutively starting at `NumLocalVertices` $-$ `NumBoundaryVertices` $\approx 465 - 30 = 435$. Element 30 is initialized to $465 + 60 - 1 = 524$.

The next step is to copy the entire application-ordering index array `aoOrdering` into both `map.local` and the first `NumLocalVertices` ≈ 465 elements of `local`.

The rest of `local` is filled in with the ghost indices `map.ghosts`.

Another Petsc call is now placed. `ISLocaltoGlobalMappingCreate()`, as the name suggests, creates a context that will allow Petsc to map indices in local vectors and arrays (which generally include ghost values) to indices in the corresponding global distributed vectors and arrays using index sets. An index set (IS), as defined in the Petsc manual, is "a generalization of a set of integer indices, [which] is used to define scatters, gathers, and similar operations on vectors and matrices." The first invocation of the local-to-global mapping creation routine informs Petsc of the contents (local and ghost vertices) of `local`, and creates local index set mapping `LocaltoAO`. The routine is immediately invoked again, this time to pass the elements of `aoOrdering` to `map` structure element `map.ctx`.

Next, Petsc routine `AOCreateBasic()` is called to create a mapping between the elements of `aoOrdering` (the application ordering) and those of `petscOrdering`. The mapping is stored in `map.ao`. It is then used in a call to `AOApplicationtoPetsc()`, which has the effect of converting the indices stored in `local` from the application ordering to the Petsc ordering. The first 465 elements of `local` are simply mapped into their indices; i.e. `local[i]`→`low + i` for `i`∈{`0,...,NumLocalVertices-1`}. This is a consequence of that fact that, as was seen above in the discussion of `findOrdering()`, the interior vertices are numbered sequentially in the Petsc ordering. The rest of `local`'s indices refer to ghost vertices; their mapped Petsc ordering values refer to vertices that reside on other processors within the poloidal communicator. A final call to `ISLocaltoGlobalMappingCreate()` then stores this new local+ghost→Petsc mapping in `map.LGtoPetsc`.

As they are no longer needed, the dummy vector and integer index arrays are destroyed to free up memory. `dummy` is destroyed with Petsc routine `VecDestroy()`, while `petscOrdering`, `aoOrdering`, and `local` are destroyed with **utility/utility.c** routine `free_iVector1D()`.

With the vertex labels in place, the code next turns its attention to the triangular elements. A call is placed to the function `findElementOrdering()` in **mesh/index/findElementOrdering.c**, in order to set up the indices of the elements in the application ordering, analogous to `findOrdering()`'s construction of the vertex indices. ↓`findElementOrdering()`

After determining the number of radial grids (30 in the present case), the routine checks to see whether its bottom-most radial grid is at r_{min}. The present case has `iNumRadialCPU` = 1 as we've seen, so all the poloidal sections are wedge-shaped, and the $r = r_{min}$ condition is satisfied on all 12 processors. Accordingly, the first action taken is to assign vertices 0, 1, and 2, which encircle the origin in minor radius, to element 0 (Fig. 2.3, center). `findElementOrdering()` then loops through the remaining radial grids r_i, using `findThetaIndexRange()` in the **mesh/index** source file of the same name to determine the range of θ indices at each radius. The first block of statements within the radial loop works out the three elements associated with θ_{low}. In figure 2.3, these are elements 1–3 and 6–8. The pattern of each of these sets of three would be repeated for any further radial grids. Next, there is a loop over the remaining θ indices at radius r_i, assigning vertices to elements over the bulk of the partition; in the figure,

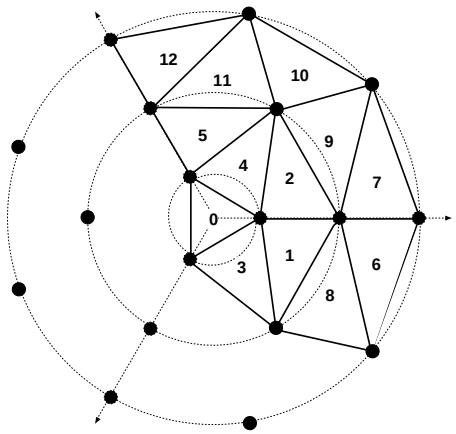

Figure 2.3: Element ordering for partition 0 for a sample case with three radial grids.

these are elements 4–5 and 9–12. Again, the pattern evident in these two sets is repeated for all further radii.

Unlike the vertex orderings, the element ordering assigned in this routine is purely local. That is, each processor will have its own definition of elements 0–12, etc., so that, with the exception of 0, a given element index will refer to different elements on different processors within a poloidal communicator. Thus the element numberings for partitions 1 and 2 are identical to that shown for partition 0 in Figure 2.3, but rotated by 120° and 240° respectively.

The rest of the radial loop in `findElementOrdering()` is devoted to correcting an error in vertex assignments that is made by the θ loop at the last two elements of the third partition. There is then a block that sets up ghost elements at the upper radial boundary of the partition if it does not coincide with the outer boundary of the poloidal section; since `iNumRadialCPU` = 1 in this case, it is skipped and the function returns.

`indexMapping()` now loops over the newly created elements, calling Petsc routine `ISGlobalToLocalMappingApply()` with the `LocaltoAO` mapping created earlier to convert the vertex indices from their global (application ordering) values to the corresponding values in the local ordering. The routine is called with the `IS_GTOLM_DROP` mapping type, which will not map vertices that fall outside the range of the local array; this is used for error-checking, as all assigned vertices *should* be in range. ↑`indexMapping()`

The next statement block is intended to map the number of symmetry-breaking vertices to their local indices, but since the macro `SWEEP` is undefined in this example, the preprocessor does not compile this block, and it is not executed. The local to global mapping index set `LocaltoAO` is destroyed to free

up memory, and the function returns the `IndexMapping map` it has created.

2.3.3 Geometry Initialization

Control reverts momentarily to `initProfileVmec()`, which checks the command line parameter list for the "`-vmecChease`" option and, failing to find it, calls `vmecmesh()` to scale down the 99-radial-grid Vmec data structure read in by `vmecread()` to the 30-radial-grid size that will be used in this run. ↑`initProfileVmec()`

Located in **mesh/vmec/vmecmesh.c**, `vmecmesh()` begins by creating a new `VmecData` structure, `mesh`, into which it then copies the basic size and field parameters from the old `VmecData` structure (which it calls `vmec`) that was read in from the Vmec data file **vmec4.dat**. The only value *not* copied from `vmec` is `iNumRadialGrids`, which is instead set to the value specified in the **config.dat12** file for this particular run, 30. ↓`vmecmesh()`

A series of **utility.c** routine calls is made to allocate space for the new structure's various arrays. A local 1D array, `sgrid`, is initialized to hold the grid's s coordinates, where $s = r^2$, and, since no mesh packing file was specified on the command line, `sgrid` is then copied directly into `mesh.sgrid`.

There follows a series of four calls to `vmecfit()` and one to `vmecfit0()`, both of which are in file **mesh/vmec/vmecfit.c**. Given two 2D Vmec meshes of size `iNumRadialGrids` by `iAccumModeNumber`, where the number of radial grids may be different, but the number of modes must be the same, `vmecfit()` maps the former onto the latter using a 5-point polynomial interpolation. It does this by making repeated calls to `polfit()`, a modified version of *Numerical Recipes* routine `polint()` residing in **mesh/vmec/polfit.c**. Once elements `RMNC`, `ZMNC`, `bsupvmn`, `currvmn`, and `Pressure` have been interpolated in this manner, `vmecmesh()` frees up the local `sgrid` variable and returns. ↕`vmecfit()` ↑`initProfileVmec()`

The downsized mesh is then immediately passed (along with pointers to `dev`, `cell`, `phone`, `map`, and `grid`) to `constructGrid()`, located in **mesh/grid/constructGrid.c**. This routine will convert the Vmec geometry information into a form that can be used by M3D.

`constructGrid()`'s first action is to allocate the `sgrid` element of `grid`, a global variable of type `struct Mesh` (defined in **mesh/grid/grid.h**). `sgrid` is a 1D array of doubles that will store the s value of each radial grid. ↓`constructGrid()`

Next, `grid.R` is allocated. Like most of the data-containing elements of `grid`, element `R` is of type `ParData`, defined in **mesh/index/pardata.h**. A `ParData` structure is set up to hold parallel distributed data; it consists of two arrays of Petsc vectors — one global and one ghosted local for each poloidal plane — that hold vertex indices, and a 2D `Scalar` array that holds the actual data:

```
typedef struct {
  Vec *g; /* global in a poloidal communicator, array in toroidal direction */
  Vec *l; /* ghosted local (ghosts accessed 'past the end' of the array),
             created via VecGhostLocalRepresentation,
             also ghosted in the toroidal direction, so l[] is of higher
```

```
                dimension (NumGhostLayer on both ends) than g[] */
    Scalar **array; /* array[whichPhiSection][vertex_index] */
  } ParData;
```

In order to set up this structure, `constructPData()` is called.

Located in **mesh/index/constructPData.c**, this routine is generic in the sense that it requires only communicator and index mapping information (and not, for example, information about the nature of the specific variable it is allocating) to set up a `ParData` structure. It first makes two calls to **utility/utility.c** routine `VecArray()` to allocate space for the global and local arrays of Petsc vectors within the structure. Because each processor spans two poloidal planes in the present instance, and because the (two) ghost planes as well as the actual planes must be included, the indices of these new arrays each range from -2 to 3. Space for an equal number, i.e. six, pointers of type `Scalar *` for the data array is allocated using `malloc()`. ↓`constructPData()`

The first non-ghost Petsc vector in the global vector array is initialized with a call to Petsc routine `VecCreateGhost()`; this routine is used to create vectors whose local representations contain storage for both local and ghost indices. Any other non-ghost Petsc vectors in the global array (there's just one in this case) are initialized by copying their parameters from the first one with Petsc routine `VecDuplicate()`. The ghost vectors in the global array are then initialized in the same way. `constructPData()` then returns, without having made any attempt either to initialize the Petsc vectors in the *local* array or to allocate space for the actual data in each poloidal plane. ↑`constructGrid()`

The remaining five non-gradient parallel data structures in `grid`, namely z, $dR/d\varphi$, $dz/d\varphi$, $d^2R/d\varphi^2$, and $d^2z/d\varphi^2$, are each created with a call to **mesh/index/duplicatePData.c** routine `duplicatePData()`, which allocates new arrays as in `constructPData()` but copies the array of global Petsc vectors from the one in `grid.R`.

If `map.NumBoundaryVertices` is greater than zero (and it is), space for `grid`'s 2D double arrays `NormPhi` ($\mathbf{n} \cdot \hat{\varphi}/\|\mathbf{n}_\perp\|$) and `dl` ($|\mathbf{r}_{i+1} - \mathbf{r}_i|$ on the boundary) and 3D double array `gradsBdy` (normalized ∇s on the boundary) is allocated with **utility.c** calls. Local equivalents to all these `grid` arrays are also allocated. Transfer of data can now begin.

The 30 values of s are copied from the Vmec mesh to `grid.sgrid`. The previously computed mapping `map.ctx` is used by Petsc routine `ISLocaltoGlobalMappingApply()` to create an array containing the global index of each local vertex. The boundary vertex indices are similarly mapped, first from the local to the Petsc ordering, then from that to the global application ordering. The list of vertices in the application ordering is duplicated, and the copy is converted to the Petsc ordering.

`constructGrid()` then enters a loop over the (two) local poloidal planes. Within each plane it computes the toroidal angle φ and then loops over all local vertices. The index of each vertex (in the application ordering) is passed to `mapIndexToVmecCoordinates()` in the file by that name in the **mesh/grid** subdirectory. This function computes and returns r and θ, the Vmec ra-

dial index and poloidal angle of the given vertex. These coordinates, along with φ and the Vmec data structure are then handed off to `vmecpoint2()` in **mesh/vmec/vmecpoint2.c**, which sums over the 19 available terms in Vmec's Fourier series representation to arrive at values for R, z, and their first and second derivatives with respect to φ at the given location. All these values are stored in the appropriate `Scalar` geometry arrays local to `constructGrid()`, and the inner loop proceeds to the next vertex. ↕`vmecpoint2()`

Once geometry information has been computed for all vertices in the current poloidal plane, it is copied from the local arrays to the global Petsc vectors in the corresponding elements of `grid` *via* the standard Petsc technique of first calling `VecSetValues()` with the array of values and the array of their indices in the Petsc ordering; and then calling `VecAssemblyBegin()` followed by `VecAssemblyEnd()` for the Petsc vector. This counterintuitive sequence of calls conceals a great deal of complexity; Petsc handles all the MPI communication that ensures that all values will end up on the appropriate processors. The beginning and end of the vector assembly process are kept separate in order to allow additional calculations to be performed while the communication is in progress, though this feature is not used here.

A similar operation is performed for the boundary vertices, but the only information that is kept after the Fourier sum is `NormPhi` and `gradsBdy`. These and the sum are computed by `vmecnorm()`, which is also in source file **constructGrid.c**. This concludes the outer loop. ↕`vmecnorm()`

After computing $\Delta\varphi$ for the local partition to be approximately $\pi/4$ radians (or 45°), `constructGrid()` places two calls to `GetArrayInPData()` in **mesh/index/getArrayInPData.c**. The purpose of these is to copy R and z data from the global Petsc vectors of their respective `ParData` structures to the local Petsc vectors and `Scalar` arrays, and to exchange data between the boundary and ghost cells of neighboring processors. Exchanges within a poloidal plane are handled by Petsc routines, while those between planes on different processors are handled "manually" by calling `ToroidalUpDate()`, which uses MPI calls directly and can be found in the same source file. ↕`GetArrayInPData()`

If, as in the present case, there are boundary vertices, the distances $d\ell$ between successive pairs are now computed. Storage for all local arrays is freed, and the function returns. ↑`initProfileVmec()`

2.3.4 Finite Element Initialization

Control is next transferred to routine `ConstructLocalFEMatrix()`, which receives the newly computed R and z values as arguments, along with references to global variables `phone`, `map`, and `lfe`. Located in **mesh/felement/constructLocalFEMatrix.c**, this function has the task of initializing the local finite element matrix. The matrix `lfe` is of type `struct LocalFE`, defined in **mesh/felement/felement.h**, and contains two elements. The first is `matrixInfo`, of type `struct MatrixAIJ` (defined in **mesh/index/matrixAIJ.h**). `matrixInfo` keeps track of the MPI communicator associated with the sparse finite element matrix as well as its size and the numbers of on- and off-diagonal non-zero elements in each row. ↓`ConstructLocalFEMatrix()`

The other element of `lfe` is `matrix`, a pointer of type `FEMatrix *`, also defined in **felement.h**. An `FEMatrix`, used for temporary storage on a local partition, consists of a 1D array of `double`s to keep track of the area Δ of each triangular element, a 2D array of `int`s to keep track of the connectivity between them, and the following 2D `double` arrays:

Name	Definition
`drdr`	$\int (\nabla_\perp \lambda_j \cdot \nabla_\perp \lambda_k)\lambda_i d^2x \;=\; \frac{d\mathbf{r}_\alpha \, d\mathbf{r}_\beta}{12\Delta}$
`mass`	$M_{ij} = \int \lambda_i \lambda_j d^2x$
`stiff`	$S_{ij} = -\int \nabla\lambda_i \cdot \nabla\lambda_j d^2x$
`dRoverR`	$\int \frac{\lambda_i}{R}\frac{\partial\lambda_j}{\partial R} d^2x$

Explanation For a triangular element with vertices at $\mathbf{r}_\alpha$, with $\alpha = 1, 2, 3$, the side segment vectors are defined as

$$d\mathbf{r}_\alpha \equiv \frac{1}{2}\sum_{\beta\gamma} \epsilon_{\alpha\beta\gamma}\,(\mathbf{r}_\beta - \mathbf{r}_\gamma) \tag{2.1}$$

i.e., $d\mathbf{r}_1 = \mathbf{r}_2 - \mathbf{r}_3$, etc., so the area may quickly be computed as

$$\Delta = \frac{1}{2} d\mathbf{r}_1 \times d\mathbf{r}_2 \cdot \hat{n}, \tag{2.2}$$

where $\hat{n}$ is the unit normal to the triangle.

The definitions of the mass matrix M_{ij} and stiffness matrix S_{ij} above are valid for any basis set of finite elements $\lambda_i(\mathbf{r})$. For the particular case of piecewise linear elements, as in the current implementation of this code,

$$\lambda_\alpha(\mathbf{r}) = \frac{1}{4\Delta}\sum_{\beta\neq\alpha}(\mathbf{r} - \mathbf{r}_\beta) \times d\mathbf{r}_\alpha \cdot \hat{n} \tag{2.3}$$

Each triangle has three basis functions, each of which takes the value unity at one vertex and decreases linearly to zero at the others. To put (2.3) more explicitly,

$$\begin{aligned}\lambda_\alpha(R, z) \;=\; & \frac{1}{4\Delta}\,[(z_{\alpha+1} - z_{\alpha+2})(2R - R_{\alpha+1} - R_{\alpha+2}) \\ & -(R_{\alpha+1} - R_{\alpha+2})(2z - z_{\alpha+1} - z_{\alpha+2})]\end{aligned} \tag{2.4}$$

where $\alpha + 1$ and $\alpha + 2$ are the indices of the other two vertices in the same triangular element as vertex α. Pairs of basis functions residing in different triangles will not contribute to the mass matrix, while those in the same triangle will contribute

$$\tilde{M}_{\alpha\beta} = \begin{cases} \Delta/6, & \alpha = \beta \\ \Delta/12, & \alpha \neq \beta \end{cases} \tag{2.5}$$

Similarly, the stiffness matrix contains only contributions from within triangles:

$$\tilde{S}_{\alpha\beta} = -\frac{d\mathbf{r}_\alpha \cdot d\mathbf{r}_\beta}{4\Delta} \tag{2.6}$$

For a more detailed discussion of these matrices, see Strauss and Longcope, *J. Comp. Phys.* **147**, 318–336 (1998). A particularly useful formula from that reference is

$$\int_\Delta \lambda_1^\ell \lambda_2^m \lambda_3^n \, d^2x \;=\; 2\Delta \frac{\ell!\, m!\, n!}{(\ell + m + n + 2)!}, \tag{2.7}$$

which instantly tells us, for example, that

$$\int_\Delta \lambda_i \, d^2x \;=\; \frac{\Delta}{3}$$

for i in Δ.

When `ConstructLocalFEMatrix()` is called, it first sets up the `matrixInfo` part of the local finite element matrix with a call to `MatrixAIJSetUp()` in **mesh/index/matrixAIJSetUp.c**. This function sets the MPI communicator of the matrix to the poloidal communicator of the local processor. It sets the linear size (number of rows = number of columns) of the matrix to `map.NumLocalVertices` $\approx$ 465. It then allocates space for the on- and off-diagonal number-of-non-zero-entries-per-row arrays and initializes them to zero. ↓`MatrixAIJSetUp()`

The actual numbers of non-zero entries are then tallied. Each such entry represents an interaction between vertices arising from an overlap of finite elements λ. The relative numbers of diagonal and off-diagonal entries will depend on whether the vertices of local triangles are themselves local or non-local to the processor. For each triangular element, the off-diagonal entry count for each local vertex is incremented by the number of neighboring vertices that are non-local (a maximum of two per element). The diagonal non-zero entry count for each local vertex is incremented by by the number of its neighbors that *are* local. Thus, the total number of non-zero entries per vertex is always incremented by two for each element it belongs to. This procedure results in an overcount of interactions since each edge joining a pair of non-boundary vertices is a part of exactly two triangles. The overcount is corrected by dividing each total by two. The number of diagonal non-zeroes is then incremented by one for each vertex to account for self-interactions. The boundary vertices are now undercounted and must be incremented accordingly. Finally, the new matrix info structure is returned. ↑`ConstructLocalFEMatrix()`

The matrices themselves are now allocated with a call to `AllocLocalFEMatrix()`, located in **mesh/felement/allocateLocalFEMatrix.c**. This routine allocates an array of `map.NumPhiSections` = 2 `FEMatrix` structures, one for each poloidal plane on the local processor. The `area` and `drdr` arrays of each new `FEMatrix` are allocated with `dVector`*n*`d()` **utility** calls. The remaining arrays are sparse (only non-zero elements are stored, with the number of such elements potentially varying from row to row) and must be allocated with specialized routines. The integer connectivity matrix is created and initialized to zero with a call to `iMatrix()`, located in **mesh/felement/iMatrix.c**, while the three remaining matrices of `doubles` are created by the analogous routine `fMatrix()`, residing in **fMatrix.c** in the same directory. ↓`AllocLocalFEMatrix()`

Once the two pieces of the local finite element matrix have been allocated, their values must be initialized. In order to do this, we must first get access to the local storage for the values, which will allow access to ghost values from neighboring processors. The function loops over the two local poloidal sections. With calls to Petsc routines `VecGhostUpdateBegin()` and `VecGhostUpdateEnd()`, it first updates the ghost values in the global vector pieces of the R and z `ParData` variables. `VecGhostGetLocalForm()` is then called for each of these vectors to determine its local form, which is stored in the local vector piece of the appropriate variable. The local vector shares the same array space and numerical values as the global vector, but allows access to ghost values past the end of the array. Elements of the local vector are accessed using the local numbering. ↑`ConstructLocalFEMatrix()`

Petsc routine `VecGetArray()` is now invoked to provide temporary direct access to the values stored in the local R and z vectors. Pointers to the values are passed to `AccumLocalFE()` in **mesh/felement/accumLocalFE.c**. This is the routine that will actually set the values. It consists primarily of an outer loop over all local triangular elements. ↓`AccumLocalFE()`

For each element, the three two-vectors $d\mathbf{r}_i$ are first computed. These are used to calculate the area of the element using formula (2.2); the area is then stored in the appropriate component of `lfe`. Next, the six components of the `drdr` matrix for the current element are computed using the formula given in the table on page 24. After this, another loop initializes each of the nine components of a local instance of the mass matrix for the current element to either $\Delta/6$ or $\Delta/12$ in accordance with equation (2.5). The same loop initializes a local stiffness matrix according to (2.6) and a local `dRoverR` matrix to

$$\texttt{dRoverR}_{\alpha\beta} \rightarrow \frac{\hat{z}\cdot d\mathbf{r}_\beta}{24}\left(\frac{1}{\hat{R}\cdot\mathbf{r}_\alpha}+\sum_{\gamma=1}^{3}\frac{1}{\hat{R}\cdot\mathbf{r}_\gamma}\right).$$

One more inner loop follows, incorporating the temporary local data just generated into the appropriate matrices of the permanent, global `lfe` structure. For each vertex in the current triangular element that is local to the processor, three calls are made to `UpdateMassStiff()`, located in the same source file as the calling function. `UpdateMassStiff()` is a simple three-line function that increments the element at a specified row and column of the mass, stiffness, and `dRoverR` matrices in a specified poloidal section of a specified finite element

matrix structure. Each is incremented by the value of a specified element of the corresponding local matrix.

The first call adds up self-interaction terms by specifying the addition of elements on the main diagonals of the three local matrices to column 0 (the main diagonal column in the sparse representation) of the row in the corresponding global matrix corresponding to the current vertex of the current element. The second call takes care of the interaction between the current vertex and the next one (local or not) in the current element in counterclockwise order. The column in the global sparse matrices to which *these* elements will be added is determined by a call to `findLocalColumnNum()`, also in the same source file. This versatile little function not only searches the connectivity matrix to determine the appropriate local column, but also *constructs* the connectivity matrix in the process. (Each row of this matrix corresponds to a local vertex. The first column in the row holds [for now] the number of other vertices to which it is currently connected; the rest of the columns give the indices of those vertices.) The third call to `UpdateMassStiff()` in the inner loop handles the interaction with the remaining vertex. This concludes the outer loop.

↕`UpdateMassStiff()`

↕`findLocalColumnNum()`

`AccumLocalFE()` now proceeds through a last loop over the local vertices, checking to make sure that the number each is connected to is one less than the sum of the number of nonzero diagonal and off-diagonal columns in the corresponding row of the sparse matrices. If a vertex fails this test, an error is returned. Otherwise, the vertex is connected to itself by replacing the connection count in the first column of its row in the connectivity matrix with its own index.

Finally, each process containing boundary vertices (in the present case, that includes all processes) calls `fixBdyLocalFE()` in the same source file to fix the matrices on those vertices. This function in turn works by making repeated calls to `fixTwoNeighbors()`, the last routine in **accumLocalFE.c**. Given the index of a "center" vertex and those of two of its neighbors, `fixTwoNeighbors()` determines the sparse matrix columns corresponding to each neighbor, and then simply returns, its only active statement block (which decrements the stiffness matrix entry relating the center vertex to the first neighbor by 1/2 and increments the entry for the second neighbor by the same amount) having been commented out. `fixBdyLocalFE()` calls `fixTwoNeighbors()` once for the first boundary vertex, once for the last one, and then once for each of the ones in between, but since this accomplishes nothing, it evidently should not be called at all, at least not in this version of the code.

↕`fixBdyLocalFE()`

Execution now goes up a level once more. Now that the local finite element matrices have been set up, it is necessary to release the pointers to the R and z vectors, so that Petsc will again be free to move the data around as it sees fit. This is accomplished with two calls to Petsc routine `VecRestoreArray()` and two to `VecGhostRestoreLocalForm()`. `constructLocalFEMatrix()` is now finished.

↑`constructLocalFEMatrix()`

↑`initProfileVmec()`

`initProfileVmec()` now immediately calls function `ConstructFEMatrix()` in **mesh/felement/constructFEMatrix.c**. Whereas the previous routine initialized the local matrix `lfe`, this one has the task of setting up global matrix `gfe` of type `struct FiniteElementMatrix`, defined in **mesh/felement/felement.h**.

This structure consists of three arrays of distributed Petsc matrices and nothing else:

```
typedef struct {
  Mat *Mass;
  Mat *Stiffness; /* implementation for ∇²⊥ */
  Mat *dRoverR;   /* (1/R) ∂/∂R */
} FiniteElementMatrix;
```

`ConstructFEMatrix()` first uses `malloc()` to allocate space for one matrix per poloidal section for each of the three components of `gfe`, for a total of six. It then loops over the two poloidal sections. Petsc routine `MatCreateMPIAIJ()` is called for the mass, stiffness, and `dRoverR` matrices in turn. "MatCreate" indicates that a Petsc matrix should be created; "MPI" indicates that it should be distributed over several processors; and "AIJ" indicates that it should be in the sparse AIJ format already described. The arguments passed to the routine tell it to use the same MPI communicator as the corresponding `lfe` variable; to have the same size locally as the `lfe` variable, with Petsc deciding on the global size; and to copy the number-of-non-zero-elements-per-row information from `lfe` as well. ↓`ConstructFEMatrix()`

Three calls to Petsc routine `MatSetLocalToGlobalMapping()` assign `map.LGtoPetsc` as the local-to-global mapping context for each new Petsc matrix. For each row within the current poloidal plane, three calls are made to Petsc routine `MatSetValuesLocal()`, telling Petsc that the values from that row of the `lfe` matrix should be inserted into the same row of the corresponding `gfe` matrix at column positions given by the values in that row of the connectivity matrix. Petsc now knows what to do, but it still must be instructed to do it; three pairs of calls to Petsc routines `MatAssemblyBegin()` and `MatAssemblyEnd()` actually make the value assignments.

The laboriously constructed local finite element matrices in `lfe` have now served their purpose and can be disposed of to free up memory. This is done by making three calls to `Free_fMatrix()`, a simple routine in **mesh/felement/fMatrix.c** that uses standard library routine `free()` to deallocate arrays. Assembly of the finite element matrices is now complete. ↑`initProfileVmec()`

2.4 Initializing the Physics

All the routines thus far have been concerned with numerics and geometry. They have painstakingly constructed a distributed triangular mesh with a shape dictated by the Vmec file, and have set up piecewise linear finite elements on it, but have had nothing to do with the physics of the problem. The next stage of initialization begins with a call to `constructMHDdata()` in **mhd/allocation/constructMHDdata.c**, whose return value, of type `struct MHD_DATA` (defined in **mhdData.h** in the same subdirectory), is assigned to global variable `userData`. The `MHD_DATA` structure is a colossal amalgam of elements (whose types, `MHD_xxxx`, are all defined in **mhdData.h**) that together hold *all* the information the code has

about the values of the physical variables, including all the variables that will be advanced on the mesh. First it must be allocated and initialized.

2.4.1 Mesh Allocation

Only communicator and index information is passed to `constructMHDdata()`; grid and finite element information are unnecessary for this routine. The declaration section includes the allocation of a local variable of type `struct MHD_DATA` called "`mhd`", which will eventually be used as the return value. The first executable statement allocates sub-element `mhd.coeff.Rinverse`, which is of type `struct RParData` (identical in form to `ParData`, and defined in the same header file), with a call to `constructRPData()`, located in **mesh/index/constructPData.c**. ↓`constructMHDdata()`

`ConstructRPData()` is simpler than `ConstructPData()`. It allocates space for the arrays of global and local Petsc vectors in the `RParData` structure with two calls to `VecArray()`; allocates space for a number of `Scalar *` pointers equal to the number of φ sections (2); calls Petsc routine `VecCreateGhost()` to set up a distributed local+ghost vector at the first element of the global vector array; and then uses `VecDuplicate()` to make copies of this vector at all other elements of this array. ↓`ConstructRPData()` ↑`constructMHDdata()`

The call to `ConstructRPData()` is followed by three calls to `duplicateRPData()` in **mesh/index/duplicatePData.c** for each of the three remaining `RParData` components of `mhd.coeff`: `dInverse`, `resistivity`, and `viscosity`. This has the effect of allocating space for the global and local vector arrays and the `Scalar *` array as in the construction call; and then using repeated calls to `VecDuplicate()` to make new Petsc vectors for each component of the global array. Calling `ConstructRPData()` for these elements instead would have had the same effect, but doing it this way avoids the communication overhead of the call to `VecCreateGhost()`.

The final element of `mhd.coeff` is `vphiDriving`, a Petsc vector. One more call to `VecDuplicate()` is sufficient to allocate it.

We now enter what the inline comments describe as the "first tier" of allocation. This statement block will allocate the space needed for the three elements of array `mhd.basic`, each of type `struct MHD_basic`. These structures will each hold one time-slice of the main physics data on the mesh, in the form of the nine variables $\tilde{I}$, $\Delta^*\psi$, $\Delta^\dagger U$, v_φ, $\Delta^*\chi$, d $(= R^2\rho)$, p, and artificial pressure and velocity, each of which is a `ParData` structure. The first variable initialized is the $\tilde{I}$ component of the first element of `mhd.basic`. This is done with a call to `constructPData()`, whose operation has already been discussed on page 22. In order to once again avoid needless interprocess communication, the rest of `mhd.basic`'s components will be copied from this first one with repeated calls to `duplicatePData()`, discussed above on p. 22. First, the $\tilde{I}$ components of the second and third elements of the `basic` array are copied. Then a loop copies all three $\tilde{I}$ frames to the three frames of each of the remaining variables.

The "second tier" of allocation creates the space for `mhd.extend`, of type `struct MHD_extend`, which will hold a single time-slice of auxiliary physics data in the form of the 13 `ParData` variables $\partial\tilde{I}/\partial R$, $\partial\tilde{I}/\partial z$, F, Φ, $C \equiv -RJ_\varphi$, δC

(the perturbed current), ψ, $\delta\psi$, $\delta\tilde{I}$, δp, χ, U, and $\nabla_\perp^2\Phi$. Each of these is constructed by copying it in turn from `mhd.basic[0].`$\tilde{I}$ with `duplicatePData()`.

The "third tier" allocates `mhd.dphis`, a structure of type `MHD_dphis` that holds the partial derivatives of ψ, F, $\tilde{I}$, U, χ, v_φ, and d with respect to φ in `RParData` structures. This requires a second call to `constructRPData()` followed by six more to `duplicateRPData()`.

The "fourth tier" allocates `mhd.dRdZs` of type `struct MHD_dRdZs`, which stores more auxiliary data: the partial derivatives with respect to R and z of ψ, F, U, and χ. `duplicateRPData()` suffices to construct these variables.

Additional blocks of duplication calls set up `mhd.work` of type `struct MHD_work`, three `ParData` objects that serve as a workspace; and `mhd.Rhs`, an array of two structures of type `MHD_RHS` that store right-hand-side values for multi-level explicit time-stepping in seven `RParData` variables.

The last component of `mhd` to be allocated by this routine is also the most complicated. `mhd.canvas`, a variable of type `struct MHD_Scratch`, provides scratch space for mathematical operations on MHD data. It consists of two `ParData` structures that will be used for temporary storage of intermediate quantities such as second derivatives with respect to φ, as well as two right-hand-side variables of type `struct TmpData` (defined in **mesh/index/tmpData.h** to be nothing more than a distributed Petsc vector and a local 1D array of `Scalars`) and a variable "`scratch`" of type `struct WorkSpace` (defined in **mesh/operator/workSpace.h** to consist of three `TmpData` structures and two Petsc vectors).

The two `TmpData` structures (`rhs` and `rhs_algebraic`) are first allocated with calls to `constructTmpData()`, located in **mesh/index/constructTmpData.c**, which simply uses Petsc routine `VecCreateMPI()` to set up a distributed Petsc vector in each one. The two `ParData` structures (`pDataR` and `pDataz`) are simply copied from the first $\tilde{I}$ in `mhd.basic` with calls to `duplicatePData()`. Each of the three `TmpData` structures in `scratch` is allocated with another call to `constructTmpData()`, while its two vectors are created with calls to `VecDuplicate()`. The array portions of the two `TmpData` structures in `mhd.canvas` and the three in `mhd.canvas.scratch` are each copied from their associated Petsc vectors with calls to `VecGetArray()`, following which those vectors are released again with five calls to `VecRestoreArray()`. The function then returns. ↕`constructTmpData()` ↑`initProfileVmec()`

2.4.2 Parameter Initialization

`initProfileVmec()` now begins to fill in the information in this new blank structure. First, it calls `setOptions()` in its own source file, **mhd/init/initProfileVmec.c**. This routine sets the parameter flag `userData.parameter.Flag` by using Petsc utility routines `OptionsGetDouble()` and `OptionsHasName()` to scan the command line for the following options: ↓`setOptions()`

Option	Result	Value added to Flag
`-resistivity`	$10^{-7} > 0$	`RESISTIVE_MHD`$\equiv 2^1$
`-viscosity`	$10^{-3} > 0$	`VISCOUS_FLUID`$\equiv 2^2$
`-noFlow`	not found	`FLOW_IS_NONE_VANISHING`$\equiv 2^3$
`-2drun`	not found	0
`-AdamMoulton`	not found	0
`-drop`	not found	0
`-rescale`	found	`RESCALING`$\equiv 2^{11}$
`-cylinder`	not found	0
`-varyViscosity`	not found	0
`-varyResistivity`	not found	0
`-nonlinear`	not found	0

As a result, the options flag is set to $2^1 + 2^2 + 2^3 + 2^{11} = 2,062$.

On the return of execution to `initProfileVmec()`, further parameter values are set. The major radius, aspect ratio, toroidal field, number of field periods, and number of radial grids are copied into `userData.parameter` from the Vmec mesh that was initialized earlier. The "`gamma`" (ratio of specific heats) parameter is set to 5/3. The following command line options are read with `OptionsGetDouble()`, and their values are assigned to the `parameter` elements with the same name: ↑`initProfileVmec()`

Option	Value
`-timestep`	10^{-2}
`-viscosity`	10^{-3}
`-resistivity`	10^{-7}
`-gamma`	1.666667

`parameter.timestep_target` is also set to `timestep` $= 0.01$, while `parameter.timestep1` and `parameter.timestep2` are set to zero.

2.4.3 Field Initialization

The magnetic field and pressure profiles computed by Vmec must now be set up on the mesh. This is accomplished by calling `setupIJPonGrid()`, located in **mhd/init/setupIJPonGrid.c**. The function begins by reading in more options from the command line: ↓`setupIJPonGrid()`

Option	Value
`-vmecChease`	Absent
`-betaFraction`	1.0
`-sGridCoeff`	Absent (default = 1.0)
`-sWidthCoeff`	Absent (default = 0.01)

It then allocates two local 1D arrays of integers, `in` and `out`, using `iVector1D()`, giving each a size equal to the number of local vertices. Local 1D scalar arrays `bsupvmn`, `currvmn`, `pressure`, `density`, and `coeff` are allocated to the same size with calls to `malloc()`.

The elements of `in` are initialized to the indices of the vertices in the local ordering with the simple mapping $\texttt{in}_i \to i$. The equivalent list in the application ordering is then found by passing this array to `ISLocaltoGlobalMappingApply()`; the result is stored in `out`. It is then copied back into `in`, which is then converted from the application to the Petsc ordering with a call to `AOApplicationToPetsc()`. The three `MHD_basic` pointers in `userData` — `old`, `new`, and `base` — are set to point respectively to the first, second, and third elements of its `MHD_basic` array, `basic[]`.

We now enter the main body of the routine, which consists of a loop over (two) local φ sections. The value of φ for the current section is computed. Then comes a loop over local vertices. The application index $\texttt{out}_i$ of the vertex is passed to `mapIndexToVmecCoordinates()` so that its Vmec position (r, θ) can be calculated. This information is then fed into `vmecBfield()` (in **mesh/vmec/vmecBfield.c**) along with φ, the existing Vmec mesh data, and the addresses of the elements of `bsupvmn`, `currvmn`, `pressure`, and `density` corresponding to the vertex. `vmecBfield()` computes the appropriate values of `bsupvmn` and `currvmn` by summing over the 19 terms in the Vmec Fourier series expansions for those variables: ↓`vmecBfield()`

$$B(r, \theta, \varphi) = \sum_{j=0}^{18} B_j(r) \cos(m_j \theta - n_j \varphi)$$

`vmecBfield()` then sets the pressure equal to the Vmec structure's pressure at the given radius, sets the density to 1.0, and returns. The value of `coeff` at the current vertex is computed as ↑`setupIJPonGrid()`

$$\texttt{coeff}_i \to 1 + \frac{0.01}{0.01^2 + [s(r_i) - 1]^2},$$

concluding the loop over vertices.

The values in `bsupvmn` are now copied into the Petsc vector storing the values of $\tilde{I}$ in the current poloidal plane for the `userData` `MHD_Basic` structure pointed to by `userData.old`. As usual, this requires calls to `VecSetValues()`, `VecAssemblyBegin()`, and `VecAssemblyEnd()`. Because the "`chease`" flag was not set on the command line, the values in this vector are multiplied by the squares of their major radial positions with two calls to Petsc arithmetic routine `VecPointwiseMult()`. Petsc routine `VecShift()` is then called to subtract the aspect ratio (= 1.6) from each of these values.

Petsc vector $C \equiv -RJ_\varphi$ in `userData.extend` is now copied from the `currvmn` array in the same manner as the previous copy. The absence of the "`chease`" flag mandates the multiplication of each value by the local major radius, following which Petsc routine `VecScale()` is called to invert their signs. The pressure profile is copied into `userData.old->p` and scaled by `betaFraction` = 1.0.

The routine now checks whether the `VARY_VISCOSITY` bit is set in the parameter flag. It isn't, so all elements of the `userData.coeff.viscosity` vector are set to the constant `userData.parameter.viscosity` $= 10^{-3}$. They are then

scaled by the square of the inverse aspect ratio, $(1/1.6)^2 = 0.390625$. A similar process is used to set the resistivity; since the flag is not set, it gets the constant value `userData.parameter.resistivity` $= 10^{-7}$. It is not rescaled.

The final statement block in the main loop copies the density data from `density` to `userData.old->d`. It multiplies each value by the square of the local major radius, and then divides them all by the square of the aspect ratio. After the main loop, `setupIJPonGrid()` simply frees up its local arrays and returns. ↑`initProfileVmec()`

2.4.4 Solver Initialization

The solver is a global variable called "`userSolver`", of type `struct MHD_Solver`, defined in **mhd/allocation/mhdSolver.h**. The structure consists of a number of sparse Petsc matrices, index sets, and `Scalar` and integer arrays as well as 13 pointers of type "`SLES`", a linear system solver context defined by Petsc to provide an interface to its linear algebra package. `userSolver` is allocated by setting it equal to the return value of function `constructMHDsolver()`.

Situated in **mhd/allocation/constructMHDsolver.c**, this routine begins by calling Petsc routine `PetscMalloc()` twice, to allocate space for the two Petsc matrix array components of the solver, `starMatrix` and `daggerMatrix`, each of which contains as many Petsc matrices as there are poloidal sections in the local processor, i.e., two. If there are boundary vertices (and there are), this will require the allocation of special structures within the solver to deal with them. 1D integer arrays `bdyRowIndex` and `bdyColumns` are allocated with calls to `iVector1D()`. The former stores the row index of each boundary vertex in the Petsc ordering, while the latter specifies the number of non-zero columns in each row, i.e., the number of vertices connected to each boundary vertex. `bdyColumnIndex`, an irregular 2D integer array that will hold the column indices for each boundary row, and `bdyMatEntree`, an irregular 2D `Scalar` array that will store the actual matrix entries for boundary terms, are allocated next with `malloc()`. The values of `bdyRowIndex` are initialized by invoking Petsc routine `ISLocaltoGlobalMappingApply()` to convert the local ordering of the boundary vertices stored in `map.indexBdyVertice` to the Petsc ordering. ↓`constructMHDsolver()`

To initialize the boundary vertex matrix entries, the routine now loops over these vertices. For each one, it reads out the column indices and entries of the corresponding row in the global finite element mass matrix with a call to the Petsc read-only function `MatGetRow()`. (This is one of those functions that also has the effect of temporarily "locking" the matrix data from tampering by Petsc; it must be "unlocked" afterward if further Petsc operations are to be performed on the matrix.) The number of entries in the row is now known and can be used to allocate space in the current rows of `bdyColumnIndex` and `bdyMatEntree`; the just-read values are then copied into these rows. The Petsc matrix is "unlocked" with a call to `MatRestoreRow()`, and the loop continues. The completion of the vertex loop marks the end of the special case for boundary vertices.

We now proceed to the main solver initialization. The first process in each poloidal communicator contains a special structure in the solver, a sin-

Table 2.1: The MHD solver's `SLES` arrays

Name	Purpose
`massInverter`	Invert the mass matrix
`FInverter`	Invert $\nabla_\perp^2 F$ to find F
`psiInverter`	Invert $\Delta^* \psi$ to find ψ
`PhiInverter`	Invert $\nabla_\perp^2 \Phi$ to find Φ
`UInverter`	Invert $\Delta^\dagger U$ to find U
`chiInverter`	Invert $\Delta^* \chi$ to find χ
`tIStepper`	Step $\tilde{I}$
`psiStepper`	Step ψ
`UStepper`	Step U
`vphiStepper`	Step v_φ
`chiStepper`	Step χ
`dStepper`	Step $\mathrm{d} = R^2 \rho$
`pStepper`	Step pressure

gle "reference" row of matrix entries. Its indices are stored in 1D integer array `refColumnIndex`, and its values are stored in 1D `Scalar` array `refMatEntree`. The process calls `MatGetRow()` again to read out the first row in the mass matrix; this is then used to allocate and initialize the reference row. The process then restores the row with `MatRestoreRow()`.

All processes now allocate space for their 13 `SLES` arrays (see Table 2.1) with calls to `PetscMalloc()`. Petsc routine `MatDuplicate()` is invoked to set up the solver's Petsc matrix `lhsMatrix` by copying the structure, but not the values, of the finite element mass matrix. A loop over local poloidal sections begins, to initialize the new solver contexts.

The first solver pieces initialized in each iteration of the loop are `starMatrix` and `daggerMatrix`. Each of these is created by setting it equal to the stiffness matrix for the current poloidal section with Petsc matrix operation `MatConvert()`. The `dRoverR` matrix is then subtracted from the resulting `starMatrix` and added to `daggerMatrix`; this conforms to the definitions

$$\Delta^* \equiv \frac{\partial^2}{\partial R^2} - \frac{1}{R}\frac{\partial}{\partial R} + \ldots; \quad \Delta^\dagger \equiv \frac{\partial^2}{\partial R^2} + \frac{1}{R}\frac{\partial}{\partial R} + \ldots.$$

The procedure for setting up each of the 13 `SLES`s follows a similar pattern. Each begins with a call to Petsc command `SLESCreate()`, which induces Petsc to allocate space for a `SLES` structure distributed over a specified MPI communicator, in this case, the local poloidal communicator. After some optional calls to set up the matrices associated with the linear system and the solution options, construction of each `SLES` is completed with a call to Petsc routine `SLESSetFromOptions()`. For the first `SLES` in the solver, `massInverter`, the first intermediate call sets up the matrices associated with the linear system by calling Petsc routine `SLESSetOperators()`, informing Petsc that the global

finite element mass matrix is to be used both to define the linear system and as the preconditioner matrix, and further, that the preconditioner matrix will not change between successive linear solves. Petsc uses a combination of a Krylov subspace method and a preconditioner to perform iterative linear solves [for a discussion of such methods, the Petsc manual refers the reader to Freund, Golub, and Nachtigal, *Iterative Solution of Linear Systems*, pp. 57–100, Acta Numerica, Cambridge University Press, 1992]. In order to customize the Krylov method for the present (mass inverter) system, the Krylov context for the new `SLES` must be extracted by calling `SLESGetKSP()`. This context is then used to adjust the tolerances that will be used in the convergence check for the iterative solution. In particular, the routine calls `KSPSetTolerances()` to instruct Petsc to consider a solution to have converged if the l_2-norm of the residual $r_k \equiv b - Ax_k$ is less than either 10^{-10} of the l_2-norm of the initial residual r_0, or 10^{-50} in absolute value, whichever is greater. The criterion for the *divergence* test and the maximum allowable number of iterations are not adjusted.

The `FInverter` initialization makes the same Krylov solver adjustments but additionally invokes `KSPSetInitialGuessNonzero()` to keep the solver from using zero as its initial guess for the solution vector. No call is made to `SLESSetOperators()` for `FInverter`, so it does not yet have an associated matrix. Inverters `psiInverter`, `PhiInverter`, `UInverter`, and `chiInverter` receive identical treatment, as do steppers `psiStepper`, `UStepper`, `vphiStepper`, and `dStepper`. The remaining `SLES`s, `tIStepper`, `chiStepper`, and `pStepper`, are initialized without any matrices *or* Krylov option adjustments.

Following the loop over poloidal sections come two calls to Petsc routine `ISCreateGeneral()`. The first of these initializes `userSolver` component `isInterior` to be a Petsc index set containing the single index "0". The second sets up `isBdy` as an index set containing the row indices of all boundary vertices. This concludes the solver construction, and the function returns. ↑`initProfileVmec()`

The final act of the profile initialization routine is to construct the `lmass` `ParData` structure in `userData.work`. The function `ConstructLmass()`, in **mhd/allocation/constructLmass.c** accomplishes this. For each φ section, ↓`ConstructLmass()` it sets all elements of the `userData.canvas.scratch.tmpData.g` Petsc vector to unity; multiplies this vector from the left by the mass matrix; and stores the resulting vector — each of whose elements is the sum of the elements in the corresponding row of the mass matrix — in the appropriate poloidal section of the global Petsc vector of `userData.work.lmass`. The data are then shared with the other parts of `lmass` using `GetArrayInPData()`. ↑`initProfileVmec()`

A comment in the source file notes that some memory should be freed up at this point. However, in the present version of the code, this is not done. Profile initialization is complete, and the function returns. ↑`main()`

2.5 Preparing to Iterate

The driver that causes the physics routines to advance the mesh by one time step is `timeadv()`, discussed in detail below. The first half of this routine solves

the time-dependent equations, while the second half solves elliptic equations to invert the flux and stream functions for use in the following time step. Consequently, on each entry to the `timeadv()` routine, it will expect these functions to already have been computed for the previous time step. In order to prepare for the first call to `timeadv()`, the code must therefore do the flux and stream function inversions once based on the initial profile. This is done by calling the routine `initMHD()`.

Located in **mhd/init/initMHD.c**, `initMHD()` is similar to the (unused) second half of `timeadv()`. It begins by calling Petsc routine `VecGetArray()` twice in order to point the 2D `Scalar` array pointers in `userData.canvas.rhs` and `userData.canvas.rhs_algebraic` to the data contained in their respective global Petsc vectors. It then immediately calls `VecRestoreArray()` for each of these vectors, passing it a dummy pointer instead of the one assigned by the previous call. The intention appears to be to "copy" the contents of the Petsc vectors to the accessible arrays, while freeing Petsc to rearrange them as necessary afterwards, but this is not the effect, and appears to be a dangerous misuse of the routines; the space pointed to by the array pointers could conceivably be deallocated before the pointers are next dereferenced, resulting in a segmentation fault.[1] A series of calls to `GetArrayInPData()` follows, using the same `GetArray/RestoreArray` trick to transfer array information for R, z, and their derivatives with respect to φ. The difference between this and the transfer in `userData.canvas` is that here, the ghost cells will also be updated prior to data "copying". ↓`initMHD()`

The command line is next searched for the `-initProfile` flag and the specification of a checkpoint file containing equilibrium data. The flag is not found, so no file is read.

In preparation for solving the elliptic equations, certain of the velocity values are zeroed out with a call to `zeroFlowField()`, in **mhd/init/zeroFlowField.c**. This routine loops through the local φ sections, using Petsc routine `VecSet()` to zero the global Petsc vectors storing velocity-related variables $\Delta^\dagger U$, v_φ, and $\Delta^\star\chi$ in `userData.old`; the vectors storing χ and U in `userData.extend`; and the auxiliary `userData` vectors storing the first derivatives of χ and U with respect to R, z, and φ and of v_φ with respect to φ. It then calls `GetArrayInPData()` or `GetArrayInRPData()` as appropriate to provide direct array access to each of these newly zeroed vectors. ↓`zeroFlowField()` ↑`initMHD()`

In yet another loop over poloidal sections, `initMHD()` now repeatedly invokes Petsc routine `VecCopy()` to duplicate the Petsc vectors storing $\tilde{I}$, $\Delta^\star\psi$, $\Delta^\dagger U$, v_φ, $\Delta^\star\chi$, d, and p in `userData.old`, storing the copies in `userData.base`. Access to the copies is then provided with several calls to `GetArrayInPData()`.

[1] According to inline comments elsewhere in the code, however, this "trick" was suggested by Barry Smith, one of the Petsc authors, so it is likely to be compatible, if not with the Petsc specification, then at least with its current implementation.

2.5.1 Toroidal Derivatives

At this point, the code performs its first real operation on the physical data: it calls `dXdphi()`, located in **mhd/operator/dXdphi.c**, to compute $\partial d/\partial\varphi$. Because there is the potential for the mesh received from Vmec to "twist" with changing φ in order to follow the field lines, the actual quantity that must be computed is

$$\frac{d\mathtt{d}}{d\varphi} = \frac{\partial d}{\partial R}\frac{\partial R}{\partial \varphi} + \frac{\partial d}{\partial z}\frac{\partial z}{\partial \varphi} + \frac{\partial d}{\partial \varphi} \tag{2.8}$$

The last term in this expression might be computed by ordinary finite differencing, but the use of finite elements makes the evaluation of the first two more complicated. The code solves the problem by moving the explicit partial derivative in (2.8) to the left-hand-side and applying the Galerkin method: it multiplies both sides by each basis function, and then integrates over the poloidal plane:

$$\int \left(\frac{d\mathtt{d}}{d\varphi} - \frac{\partial d}{\partial \varphi}\right)\lambda_i d^2x \;=\; \int \left(\frac{\partial d}{\partial R}\frac{\partial R}{\partial \varphi} + \frac{\partial d}{\partial z}\frac{\partial z}{\partial \varphi}\right)\lambda_i d^2x \tag{2.9}$$

The left-hand-side of (2.9) is by definition the vector product $M_{ij}\delta_j$, where M is the mass matrix and δ is the vector of values of the quantity in parentheses at each vertex. Once the right-hand-side vector has been evaluated explicitly, the equation can be inverted to find δ, to which $\partial d/\partial\varphi$ will then be added to get the desired quantity.

The function `dXdphi()` encloses its statements in the now-familiar loop over φ sections. Within the loop, its first act for each section is to zero the right-hand-side vectors in the scratch space provided to it by the calling routine. These right-hand-sides are now passed, along with the vector being differenced, to function `dPhi()`, located in **mesh/operator/dPhi.c**. Its job will be to calculate the right-hand-side vector of (2.9) and the "algebraic" right-hand side $\partial d/\partial\varphi$. ↓`dXdphi()`

`dPhi()` is set up to calculate its finite differences only for 3D runs; in the 2D special case, the right-hand side is, of course, identically zero. It determines whether or not the current run is 3D by checking the extent of the local mesh in toroidal angle. If the extent is less than 2π, the run is 3D and the differencing is performed. The present run *is* 3D, with a φ span of $\pi/4$ per toroidal segment, so the function proceeds. ↓`dPhi()`

To compute the first term on the right-hand side, it calls `xDR()`, located in file **mesh/operator/dR.c**, passing it the `dRdPhi` and z values for each vertex in the local mesh, along with the `ParData` variable to be differenced and space to store the result. (`dRdPhi` is the vector of $\partial R/\partial\varphi$ values that were precomputed by function `vmecpoint2()` back on page 23.) `xDR()` is merely a wrapper; it extracts the appropriate φ section from each of the three vertex arrays it is given, and passes them on to `xDR0()` in the same source file. ↓`xDR()`

It is down at the `xDR0()` level that actual floating-point operations are finally performed on the data. The quantity to be computed is ↓`xDR0()`

$$\int \frac{\partial d}{\partial R}\frac{\partial R}{\partial \varphi}\lambda_i d^2x.$$

Since

$$d(\mathbf{x}) = \sum_j d_j \lambda_j(\mathbf{x}),$$

this can be rewritten as

$$\sum_j d_j \int \frac{\partial \lambda_j}{\partial R}\frac{\partial R}{\partial \varphi}\lambda_i d^2x \;=\; \frac{1}{2}\sum_j \left[\frac{d_j(z_{j+1}-z_{j+2})}{\Delta}\int \frac{\partial R}{\partial \varphi}\lambda_i d^2x\right]$$

where Δ is the area of the triangle containing basis function j and the subscripts on z are understood to be modulo three, so that they always refer to the vertices one increment clockwise.and one increment counterclockwise from the one referred to by the subscript on d within the same triangular element. The integral can be evaluated by assuming that $\frac{\partial R}{\partial \varphi}$ varies linearly between vertices so that the known values at the vertices may be taken as coefficients in an expansion over linear basis functions. In this case,

$$\int R'\lambda_i d^2x \;=\; \sum_k R'_k \int \lambda_k \lambda_i d^2x$$

The value of *this* integral, according to (2.7), is simply

$$I_{ki} = 2\Delta \begin{cases} 1/12, & k = i \\ 1/24, & k \neq i, \text{ but in same triangle} \\ 0, & \text{otherwise} \end{cases} \tag{2.10}$$

so that the ith component of the vector we are interested in is

$$\left(\frac{R'_i}{12} + \frac{R'_{i+1}}{24} + \frac{R'_{i+2}}{24}\right)\sum_j d_j(z_{j+1}-z_{j+2})$$

This is what the routine computes.

For efficiency, it loops over local triangular elements. For each triangle, it begins by computing the sum over the three vertices of the quantity

$$d_j(z_{j+1}-z_{j+2})$$

where d_j are the values taken on by the density variable being differenced at each vertex. A second loop over the same vertices is then begun. For each one that is local to the processor, the ith value of the right-hand-side array is incremented by the product

$$-\sigma\left(\frac{x_i}{12} + \frac{x_{i+1}}{24} + \frac{x_{i+2}}{24}\right)$$

where σ is the sum computed in the first vertex loop and the x_i are the values of the `dRdPhi` array at each vertex.[2] After the triangle loop comes a call to Petsc

profiling utility macro `PLogFlops()`, instructing it to increment the count of floating point operations executed by 33 times the number of local elements. ↑`xDR()`

An analogous set of operations is now performed to compute the second term on the right-hand side of (2.9) and add it to the first. This term, ↑`dPhi()`

$$\int \frac{\partial d}{\partial z}\frac{\partial z}{\partial \varphi}\lambda_i d^2x \;=\; \left(\frac{z_i'}{12}+\frac{z_{i+1}'}{24}+\frac{z_{i+2}'}{24}\right)\sum_j d_j(R_{j+2}-R_{j+1})$$

will be computed by a call to `xDZ()`, located in **mesh/operator/dZ.c**. Once again the routine is just a wrapper. It passes the arrays to `xDZ0()`, which tidily computes its sums in the familiar manner, logs the number of floating point operations, and returns. ↓`xDZ()` ↕`xDZ0()`

Now that `dPhi()` has computed the chain-rule derivatives on the right-hand-side, it must actually find $\frac{\partial d}{\partial \varphi}$ to use on the left. It hands this task off to the function `dfdphi()`, located in **mesh/operator/dPhi.c**. ↑`dPhi()` ↓`dfdphi()`

Unlike the chain-rule routines, which worked directly on `Scalar` data, `dfdphi()` will perform its finite differencing on Petsc vectors. It contains several distinct blocks of code that perform finite differencing to different orders; only one of these will be compiled. The flag set at the beginning of the source file specifies that, in this case, fourth-order differencing will be performed. Accordingly, a series of basic Petsc vector operations are invoked with the net effect of computing the approximation

$$\frac{\partial d}{\partial \varphi} \approx \frac{d_{\varphi-2}-8d_{\varphi-1}+8d_{\varphi+1}-d_{\varphi+2}}{12\Delta\varphi}$$

at all local vertices and storing the result in a scratch vector. Since, in this run, only two poloidal planes physically reside on a given processor, this is one of those routines in which Petsc's implicit handling of ghost values really pays off. ↑`dPhi()`

The only remaining task for `dPhi()` is to copy the new scratch vector into the "algebraic" right-hand-side vector. This is done with a call to Petsc routine `VecAXPY()`, which also flips the signs of the values for good measure.

Back in `dXdphi()`, a call is placed to `SLESSolve()`, Petsc's linear equation solver, which will effectively invert the mass matrix to solve equation (2.9) for $d\mathtt{d}/d\varphi - \partial\mathtt{d}/\partial\varphi$. `VecAXPY()` is used to add the just-computed $\partial\mathtt{d}/\partial\varphi$ to this Petsc vector, leaving it with the correct values. A final call is then made to `GetArrayInRPData()` to make this new vector available in array form, and the function returns. ↑`dXdphi()` ↑`initMHD()`

In preparation for the next round of initializations, `initMHD()` now creates useful pre-multipliers by inverting duplicate copies of the `R` and `d` vectors for each poloidal plane using Petsc routines `VecCopy()` and `VecReciprocal()`. Four calls to `GetArrayInRPData()` follow, making these new inverses, as well as the resistivity and viscosity vectors, available as arrays.

[2] The reason for the minus sign is most likely a difference in cylindrical coordinate ordering conventions between Vmec and M3D.

The next step is to compute $\nabla\tilde{I}$. First we compute the $\hat{R}$ component, $\frac{\partial\tilde{I}}{\partial R}$, with a call to `dXdR_PData()`, located in **mhd/operator/dXdR.c**. In particular, we are looking for the coefficients in the linear basis function expansion of this partial derivative, so we must solve ↓`dXdR_PData()`

$$\sum_j M_{ij}\left(\frac{\partial\tilde{I}}{\partial R}\right)_j = \int \frac{\partial\tilde{I}}{\partial R}\lambda_i d^2x = \sum_{\substack{j \text{ in} \\ \text{triangle } i}} \frac{\tilde{I}_j(z_{j+1}-z_{j+2})}{6}.$$

This must be done once for each poloidal plane. The first step is to compute the right-hand side, which is done by passing $\tilde{I}$, z, and the right-hand-side scratch space [zeroed with a call to `VecSet()`] to routine `dR()`, located in **mesh/operator/dR.c**.

Like `xDR()`, `dR()` is a wrapper, simply repackaging its arguments and passing them on to `dR0()`, in the same source file. It is `dR0()` that actually loops over vertices within each local triangle, adding the quantity ↓`dR()` ↓`dR0()`

$$\frac{\tilde{I}_{i+1}z_{i+2}-\tilde{I}_{i+2}z_{i+1}}{6}$$

to the right-hand-side element for vertex i. Why add only two of the six terms implied above? Because for any vertex that is not on the boundary, all the other terms will cancel when the sum is conducted over all triangles of which the vertex is a part. Not adding them in the first place saves effort.

Of course, for vertices that *are* on the boundary, the sums will come out wrong. This must be corrected once the element loop is finished. Accordingly, for the last boundary vertex, the first boundary vertex, and then for all boundary vertices in between, `dR0()` calls `FixBdyDR()`, located in the same source file. This function adds the four missing terms ↕`FixBdyDR()`

$$\frac{z_i(\tilde{I}_{i+2}-\tilde{I}_{i+1})+\tilde{I}_i(z_{i+1}-z_{i+2})}{6}$$

to vertex i. Some cancellations are likely to occur with this correction method, but this seems unavoidable. `dR0()` logs 18 flops for each local element with `PLogFlops()` and returns. ↑`dR()`

`dR()` immediately returns as well, transferring control to `dXdR_PData()`, which passes the new right-hand side to `SLESSolve()` along with the mass matrix inverter and the storage location for the solution vector of $\frac{\partial\tilde{I}}{\partial R}$ coefficients, which is promptly computed by the Petsc routine. Following a call to `GetArrayInPData()` to make this new solution vector array-accessible, this routine returns as well. The result is placed in `userData.extend.dtIdR`. ↑`dXdR_PData()` ↑`initMHD()`

Now the $\hat{z}$-component of the gradient, $\partial\tilde{I}/\partial z$, is computed by calling `dXdZ_PData()` in **mhd/operator/dXdZ.c**. The operation of this function is so closely analogous to that of `dXdR_PData()` above, that little further elaboration seems necessary. Replace the "R"s with "z"s and the "z"s with "R"s, and you have a complete description. The result is stored in `userData.extend.dtIdZ`. ↕`dXdZ_PData()`

The final component of the gradient is $1/R\,\partial\tilde{I}/\partial\varphi$. Just $\partial\tilde{I}/\partial\varphi$ is computed, by calling `dXdphi()`, which has already been described. The result is stored in `userData.dphis.dtIdphi`. ↕`dXdphi()`

2.5.2 The F Equation

We are now ready to solve the first elliptic equation,

$$\nabla_\perp^2 F = -\frac{1}{R}\frac{\partial\tilde{I}}{\partial\varphi}. \tag{2.11}$$

For each local poloidal section, a call is placed to `feqn()` in **mhd/feqn/feqn.c**. The function begins by calling Petsc routine `PLogEventBegin()`. This is a profiling call that will enable Petsc to track the performance of this particular routine. ↓`feqn()`

`feqn()` will solve (2.11) by treating it in the usual way: both sides are multiplied by λ_i and integrated over the plane. The right-hand side becomes

$$-\int \frac{1}{R}\frac{\partial\tilde{I}}{\partial\varphi}\lambda_i d^2x \;=\; \sum_j M_{ij}\left(-\frac{1}{R}\frac{\partial\tilde{I}}{\partial\varphi}\right)_j,$$

where the subscript j denotes the value of the expression at the jth vertex, while the left-hand side is

$$\int \nabla_\perp^2 F\lambda_i d^2x \;=\; \sum_j F_j \int \lambda_i \nabla_\perp^2 \lambda_j d^2x,$$

or, applying the chain rule,

$$\begin{aligned}\text{LHS} &= \sum_j F_j\left[\int \nabla_\perp\cdot(\lambda_i\nabla_\perp\lambda_j)d^2x - \int \nabla_\perp\lambda_i\cdot\nabla_\perp\lambda_j d^2x\right]\\ &= \oint \lambda_i\hat{n}\cdot\nabla_\perp F\,d\ell \;+\; \sum_j S_{ij}F_j \end{aligned}\tag{2.12}$$

where the line integral is taken around the boundary of the poloidal section and S is the stiffness matrix from page 25. The vector $\hat{n}$ is the unit normal to the boundary,

$$\hat{n} \;=\; \frac{\frac{\partial z}{\partial\theta}\hat{R} - \frac{\partial R}{\partial\theta}\hat{z}}{\sqrt{\left(\frac{\partial R}{\partial\theta}\right)^2 + \left(\frac{\partial z}{\partial\theta}\right)^2}}$$

for the axisymmetric case, where θ is the poloidal angle in Vmec coordinates, so the integral called for is actually

$$\oint \lambda_i\frac{\partial F}{\partial n}d\ell,$$

i.e., the λ-weighted line integral over the normal derivative of F. The normal derivative will be determined by the boundary conditions. The relevant conditions are that the boundary is both a flux surface (a surface of constant poloidal flux ψ) and a perfect conductor. At the surface of a perfect conductor, the normal component of the magnetic field vanishes. Hence, using (1.10), the boundary condition for an axisymmetric boundary is

$$\hat{n} \cdot \mathbf{B} \;=\; \frac{1}{R}(\hat{t} \cdot \nabla\psi + \hat{n} \cdot \nabla_\perp F) \;=\; \frac{1}{R}\frac{\partial F}{\partial n} \;=\; 0$$

and the normal derivative of F vanishes at the boundary as well, zeroing out the line integral and leaving only the vector product with the stiffness matrix on the left-hand side.

We now have a simple linear system of equations for F, but we also have a problem: the specification of the derivative of F everywhere on the boundary overdetermines the solution. The right-hand side of (2.11) is apparently arbitrary, yet if we integrate the left-hand side over the plane, we find that

$$\int \nabla_\perp^2 F \, d^2x \;=\; \int \nabla_\perp \cdot \nabla_\perp F \, d^2x \;=\; \oint \hat{n} \cdot \nabla_\perp F \, d\ell \;=\; 0.$$

In order for a meaningful solution to exist, it must therefore be the case that *the surface integral of the right-hand side vanishes as well.* Ideally, this would always be the case; however, round-off errors during numerical integration will inevitably result in the accumulation of a small residual non-zero value of this integral, eventually corrupting the solution if steps are not taken to prevent it. The sensible step is to subtract the residual value from the right-hand side, guaranteeing that the integral will vanish. To find F_j, the code must therefore solve the linear equation

$$S_{ij} F_j = M_{ij} \left[\left(-\frac{1}{R}\frac{\partial \tilde{I}}{\partial \varphi} \right)_j - \left\langle -\frac{1}{R}\frac{\partial \tilde{I}}{\partial \varphi} \right\rangle \right] \tag{2.13}$$

where

$$\langle u \rangle \;\equiv\; \frac{\int u \, d^2x}{\int d^2x}$$

is the average value over the poloidal plane.

The first calculation performed is of the area of the (local) plane. This area is, of course,

$$\int d^2x,$$

which can be found by submitting a constant function that takes the value unity at all vertices to a general-purpose surface integration routine. To create such a function, the `tmpGhostedVector` variable within the `scratch` element of the `userData.canvas` data structure is set to unity with `VecSet()` and shared among ghost vertices on neighboring processors with `VecGhostUpdateBegin()`

and `VecGhostUpdateEnd()`. Local access to the vector is provided by a call to `VecGhostGetLocalForm()`. Array access to the local vector is then obtained by calling `VecGetArray()`.

A pointer to the array thus obtained is then passed to routine `SurfaceIntegral0()`, along with the area array for the local finite element matrix and the index mapping, with the result to be stored in local `double` variable `areaNode`. Located in file **mesh/operator/surfaceIntegral.c**, this function is the general-purpose routine described above. Given the expansion of a function $f(\mathbf{x})$ in terms of piecewise linear basis functions, it approximates the surface integral ↓`SurfaceIntegral0()`

$$\int_{\text{local section}} f(\mathbf{x}) d^2x \approx \sum_j f_j \int \lambda_j(\mathbf{x}) d^2x = \sum_j f_j \frac{\Delta_j}{3},$$

where Δ_j is the area of the triangle containing vertex j. It does this by simply summing over local vertices within all local triangles. Given the function $f_j \equiv 1$, it simply sums up the areas. It then logs nine flops per local element and returns.

The unity array has served its purpose and may now be freed by calling ↑`feqn()` `VecRestoreArray()`. This then makes it possible to free the local vector with a call to `VecGhostRestoreLocalForm()`. The computation of the area of the poloidal plane is completed by summing the areas of the local sections with MPI collective communication routine `MPI_Allreduce()`, which will sum the values of `areaNode` over all processors within each poloidal communicator, and store the result in `double` variable `area` on each processor.

Next, `feqn()` calls Petsc routine `VecNorm()` with option `NORM_2`, to compute the 2-norm of `dtIdphi`,

$$\left\| \frac{\partial \tilde{I}}{\partial \varphi} \right\|_2 \equiv \left[\sum_i \left(\frac{\partial \tilde{I}}{\partial \varphi} \right)_i^2 \right]^{1/2}.$$

This action is questionable since the value returned will never actually be used by the remainder of the subroutine. It was probably inserted for error-checking purposes and ought to be removed or commented out for efficiency.

It is now necessary to prepare for inversion by computing the right-hand side of Equation (2.13). First, Petsc routine `VecPointwiseMult()` is called, multiplying each element of `dphis.dtIdphi` by the corresponding element of `coeff.Rinverse`, and placing the result $= 1/R\ \partial\tilde{I}/\partial\varphi$ in the appropriate position of `canvas.rhs_algebraic`. The signs of all elements are then flipped by passing the factor -1 to Petsc function `VecScale()`.

Once the individual components on the right-hand-side have been computed, it is possible to compute the residual

$$-\int \frac{1}{R} \frac{\partial \tilde{I}}{\partial \varphi} d^2x.$$

First, a copy of the vector is made with a call to `VecCopy()`. Within the copy, ghost values are exchanged by making calls to `VecGhostUpdateBegin()` and

`VecGhostUpdateEnd()`. The local form, and then array access are obtained with calls to `VecGhostGetLocalForm()` and `VecGetArray()` respectively. The array is passed to `SurfaceIntegral0()`, which computes the integral over the local processor. The array is freed with `VecRestoreArray()`; the vector is freed with `VecGhostRestoreLocalForm()`, and finally the sum of the integrals is computed with `MPI_Allreduce()`. At this early stage of the computation, this sum is, comfortingly, exactly zero to within machine precision. ↕`SurfaceIntegral0()`

Before the right-hand side can be multiplied by the mass matrix, one more correction needs to be made to the residual. This one arises from the possibility that the outer boundaries may vary in size and shape with toroidal angle φ. Because the boundary condition applies to the derivative of F normal to the boundaries, any departure of $\hat{n}$ from the poloidal plane must be accounted for in its application. Thus, re-examining (1.10) more carefully, we find that in fact

$$\hat{n} \cdot \nabla_\perp F \;=\; -\frac{I}{\epsilon}\hat{n} \cdot \hat{\varphi}.$$

This in turn means that the surface integral of the right-hand side should *not* necessarily be precisely zero in the general case but rather

$$\int \mathrm{RHS}\, d^2x \;=\; \oint \hat{n} \cdot \nabla_\perp F \, d\ell \;=\; -\oint \frac{I}{\epsilon}\hat{n} \cdot \hat{\varphi}\, d\ell$$

To ensure this condition, we must explicitly compute this last line integral, divide it by the area, and add it, along with the residual already computed, to each component of the right-hand-side vector.

The routine performs the line integral by calling `LineIntegralF()`, located in the **feqn.c** source file. This function performs its integration in a fairly straightforward manner that is amenable to parallelization: it computes the values of its integrand at triplets of successive boundary vertices and adds weighted sums of them to a running total, which is then summed over processors. For each boundary vertex i, the routine evaluates $I = 1 + \epsilon\tilde{I}$ at vertices $j = i-1$, i, and $i+1$. It then divides each of these values by the inverse aspect ratio ϵ and multiplies by ↓`LineIntegralF()`

$$\hat{n} \cdot \hat{\varphi} \;=\; \frac{\frac{\partial R}{\partial \theta}\frac{\partial z}{\partial \varphi} - \frac{\partial R}{\partial \varphi}\frac{\partial z}{\partial \theta}}{R\sqrt{\left(\frac{\partial R}{\partial \theta}\right)^2 + \left(\frac{\partial z}{\partial \theta}\right)^2}}$$

evaluated at vertex j, which is actually the variable `NormPhi` computed earlier. Having computed these values of the integrand, it now forms the sum

$$\frac{(\mathrm{integrand}_{i-1} + 2 \times \mathrm{integrand}_i)d\ell_i}{6} + \frac{(\mathrm{integrand}_{i+1} + 2 \times \mathrm{integrand}_i)d\ell_{i+1}}{6}$$

and subtracts it from the running total. Once this operation has been completed for all boundary vertices in the local processor, the totals are summed up with a call to `MPI_Allreduce()`. The sum is then returned by the function.

The surface integral computed in the previous integration is now combined ↑`feqn()`

with the line integral to form

$$\text{residual} = \frac{\text{line integral} - \text{surface integral}}{\text{area}}$$

and added to the `rhs_algebraic` vector computed initially. This vector is then multiplied by the mass matrix using Petsc routine `MatMult()`, with the result stored in the right-hand-side vector `rhs`. Direct access to this vector is obtained by calling `VecGetArray()`, following which it is passed to `ApplyBdyonF()` in the **feqn.c** source file.

`ApplyBdyonF()` exists to apply the boundary condition on F, which takes the form of the line integral in (2.12). We saw that in the axisymmetric case, this integral vanished. In the general case, it will be equal to ↓`ApplyBdyonF()`

$$-\frac{1}{\epsilon}\oint \lambda_i I \hat{n} \cdot \hat{\varphi}\, d\ell.$$

The basis function λ_i picks out the piece of this integral centered on vertex i. `ApplyBdyonF()` cycles through the same triplets of boundary points as `LineIntegralF()`, and computes the same integrand values at each vertex, but instead of subtracting the weighted sum of these values from a running total, it adds it to the ith component of the right-hand-side vector.

The stiffness matrix `gfe.Stiffness` for the current φ section is passed to `matZeroInterior()`. This function is located in **mhd/operator/matZeroInterior.c**. ↑`feqn()`

Making use of Petsc's ability to automatically decide on and carry out the necessary interprocess communication for updating distributed data structures, it alters the matrix by first setting the values on a single processor within each poloidal communicator; then having all processors update as required. The nature of the alteration will be to store a copy of the top row of the stiffness matrix and then replace that row with the top row of the identity matrix. The active processor begins by extracting a copy of the zeroth row of the sparse matrix for reading using Petsc routine `MatGetRow()` It copies the column indices and values of the nonzero elements of the row to the "reference row" structures in the MHD solver `userSolver` for temporary storage and then frees the row again with `MatRestoreRow()`. Off-diagonal elements of the row are then set to zero using `MatSetValue()`, while the diagonal element is set to one. Finally, all processors assimilate this change by calling `MatAssemblyBegin()` followed immediately by `MatAssemblyEnd()`, and the function returns. ↓`matZeroInterior()`

The next order of business is to set up the system of linear equations for solution by Petsc's parallel linear system solver. This is accomplished with a call to Petsc routine `SLESSetOperators()`. The arguments passed to this routine initialize the solver context `userSolver.FInverter` using the slightly modified stiffness matrix to define the system. The stiffness matrix is also used to construct the preconditioner, and Petsc is further informed, with the `SAME_PRECONDITIONER` flag that all linear solves within the `FInverter` context from now on will use the same preconditioner (so that it only needs to be computed the first time). ↑`feqn()`

The code is now ready to solve for F. It calls `SLESSolve()`, passing it the `FInverter` context and the right-hand-side vector. F is found and stored in the Petsc vector `userData.extend.F.g` for the poloidal plane. Next, `matRestoreInterior()` is called. Located in **matZeroInterior.c**, it simply restores the first row of the stiffness matrix to the values it had before the call to `matZeroInterior`. Profiling calls `PLogFlops()` and `PLogEventEnd()` are made, and `feqn()` returns. ↕`matRestoreInterior()`

`initMHD()` now takes the gradient of the newly computed F field, first calling `dXdZ()` in **mhd/operator/dXdZ.c** to compute $\partial F/\partial z$. This routine is identical in function to `dXdZ_PData()`, described earlier, except that it produces arrays of type `RParData` rather than `ParData`. The result is stored in `userData.dRdZs.dFdZ`. Next, $\partial F/\partial R$ is computed with a call to `dXdR()` in **mhd/operator/dXdR.c**, with the result stored in `userData.dRdZs.dFdR`. Finally, the φ partial derivative is computed by `dXdphi()`. ↑`initMHD()`

2.5.3 The ψ Equation

Because, as ascertained earlier, the MHD data in the `userData` structure has *not* been obtained from a checkpoint file, $\Delta^\star\psi$ must now be explicitly computed from the known value of C (see page 32) using

$$C = \Delta^\star\psi + \frac{1}{R}\frac{\partial F}{\partial z}.$$

For each plane, the vector $\Delta^\star\psi$ is set to $(1/R)\partial F/\partial z$ using `VecPointwiseDivide()`; then `VecAYPX()` is called to flip the sign of the vector and add C to it. The now correct value of $\Delta^\star\psi$, stored in `userData.old`, is copied to `userData.base` with `VecCopy()`, and array access to both vectors is provided by two calls to `GetArrayInPData()`.

Now that we know $\Delta^\star\psi$, we can invert it to find ψ. `psieqn()`, found in **mhd/psieqn/psieqn.c**, is called once for each poloidal plane for this purpose. `psieqn()` is simpler than `feqn()` because the boundary condition for ψ is simpler: because the boundary is a flux surface, ψ is constant along it. Since ψ is only defined to within an arbitrary constant, its boundary value is chosen to be zero. The left-hand side of the elliptic equation to be solved is

$$\int \lambda_i \Delta^\star\psi \, d^2x \;=\; (\texttt{starMatrix})_{ij}\, \psi_j$$

and the right-hand side is

$$\int \lambda_i \Delta^\star\psi \, d^2x \;=\; M_{ij}\, (\Delta^\star\psi)_j \,.$$

The function begins with a profiling call, `PLogEventBegin()`. Following this, `userData.canvas.rhs_algebraic` is set to zero with a call to `VecSet()`. The $\Delta^\star\psi$ vector computed by the calling function is multiplied from the left by the mass matrix with a call to `MatMult()`, with the result stored in `canvas.rhs`, the right-hand-side vector. ↓`psieqn()`

The star matrix for the current φ section is now passed to `matZeroBdy()`, in **mhd/operator/matZeroBdy.c**, which will help to apply the condition $\psi \equiv 0$ at the boundary. It does so by looping through all local boundary vertices and extracting from the star matrix the row corresponding to each one using `MatGetRow()`. The values from that row are carefully stored in `userSolver`'s `bdyColumnIndex` and `bdyMatEntree` backup areas, and the row is then freed again with `MatRestoreRow()`. Once all boundary rows have been safely backed up, a second loop reassigns all their values to those of the corresponding rows in the identity matrix using `MatSetValue()`. After all loops are completed, calls to `MatAssemblyBegin()` and `MatAssemblyEnd()` reassemble the matrix with the new values. The practical effect of all this is that, instead of participating in the elliptic solution, the boundary vertices will simply be assigned to the values in the corresponding rows of the right-hand-side vector. These rows must, of course, be set to zero to enforce the desired boundary condition. ↓`matZeroBdy()`

The modified star matrix is now used in a call to `SLESSetOperators()` to define the system of equations and preconditioner matrix for the `psiInverter` solver context. As with `FInverter` earlier, the `SAME_PRECONDITIONER` option is used to prevent unnecessary duplication of effort in preconditioning. The $\psi = 0$ boundary condition is now formalized in a loop over boundary vertices that zeroes all of their corresponding elements in the right-hand-side vector. Calls to `VecAssemblyBegin()` and `VecAssemblyEnd()` distribute the zeroes as needed. ↑`psieqn()`

A single call to `SLESSolve()` now does all the real work. ψ is found and stored in `userData.extend.Psi`. `matRestoreBdy()`, in the **matZeroBdy.c** source file, is called to restore the boundary rows of the star matrix to their original values. Profiling information is compiled with calls to `PLogFlops()` and `PLogEventEnd()`, and the function returns. ↕`matRestoreBdy()`

For each poloidal plane on which the ψ equation is solved, the ψ vector is copied to `userData.extend.deltaPsi`. After the loop over poloidal planes, direct access to the ψ array is provided by calling `GetArrayInPData()`. The ψ vector is then passed in turn to `dXdZ()`, `dXdR()`, and `dXdphi()` to compute $\partial\psi/\partial z$, $\partial\psi/\partial R$, and $\partial\psi/\partial\varphi$ respectively. ↑`initMHD()`

2.5.4 Initial Velocity Perturbation

A call to `OptionsGetDouble()` now scans the command line for the presence of the "`-deltaU`" option, which would prescribe an initial perturbation ΔU to the equilibrium velocity distribution, which is assumed to be identically zero. The option is found and the relative amplitude it specifies, 10^{-4}, is read into the `double` precision scalar `deltaU`. This value is then passed, along with the usual mesh, data, and solver information, to subroutine `addPerturbation()`, which can be found in file **mhd/init/addPerturbation.c**.

The first thing this routine does is to read some additional command line options: ↓`addPerturbation()`

Option	Description	Value
`-pModePerturbation`	poloidal mode number of perturbation	1
`-tModePerturbation`	toroidal mode number of perturbation	not found
`-sGridPerturbation`	s location of peak perturbation	0.30
`-sWidthPerturbation`	s extent of perturbation	0.20

The default value of the toroidal mode number is one, so the routine will implement an $n = 1$, $m = 1$ perturbation.

After reading the command line options, the routine allocates space for two integer arrays, `in` and `out`, to hold vertex indices. Using `malloc()`, it also allocates a `Scalar` array of the same size to hold the perturbation itself. The values of `in` are set to their own indices, the indices of the vertices in the local ordering. This array is then passed through `ISLocaltoGlobalMappingApply()`, with the result — the indices of the same vertices in the global application ordering — stored in `out`. This is then copied back into `in`, which is then switched in-place to the Petsc ordering with a call to `AOApplicationToPetsc()`. As a result, `in` is in the Petsc ordering and `out` is in the application ordering.

Now the routine loops through local poloidal sections, keeping track of the value of φ for each one. The perturbation array for each section is computed in a loop over local vertices. The Vmec coordinates (r, θ) for each vertex are found by passing its application-ordering index to `mapIndexToVmecCoordinates()`. The size of the perturbation is then determined by the formula

$$\text{perturbation}_i \;=\; s_i^{m/2} \cos(m\theta_i + n\varphi) \exp\left[-\left(\frac{s_i - s_0}{w_0}\right)^2\right] \tag{2.14}$$

where $m = 1$ is the poloidal mode number, $n = 1$ is the toroidal mode number, $s_0 = 0.3$ is the minor radial location where the perturbation has its peak value, and $w_0 = 0.2$ is the characteristic width of the peak in s. Following the vertex loop, the array of perturbation values for the poloidal section is copied into the Petsc vector `userData.extend.U.g` with calls to `VecSetValues()` (using the Petsc-ordering indices in `in`), `VecAssemblyBegin()`, and `VecAssemblyEnd()`.

If the normalized pressure exceeds 10^{-5} anywhere on the mesh, the pressure distribution will be used to modulate the amplitude of the perturbation; otherwise, it will not. Petsc routine `VecMax()` is called to determine the maximum value attained by the normalized pressure over the current section, which turns out to be approximately 2.84×10^{-2}. Accordingly, the values in the perturbation vector `U` are rescaled to

$$U_i \;\rightarrow\; \frac{p_i}{\text{peak pressure}} U_i$$

before being multiplied by the specified perturbation amplitude of 10^{-4}. Array access to the rescaled vector is provided with a call to `GetArrayInPData()`.

If no perturbation had been specified, the 0 values currently stored in the $\Delta^\dagger U$ and various ∇U vectors would have been correct. Since there *is* a perturbation, however, these derivatives must be computed. The components of the

gradient are computed first, with calls to `dXdR()`, `dXdZ()`, and `dXdphi()`. The Galerkin method is used to take the Laplacian: we solve

$$M_{ij}\left(\Delta^\dagger U\right)_j \;=\; \texttt{daggerMatrix}_{ij}U_j$$

for $\Delta^\dagger U$ by multiplying the newly computed U vector from the left by the `daggerMatrix` to form the right-hand-side; then passing this along with the `massInverter` solver context to `SLESSolve()`. The result is stored in data structure `userData.old.DelDaggerU`. The function then returns, unfortunately neglecting to free the three allocated arrays before doing so. ↑`initMHD()`

2.5.5 The Φ Equation

The elliptic equation (1.22) defining the variable Φ is by far the most complex of all the elliptic equations in the model. It depends on many of the other variables describing both velocity and electromagnetic fields, so access to these must be provided before it can be solved. Seven calls to `GetArrayInPData()` are therefore made, providing direct array access to the variables v_φ, $\tilde{I}$, $\Delta^\star\psi$, $\Delta^\dagger U$, $\Delta^\star\chi$, p, and d now residing in `userData.old`. With this out of the way, the solution of the equation can begin.

For each φ section, the subroutine `phieqn()` is called. It can be found in **mhd/phieqn/phieqn.c**. As usual, it begins with a Petsc profiling call, to ↓`phieqn()` `PLogEventBegin()` to keep track of the time taken to solve the Φ equation. The right-hand-side vector will be pieced together one term at a time, so it and `rhs_algebraic` are first zeroed with two calls to `VecSet()`.

Terms involving the fluid velocity [the first two lines of (1.22)] are handled first. They will be needed only if the user has *not* specified the `-noFlow` option on the command line. The code checks for this condition by inspecting the parameter flag; since its `FLOW_IS_NONE_VANISHING` bit is found to be set, the terms are needed.

The operation represented by the first term, $\epsilon\nabla_\perp\tilde{I}\cdot\nabla_\perp U$, is common enough to deserve a special notation. It will be denoted the "inner bracket" of $\tilde{I}$ and U. The three quantities ϵ, $\tilde{I}$, and U are accordingly passed to the function `innerBracket()` along with a pointer to the location `canvas.rhs` to store the result. This function is located in **mesh/operator/innerBracket.c**.

`innerBracket()` is another wrapper function. Having been passed `ParData` ↓`innerBracket()` and `TmpData` structures (which contain Petsc vector and scalar array components), it extracts the arrays and passes them along with the ϵ factor and the `drdr` finite element matrix to `innerBracket0()` in the same source file to perform the actual calculation. This second function encloses its operations in a ↓`innerBracket0()` loop over triangular elements. To see why it does this, remember that we are computing

$$\epsilon\int\left(\frac{\partial\tilde{I}}{\partial R}\frac{\partial U}{\partial R}+\frac{\partial\tilde{I}}{\partial z}\frac{\partial U}{\partial z}\right)\lambda_i d^2x \;=\; \epsilon\sum_{m,n}\tilde{I}_m U_n\int\left(\frac{\partial\lambda_m}{\partial R}\frac{\partial\lambda_n}{\partial R}+\frac{\partial\lambda_m}{\partial z}\frac{\partial\lambda_n}{\partial z}\right)\lambda_i\, d^2x.$$

The sum is over all basis functions m, n, but the integral will vanish except where λ_i, λ_m, and λ_n are all nonzero in the same element. There are nine combinations of m and n for which this occurs; they each contribute one of the six unique values of the symmetric `drdr` tensor for the given element to the sum.

Because of this structure, `innerBracket0()` first passes the current element index j, the `drdr` array, and $\tilde{I}$ and U but not ϵ to function `accumInnerBracket()` in the same source file. This function computes ↓`accumInnerBracket()`

$$\sum_{m\in\Delta_j} \tilde{I}_m \left(\sum_{n\in\Delta_j} \texttt{drdr[j]}_{mn} U_n \right)$$

and returns this nine-term sum to the calling function. For each vertex of ↑`innerBracket0()` element j that resides on the local processor, the product of the sum and ϵ is added to the corresponding row of the right-hand-side vector, and the element loop continues. (This successive addition is, of course, necessary since in general each vertex is part of several triangular elements.) After the element loop, the routine `PLogFlops()` is called to log 26 floating-point operations for each element (20 for the accumulation of the sums, and 6 to add them to the right-hand-side vector), and the function returns, as does the wrapper function. ↑`innerBracket()`

The next term to be added is ↑`phieqn()`

$$I\nabla_\perp^2 U.$$

A vector storing $I = 1+\epsilon\tilde{I}$ is first constructed by copying $\tilde{I}$ into `tmpVector` with `VecCopy()`, multiplying `tmpVector` by ϵ with `VecScale()`, and then adding one to it with `VecShift()`. $\nabla_\perp^2 U$ is not available at this point, but $\Delta^\dagger U$ (which is equal to it in the straight-cylinder limit) is. `VecPointwiseMult()` is called to multiply each element of I with the corresponding element of $\Delta^\dagger U$, storing the resulting vector in `tmpData`. Before correcting for toroidal geometry, the code adds this intermediate quantity to `rhs_algebraic` with a call to `VecAXPY()`.

The code now checks the parameter flag for the `CYLINDER` option, which would indicate that the present case is in straight-cylinder geometry and that the $I\nabla_\perp^2 U$ term is therefore complete. The flag is not found, because toroidal geometry is being used. A correction must therefore be made. `tmpData`, which now contains $I\Delta^\dagger U$, is overwritten with $I\,\partial U/\partial R$, which is computed with another call to `VecPointwiseMult()`. This function is called again to multiply the new vector in place by `Rinverse`; `VecScale()` would probably have been a better choice. Finally, this new quantity, $I/R\,\partial U/\partial R$, is subtracted from `rhs_algebraic`, leaving it equal to $I\nabla_\perp^2 U$, the correct term.

Having taken care of the terms in (1.22) involving U, we proceed to those involving χ. The first of these that will be added is

$$\nabla_\perp\chi \times \nabla_\perp \left(\frac{\tilde{I}}{R}\right) \cdot \hat{\varphi} \;=\; \frac{\partial\chi}{\partial R}\frac{\partial}{\partial z}\left(\frac{\tilde{I}}{R}\right) - \frac{\partial\chi}{\partial z}\frac{\partial}{\partial R}\left(\frac{\tilde{I}}{R}\right).$$

This frequently-arising combination will be referred to as the "Poisson bracket" of χ and $\tilde{I}/R$, and compactly denoted $[\chi, \tilde{I}/R]$. It will be computed by passing `Rinverse`, χ, and $\tilde{I}$ to function `xPoissonBracketRPP()`, located in file

mesh/operator/poissonBracket.c. This wrapper function strips out the array portions from these parallel data structures and hands them off to routine `xPoissonBracket0()` in the same source file. ↓`xPoissonBracketRPP()`

When computing the residual of a Poisson bracket, a tremendous simplification is generally possible. From ↓`xPoissonBracket0()`

$$\int [a,b]\lambda_i \, d^2x \;=$$

$$\frac{1}{12\Delta_i}\left[\left(\sum_{m\in\Delta_i} a_m dR_m\right)\left(\sum_{m\in\Delta_i} b_m dz_m\right) - \left(\sum_{m\in\Delta_i} a_m dz_m\right)\left(\sum_{m\in\Delta_i} b_m dR_m\right)\right],$$

the application of a series of identities condenses this 18-term result to the simple 6-term expression

$$\frac{1}{6}\sum_{m\in\Delta_i} a_m(b_{m+1} - b_{m+2})$$

independent of the area or geometry of the element. The right-hand-side vector for vertex i must be incremented by this quantity for each basis function that is non-vanishing at i. The expression becomes even simpler when we realize that the terms contributed by an edge shared by by two elements containing vertex i will cancel each other. This means that, except for boundary vertices, only the two terms corresponding to the edge opposite the vertex will survive, and we need only add

$$\frac{1}{6}(a_{i+1}b_{i+2} - a_{i+2}b_{i+1})$$

to vertex i for each triangular element it appears in. There is a routine in the source file, `PoissonBracket0()`, that performs this operation.

For the present case, however, in which one of the quantities in the bracket is divided by R, computing the residual properly requires a bit more work. In general, the ith residual of the Poisson bracket $[a, xb] = x[a,b] + b[a,x]$ is

$$\frac{1}{4\Delta_i^2}\left\{\left[\left(\sum_{m\in\Delta_i} a_m dR_m\right)\left(\sum_{m\in\Delta_i} b_m dz_m\right) - \left(\sum_{m\in\Delta_i} a_m dz_m\right)\left(\sum_{m\in\Delta_i} b_m dR_m\right)\right]\int_\Delta x\lambda_i \, d^2x\right.$$

$$\left. + \left[\left(\sum_{m\in\Delta_i} a_m dR_m\right)\left(\sum_{m\in\Delta_i} x_m dz_m\right) - \left(\sum_{m\in\Delta_i} a_m dz_m\right)\left(\sum_{m\in\Delta_i} x_m dR_m\right)\right]\int_\Delta b\lambda_i \, d^2x\right\}$$

For the special case $x \equiv 1$, of course, the first integral is just $\Delta_i/3$, while the quantity in the second set of square brackets vanishes, so the entire expression reduces to the one above. In general, though, the top integral is equal to

$$\sum_{j\in\Delta_i} M_{ij}x_j \;=\; \frac{\Delta_i}{6}\left(x_i + \frac{x_{i+1}+x_{i+2}}{2}\right)$$

and the full expression becomes

$$\frac{1}{12}\left[\left(x_i + \frac{x_{i+1}+x_{i+2}}{2}\right)\sum_{m\in\Delta_i} a_m\left(b_{m+1}-b_{m+2}\right)\right.$$

$$\left. + \left(b_i + \frac{b_{i+1}+b_{i+2}}{2}\right)\sum_{m\in\Delta_i} a_m\left(x_{m+1}-x_{m+2}\right)\right].$$

At this point, the code only concerns itself with the first of these sums. After making all possible cancellations, the contribution from the residue of $x[a,b]$ to the right-hand-side vector element for non-boundary vertex i from points local to element j is

$$\frac{1}{12}\left(x_i + \frac{x_{i+1}+x_{i+2}}{2}\right)\left(a_{i+1}b_{i+2}-a_{i+2}b_{i+1}\right) + \frac{x_{i+2}}{24}(a_i b_{i+1}-a_{i+1}b_i)$$

$$+ \frac{x_{i+1}}{24}(a_{i+2}b_i - a_i b_{i+2}). \tag{2.15}$$

This is what `xPoissonBracket0()` will evaluate.

This routine first loops over local triangular elements j. Within each element, it loops over local vertices i. For each vertex, it computes the quantity (2.15), multiplies it by a supplied factor (one in this case), and adds it to the appropriate slot in the right-hand-side vector. This leaves all but the boundary vertices set correctly.

Correcting the values at boundary vertices is the next task. For each such vertex, starting from the first and last and then proceeding in order through the rest, the routine passes the vertex index and the indices of its two neighbors along the boundary to subroutine `FixBdyXPoisson()`, immediately following it in the source file. The missing factors for boundary vertex i just consist of the difference between the non-boundary vertex expression (2.15) and the top line of the full expression above that. If $i+1$ is the index of the neighbor in the counter-clockwise direction and $i-1$ is the index of the clockwise neighbor, the quantity to be added to vertex i is ↓`FixBdyXPoisson()`

$$\frac{1}{12}\left[\left(a_{i+1}b_i - a_i b_{i+1}\right)\left(x_i + \frac{x_{i+1}}{2}\right) + \left(a_i b_{i-1} - a_{i-1}b_i\right)\left(x_i + \frac{x_{i-1}}{2}\right)\right].$$

After adding it, the function returns. ↑`xPoissonBracket0()`

This concludes the Poisson bracket operation. 69 flops per element are logged with a call to `PLogFlops()`, and control reverts to the wrapper, which immediately returns as well. ↑`xPoissonBracketRPP()`

One more term involving χ remains, namely ↑`phieqn()`

$$\frac{1}{\epsilon R^2}\frac{\partial\chi}{\partial z}.$$

Since this term occurs only in toroidal geometry, the code checks again to make sure the present case is in this geometry before adding it. The term will be

added to `rhs_algebraic` rather than `rhs`, so it does not yet need to be integrated over basis functions. The $1/\epsilon$ factor is computed, `Rinverse` is squared by `VecPointwiseMult()` and stored in `tmpData`, and this in turn is multiplied in place by the contents of `tmpVector` with another `VecPointwiseMult()` call. The result is multiplied by the $\partial\chi/\partial z$ vector and then by $1/\epsilon$ and added to `rhs_algebraic` using `VecAXPY()`.

The first two terms in the expression for the velocity have now been exhausted, and the code now turns its attention to v_φ. The first term in the $\nabla_\perp^2\Phi$ equation involving this variable that will be added is

$$-\frac{1}{R}\nabla_\perp v_\varphi \cdot \nabla\psi.$$

The minus sign, the $1/R$ factor, the v_φ array, and the ψ array are all passed to routine `xInnerBracketRPP()` in **mesh/operator/innerBracket.c** to compute the residual of this term.

`xInnerBracketRPP()` is quite similar to `innerBracket()`, discussed above. ↓`xInnerBracketRPP()`
For one thing, it is yet another wrapper routine. It repackages its four array arguments (three in, one out) and the `drdr` matrix and passes them along to `xInnerBracket0()` along with its scalar argument. Like the Poisson bracket routine discussed above, `xInnerBracket0()` will ignore derivatives of $x = 1/R$. ↓`xInnerBracket0()`
It will only compute

$$\int x(\nabla A \cdot \nabla B)\lambda_i \, d^2x.$$

Because the basis functions are linear, their first derivatives are constants over the elements on which they are defined. This means that both in the original inner bracket and the present one, they can be taken outside the integral. In the former case, that simply left $\int \lambda_i d^2x$, while it now leaves $\int x\lambda_i d^2x$. We can therefore use the same nine-term inner product involving A, B, and `drdr` as before, computed for each element by `accumInnerBracket()`. However, the ↕`accumInnerBracket()`
sums must now be corrected by the factor

$$\frac{\int x\lambda_i d^2x}{\int \lambda_i d^2x} = \frac{\sum_{j\in\Delta_i} M_{ij}x_j}{\Delta_i/3} = \frac{1}{2}\left(x_i + \frac{x_{i+1}+x_{i+2}}{2}\right).$$

`xInnerBracket0()` differs from `innerBracket0()` only in that it computes this factor and multiplies it by each sum along with the minus sign multiplier before adding it to the right-hand-side vector. When it completes its loop over elements, it logs 44 floating point operations for each local vertex with `PLogFlops()` and returns. The wrapper function has nothing left to do and ↑`xInnerBracketRPP()`
returns as well.

The other term involving v_φ, ↑`phieqn()`

$$-\frac{v_\varphi}{R}\Delta^\star\psi,$$

will be added to the algebraic right-hand side. First v_φ/R is computed using `VecPointwiseMult()` and stored in `tmpData`. `VecPointwiseMult()` is called

again to multiply this quantity in place by $\Delta^\star\psi$. `VecAXPY()` is then called to subtract the result from the algebraic right-hand side.

This would ordinarily be the last of the terms involving the velocity. In certain cases, however, there are more, notably the

$$-\nabla_\perp F \times \nabla_\perp \left(\frac{v_\varphi}{R}\right) \cdot \hat{\varphi}$$

term. The code checks to see whether the `STELLERATOR` or `NONLINEAR` bits are set in the parameter flag to determine whether this is one of those cases. It is not; the flags are not set. The statement block adding the extra terms is therefore skipped, and the code reaches the closing bracket of the `FLOW_IS_NONE_VANISHING` condition.

The next collection of "optional" terms are those involving the resistivity, η, i.e., the last two lines of equation (1.22). They will be needed only if $\eta > 0$. It is, and the `RESISTIVE_MHD` bit of the parameter flag has therefore been set. The code checks for this, and, finding it, proceeds to add the resistive terms. It begins with line three of (1.22) — the constant-η terms.

To compute the third line, the code must evaluate the quantity in square brackets,

$$R^2\nabla_\perp \cdot \mathbf{J} \;=\; \frac{1}{R}\left(\frac{\partial F'}{\partial z} - \frac{\partial \psi'}{\partial R}\right) - \frac{\partial \tilde{I}}{\partial z} + \frac{\partial C}{\partial \varphi}.$$

To be more precise, it will evaluate $-R^2\nabla_\perp \cdot \mathbf{J}$ and then subtract it from the right-hand side. It begins with the expression in parentheses, in particular, with $\partial\psi'/\partial R$. $\psi' = \partial\psi/\partial\varphi$ is already known, so the code simply takes its first derivative with respect to R by calling `dXdR0()` with it as an argument.

Located in **mhd/operator/dXdR.c**, this routine is similar to `dXdR()`, discussed above, but performs its operations only within a single poloidal plane specified by the calling routine. After extracting the values in this plane from the data structures for the variable "x" to be differentiated and for the spatial coordinate z, `dXdR0()` zeroes its temporary right-hand-side workspace and then initializes it with a call to `dR0()` (described on page 40). It solves for $\partial x/\partial R$ using `SLESSolve()` and then returns. ↓`dXdR0()` ↕`dR0()`

$\partial F'/\partial z$ is computed next. Again, F' is already available, so the code simply calls `dXdZ0()` to take the z-derivative. Located in **mhd/operator/dXdZ.c**, this routine bears the same relation to `dXdZ()` that `dXdR0()` has to `dXdR()`. It takes the derivative and returns. $\partial F'/\partial z$ is subtracted from $\partial\psi'/\partial R$ with `VecAXPY()` and the result is multiplied by $1/R$ using `VecPointwiseMult()`. $\partial\tilde{I}/\partial z$, which has already been computed, is added to this result with another call to `VecAXPY()`. ↑`phieqn()` ↕`dXdZ0()`

The one remaining term (and the only one that would have survived in the case of straight cylinder geometry) is now computed by passing C to `dXdphi0()` in **mhd/operator/dXdphi.c**. Like `dXdphi()`, it zeroes two vectors, `rhs` and `rhs_algebraic`, sets them up by calling `dPhi()`, and solves the equation for `rhs` with `SLESSolve()`, adding `rhs_algebraic` to the result. The only apparent difference between the routines is that this one only treats a single poloidal ↕`dXdphi0()`

plane. The result, $\partial C/\partial\varphi$, is subtracted from the sum under construction with `VecAXPY()`.

The new sum is now multiplied in place twice by $1/R$ using `VecPointwiseMult()` to form $-\nabla_\perp \cdot \mathbf{J}$, the quantity multiplied by $-\eta$ in the elliptic equation. If η is constant, the fourth line of (1.22) will vanish, and the resistive term will already be complete. The code checks for this condition by looking for the `VARY_RESISTIVITY` bit in the parameter flag. The bit is not set, so the resistivity is considered constant, and the code multiplies $-\nabla_\perp \cdot \mathbf{J}$ by η and subtracts it from `rhs_algebraic`.

Having finally added in all the terms in the equation for Φ, the code is ready to solve it. Specifically, it will use the Galerkin residual method to solve

$$\sum_j M_{ij} \left(\nabla_\perp^2 \Phi - \texttt{rhs_algebraic}\right)_j \;=\; \texttt{rhs}_i \tag{2.16}$$

where the just computed right-hand-side vectors are

$$\texttt{rhs_algebraic} \;=\; I\nabla_\perp^2 U + \frac{1}{\epsilon R^2}\frac{\partial \chi}{\partial z} - \frac{v_\varphi}{R}\Delta^\star\psi + \eta\nabla_\perp \cdot \mathbf{J}$$

and

$$\texttt{rhs}_i \;=\; \int d^2x\lambda_i \left[\epsilon\nabla_\perp \tilde{I} \cdot \nabla_\perp U + \nabla_\perp\chi \times \nabla_\perp \left(\frac{\tilde{I}}{R}\right) \cdot \hat{\varphi} - \frac{1}{R}\nabla_\perp v_\varphi \cdot \nabla\psi\right].$$

The routine solves (2.16) for the quantity in parentheses by calling `SLESSolve()`, storing the result in `userData.extend.LapPhi`. To find $\nabla_\perp^2\Phi$, it then adds `rhs_algebraic` to this vector with `VecAXPY()`.

It still must solve for Φ itself. To do this, it first multiplies `LapPhi` by the mass matrix, storing the result in the `rhs` vector. It sets the boundary condition on Φ, $\Phi_b \equiv 0$ by passing the stiffness matrix (which is the finite element expression of the $\nabla_\perp^2$ operator) to `matZeroBdy()`. As discussed above, this subroutine modifies its matrix argument by temporarily replacing the rows corresponding to boundary vertices with rows from the identity matrix. The modified stiffness matrix is then stored in the `PhiInverter` solver context with a call to `SLESSetOperators()`. The `SAME_PRECONDITIONER` flag is passed to this routine so that the preconditioner will only be computed once. The actual boundary values are then set to zero by calling `VecSetValue()` in a loop over boundary vertices, followed by `VecAssemblyBegin()` and `VecAssemblyEnd()`. Solution is accomplished by passing the inverter and right-hand side to `SLESSolve()`, with the result stored in `userData.extend.Phi`. The stiffness matrix is restored to its original condition by calling `matRestoreBdy()`, profiling calls are made to `PLogFlops()` and `PLogEventEnd()`, and the function returns. Calls to `GetArrayInPData()` then provide array access to the two new variables $\nabla_\perp^2\Phi$ and Φ.

↕`matZeroBdy()`

↕`matRestoreBdy()`

↑`initMHD()`

2.5.6 Deviations from Equilibrium

All the heavy pre-computations are now out of the way, but some bookkeeping remains to be done. The `userData.new` instances of $\tilde{I}$, $\Delta^\star\chi$, and p are zeroed in each plane with calls to `VecSet()`, following which `GetArrayInPData()` is invoked as usual. `MHD_RHS *` pointer `userData.rhsOld` is pointed at the zeroth element of `userData.Rhs`, while `userData.rhsAncient` is pointed at its first element.

If the run is directed to start from a checkpoint file, certain statistical information about the variables will be available, but other initializations must be performed. A call to `OptionsGetString()` quickly determines that in the present case no checkpoint file has been specified on the command line. A similar check for the "`-checkUCD`" option also turns up empty. Accordingly, the code will not attempt to read from a checkpoint file, but will instead compute the deviations of $\Delta^\star\psi$, ψ, $\tilde{I}$, and p from their equilibrium values.

For each poloidal section, the routine first zeroes the `userData.old` instance of $\Delta^\star\chi$ with a call to `VecSet()`. Three calls to `VecWAXPY()` follow, subtracting the `base` instances of $\Delta^\star\psi$, $\tilde{I}$, and p from their respective `old` instances, and storing the results in `deltaC`, `deltatI`, and `deltaP` respectively. Then a call to `VecAYPX()` subtracts `deltaPsi` from `Psi`, storing the result in `deltaPsi`.

After the poloidal loop come five more `GetArrayInPData()` calls, this time for $\Delta^\star\chi$, δC, $\delta\tilde{I}$, δp, and $\delta\psi$. Preliminary preparations are complete, and `initMHD()` returns. ↑`main()`

2.5.7 The Multiple-degree-of-freedom Matrix

Having prepared all relevant variables for iteration, the code now calls function `constructMDoF()` to set up its multiple-degree-of-freedom matrix structure `userMdof`, of type `MDoF`, defined in **mhd/mdof/MDoF.h**. Located in **mhd/mdof/constructMDoF.c**, this routine begins by setting the number of degrees of freedom `ndof` to three. These three degrees of freedom correspond to the toroidal magnetic field, vorticity, and pressure. ↓`constructMDoF()`

The actual matrix will be in sparse AIJ format, so a special descriptor, `matrixInfo`, of type `MatrixAIJ` must be initialized first. It inherits its communicator from the local finite element matrix. It will have `ndof`$= 3$ times as many rows and columns as this matrix. Space is allocated for the two integer arrays that will keep track of the number of nonzero elements on and off the diagonal in each row. An index array is also allocated.

The number-of-nonzeros vectors are initialized as follows: each row corresponding to a toroidal field or pressure has twice as many on- and off-diagonal non-zero entries as the corresponding row in the local finite element matrix. Each row corresponding to a vorticity has three times as many of each type of entry. Once this `matrixInfo` structure has been set up, Petsc is asked to create the Matrix with a call to `MatCreateMPIAIJ()`.

Petsc routine `VecCreateMPI()` is then invoked to create a distributed Petsc vector with one element for each row in the matrix; this will be the `MdofRHS`

vector. `PetscMalloc()` is then called to allocate space for an array of solver contexts. The number of contexts will be the same as the number of poloidal sections. Each context is then initialized with a call to `SLESCreate()`, following which its Krylov solver is initialized by calling `SLESGetKSP()`. A call is made to the self-explanatory `KSPSetInitialGuessNonzero()`, and the convergence tolerances are set to the same values used for the other Krylov solvers in the code (a residual l_2-norm of 10^{-10} of that of the initial residual or 10^{-50} in absolute value) with `KSPSetTolerances()`. These adjustments are finalized with a call to `SLESSetFromOptions()`, and the function returns. ↑`main()`

2.6 Iterating

All the C language initializations are now complete. After a few last formalities — the code reads the command-line option value for `-EngUpBound`, replacing the default value of its unused variable `EngUpBdy` with 10^{-4}; sets the scaling factor for linearization in the `userData` structure to 1.0; and resets the iteration counter `i` to 0 — we are ready to plunge into the Fortran legacy code. The wallclock time taken to execute this main code block will be monitored by reading the system clock time before and after its invocation with `PetscGetTime()`. Once the "before" value is obtained, the Fortran routine `mh3d()` is called and the main loop can begin. ↓`mh3d`

Subroutine `mh3d()` can be found at the very beginning of the $\sim 15,000$-line fixed-form Fortran source file **m3d/code/m1.F**, which comprises virtually all the pure-physics Fortran routines in the code. The "**.F**" suffix is used instead of the more common "**.f**" in order to allow the file to be passed through a C-style preprocessor before compilation. The code takes advantage of this situation by employing preprocessor directives such as `#include`, `#ifdef` and `#define`.

The subroutine `mh3d()` is itself $1,818$ lines long at the time of this writing. It begins by `#include`-ing five local files, most of which define various common blocks: **param1**, **cli1**, **cli2**, **clim**, and **grfblk**. **param1**, as the name implies, defines a number of parameter variables for the code as well as setting the default precision of undeclared floating-point variables to double if the `ALLREAL` and `ALLDOUBLE` compiler flags have been set. **cli1**, the longest of the included files, defines 27 common blocks related to just about all of the code's activities. **cli2** defines 16 more. **clim** adds another brief seven, while **grfblk** adds five.

2.6.1 Reading the Parameter File

There are then three `namelists`. The first is `wdat`, which consists of the code's original parameter variables describing the numerical and physical parameters of the run. The second is `chpar`. It comprises a number of parameters altering the flow of execution that can be set by the user at runtime, including some, e.g. the `qcon` variables, whose meaning is not yet defined by the code but which are included for the convenience of later developers. The third and smallest is

`hall`, evidently a late addition, with 11 variables specifically related to two-fluid corrections to the MHD equations.

The first block of statements in `mh3d()` perform file operations. The required formatted ascii file **wxy** is opened for reading and assigned unit number 12. The contents of this file (in the **mhd/driver** subdirectory) for the example under discussion are presented in Appendix A. The code then opens file **wxi** in its current working directory for reading. Comments in the code indicate that this file was originally associated with the restart function, but it is currently not used. Requiring its presence must therefore be considered a mistake, one that will most likely have been expunged from the latest version of the code by the time this is read. Files **wxi2**, **wxo**, **wxfi**, **wxf**, **wxfc**, **eqfrank**, **wxfm**, **wxd**, **wx3d**, and **eqd** are all opened with a status of "unknown", which means that if they do not already exist (as is generally the case, since they are output files), they will be created. The only one of these that will ever actually be written to, however, is currently **wxo**.

Local integer variable `ifil` is initialized to zero to facilitate later output. Then `ncarcgm()` is called. Located in **m3d/code/mpar1.F**, the Fortran file that handles mesh-specific operations, this tiny routine was originally used to start up the NCAR graphics library for plotting purposes but currently does nothing at all. The next executable statement is a call to `wnllong()`, also in **mpar1.F**, which likewise does nothing; inline comments indicate that it was originally used to set the output record length to 80.

↕`ncarcgm()`

↕`wnllong()`

This flurry of useless activity is followed by the reading-in of the contents of the `wdat` namelist from the **wxy** file, where they occupy lines 2–32. The values are then immediately written out again to **wxo**. A `goto` statement is then used to skip over five lines that are in any case commented out, following which the `chpar` namelist is read from **wxy** (lines 36–56) and written to **wxo**. Another `goto` skips four more lines of comments, and the `hall` namelist (lines 59–61) is read and written. Four more comments are skipped with a `goto`.

2.6.2 Fortran Initializations

Another round of variable initializations now ensues; the variables, the common blocks to which they belong (if any), their assigned values, and their purposes are shown in Table 2.2. Because the **wxy** file specifies a resistivity $\eta = 10^{-3} \neq 0$, the variable `icainc`, from common block `nlwdat`, is set to one, overriding the zero value set for it in the `wdat` namelist. ??? Because `ifull` is equal to one, three more variables from the `ddat` common block are assigned: `icyl` and `ilb3` are set to zero, and `ilb4` is set to one, overriding the value in **wxy**. `nequil`, also from the `ddat` common block, is then set to zero as well, as is `its`, from the `hallsc` block.

⋮

Three tests are conducted to see whether to call `wread()`, `wreadold()`, or `wreadgk()`. All three tests fail, so none of the routines are called. The value of `ku`, from common block `conp`, is stored in local variable `kuold` even though `ku`

Table 2.2:

Variable	Block	Value	Description
`ifull`	`nlwdat`	1	
`impp`	—	1	
`oni`	`hybc`	0.0	
`omi`	`ddat`	0.0	
`wsmall`	—	10^{-10}	
`mi`	`cplot`	`jz` = 93	
`inndg`	`statics`	0	
`timprev`	`statics`	0.0	
`tedv`	`statics`	0.0	
`pgamm`	`statics`	0.0	
`wkin`	`statics`	0.0	Kinetic energy?
`wkiv`	`statics`	0.0	
`drophol`	`statics`	0.0	
`nvdrop`	`statics`	10	
`ncyp`	`statics`	0	
`neiei`	`statics`	0	
`neqeq`	`statics`	0	
`mlowest`	`cochpar`	`mhep` = 1	
`nlowest`	`cochpar`	`nhep` = 1	

has not yet been set. Eventually, this variable will hold the number of poloidal planes stored on the local processor. Then, in order to determine the mesh, `dmesh()` is called.

This routine, which takes no arguments, is located in **mpar1.F**. It includes the same five parameter files as `mh3d()`, so it will share all common blocks. It begins by setting `oni` and `omi` to 0.0 again, a redundant operation, at least in this case. The `wsmall` and `mi` initializations from the calling routine are also repeated. ↓`dmesh`

⋮

Because `itearing`, from the `extra` common block and `wdat` namelist, is equal to four ≥ 2, `initt2()` rather than `initt1()` is called. Located in the same source file, this routine includes all of the Fortran header files except for **clim**, as well as the C header files **flep.h** and **vmec.h**. It begins by placing a call to `initt()`, the first subroutine in **mpar1.F**. ↓`initt2`

`initt()` includes exactly the same header files as `initt2()`. It begins with some Fortran 90-style explicit variable declarations. Two local 100-element 1D arrays of type `DATATYPE2` (i.e. double precision in this case, as set in **param1**, since macros `ALLREAL` and `ALLDOUBLE` are both defined in the local Makefile) are allocated, `xb` and `yb`. Variables `ok` and `ok2` are declared to be of type logical, while `opt` is declared as an integer. The value of `nm`, ..., is copied to `nmesh` ↓`initt`

following which `nm` is reset to one. `pi` is declared as 4 atan(1). Doubles `oni` and `omi` are set to 0.0, while integer `iuns` is set to one. The stage is now set for the interface calls that will read the initial data from the C structures created by ParM3D.

First, `parparms()` is called. This is a C function located in **m3d/interface/p2m.c**. ("p2m" is short for "ParM3D to M3D"; the routines in this file provide the interface between the C [ParM3D] and Fortran [M3D] parts of the code.) It is a very simple and brief routine, making five straightforward assignments that exchange information regarding the distribution of data among processors between the old and new portions of the code. The first assignment sets the `ld` variable from the Fortran common block `ddat` to `map.NumLocalVertices` which, as the reader may recall, is 465 for the present case. The second sets `ldb`, from the same common block, to `map.NumLocalVertices` − `map.NumBoundaryVertices` = 435, i.e., the number of non-boundary, or interior vertices. The third sets `ku`, mentioned above, to `map.NumPhiSections` = 2. The last two assignments go in the opposite direction. The values of Fortran parameters `lz` and `kz`, defined in **param1** to be 5,000 and 12 respectively, are copied to their counterparts in the `userData.parameter` data structure. Parameter `lz` determines the leading dimension of the statically-allocated Fortran work arrays that store variable values at each node; it sets the maximum number of vertices per poloidal plane. `kz`, the second dimension of many of these work arrays, sets the maximum number of poloidal planes per processor. The function now returns. ↓`parparms()` ↑`initt`

The next two functions called by `initt()` are also interface routines from **p2m.c**. First comes `pargrid()`, which, depending on the value of its third (integer) argument, copies the R and z coordinates of all local vertices either from the Fortran storage to the C `grid.R` and `grid.Z` variables or vice versa. Here, the third argument is 2, so the latter operation will be performed. The `xH` and `yH` arrays from the `gridxy` Fortran common block defined in **grfblk** are the destinations of the transfer. Each has dimension `(lz, kz)`. ↓`pargrid()`

The R coordinates are copied first, from `grid.R` to `xH`, by passing the addresses of both to subroutine `par2m3d()`, which is specialized for copying `ParData`-type variables into Fortran arrays. The routine loops over local poloidal sections, and, within that loop, over local vertices, assigning the value from the corresponding element of the `ParData` structure's array to its appropriate spot in the Fortran array. *As written, this could prove to be a fairly expensive operation if repeated often enough — in this relatively small example it must copy 930 eight-byte values one at a time. Given that the data are being copied in order, an optimized memory copying routine such as* `memcpy()` *should probably be tried as a replacement. However, this would require that the destination memory block be contiguous, which would necessitate dynamic rather than static array allocation in the Fortran code. Ideally, of course, the data would only occupy one block of memory and only a pointer would be passed between code sections.* ↓`par2m3d()` ↑`pargrid()`

`pargrid()` now methodically subtracts the plasma major radius from all the Fortran x coordinates. It then copies the values of the ParM3D structure `grid.d2ZdPhi2.array` at all vertices in the zeroth plane to the Fortran array pointed to by its final argument, `rv`. Despite appearances, the values being

copied are *not* those of $\partial^2 z/\partial\varphi^2$, but rather the Vmec "s" coordinates, i.e., the minor radius of each point within the poloidal mesh normalized from zero to one, as previously computed on line 99 of **mesh/grid/constructGrid.c**. The function then returns. ↑`initt()`

After copying over the parameters and the vertex positions, the code swaps some coefficients between data sets with a call to `parcoefs()`. This function simply sets the normalized major radius variable `rmajor` to `AspectRatio` from the `userData parameter` structure; the inverse aspect ratio `eps` to the reciprocal of this value; and the `userData` parameter `timestep` to the `dt` value determined in the Fortran code. ↓`parcoefs()`

The next executable statement sets `nd` to `ku`, the number of planes on the local processor (=2) and sets `nu` to one less than this number. This makes it possible to compute the φ spacing `dphi` $= 2\pi/$`nd` $= \pi$, which is of course wrong; the code should divide by the *total* number of planes to come up with $\pi/4$. Fortunately, this variable is never actually used. ↑`initt()`

⋮

In order to determine the φ coordinate of each local plane, `dzset()` is now called, with no arguments. Located in the same source file, **mpar1.F**, its one line passes the `zH` array to the C routine `dzsetc()` in **p2m.c** to perform the actual calculation. The calculation is quite simple; the routine simply loops over local planes, assigning `zH` for each one based on the lower and upper φ bound information in `cell`. ↓`dzset()` ↕`dzsetc()`

`initt()` next loops over local vertices i, computing the "minor radius" ↑`initt()`

$$r_i = \sqrt{(R_i - R_{major})^2 + z_i^2}$$

for each one and assigning the values to arrays `r` and `rv` while setting the corresponding values in array `tv` to zero. `nmin`, from the `ddat` common block in **cli2**, is set to one.

⋮

The rest of `initt2()` simply duplicates the last block of `initt()`, from the `nmin`→1 assignment onward. It is therefore redundant; the same results would be achieved if `dmesh()` had simply called `initt()` directly. ↑`initt2()`

Appendix A

The wxy File

Line numbers are provided in the right margin for convenience.

```
$WDAT
imsy=0 ifull=1 rmajor=05.00 ncyl=0 nexpl=0 idbug=0
linhe=0 mhep=1 nhep=1 linto=0
pmag=0.00e-3 nkhcy=99991 noscil=099 ikhin=1
iconf=00 delmo=0.11 imapcon=0 bndel=.00 bnkay=1.0
ld=20 ml=1 mu= 9 mp= 20 nu=07 nm=1
dtf= 0.10e-0 nstp=100 npr=100 npu=100000 nparha= 07 nckpa=-2
nread=0 nprnt =10 idrop=0 xgam=0.000 ncont=-12 inistpl=0
eta=1.0e-3 etaout=0.e-5 etaout2=0.0e-5 icheta= 3 petaval=5.e-9
pkkk=1.e-4 lin= 0                                                    10
rmu=1.e-4 rmuout=0.e-4 rmuout2=0.e-4 ichrmu= 3 ibounde= 0 rdtdp= 0.5
vmu=1.e-4 vmuout=1.e-4 vmuout2=1.e-4 ichvmu= 1 idropv= 1 iqpsi=0
gammo=0.00 qn=1.  iaponly=0 vdropm=2.e-90 vdropmi=2.e-90
impa=1 iergod= 0 iharcha=0 resfssm=1.e-99

iheliac=0 ilb3=0 ilb4=0 icyl=0 istea=1 partc=1.0
heliacg=5.  nspla=10 ntk=400 npk=400 rlim=.9

ipe= 1 ifwe=1 grax=0.050 chpower=1.  rstn=.0 pmult=1.00000 umult=1.
iartp=+1 napmax=100 wkinmax=1.e+1 wkivmax=1.e00 iadjtss= 0          20

ivaex= 0 icainc= 0 iripeq= 0 igridch=0
intfrex= 0 tempc1=.0 inipur= 0 ichecb= 0
itfbv= 0 ichop=00 chopr=1.5e3 isetmk= 0 ncombgr=0

nchehar=900000 ainfato=.2e-03 nautoha=0900 iplott=4


ianp=0 igkrd=0 ncygk = 005
```

```
gfb = 1.  gfe =01.0 gfp =07.00e19 gfj=80.e1
rgk1 =0.00 rgk2 = 0.001 rgk3 = 0.0 rgk4 =0.e+00 rgk5 = 0000.
itearing = 4
$
$CHPAR

ich2d=0 ichcinv=-0 icincin=3 ipcinvk=1 ioldinp=0
artpn=-1.10 simpf=1.
pampl=1.00e-00 pbase=1.0e-02 pdissf=0.0

rtrans1 = .25 rtrans2 = 0.80 rtransw = .10
iequadr = 1 ldold= 31 facnog=.8
rcg= .57 deltg= .10 facnisg= 2.5

isoft=0 rotperi=10.0 nsofdet=30 sodista=2.81 soangma=.1167
iece=0 ecedis=.900 ecedet=25
qcon21=1.5 qcon12=1.6 qcon22 =2.50 qcon13=.400 qcon23=0.32 qcondw=2.4
qcon14=0.6 qcon24=0.340 qcon15=.80 qcon25=0.410 qcon26=0.50
i2d=0 mlowest=1 nlowest=0 amldval=.000 pres1=-3.18

qnought=.80 apmulf=-1.5 ipchan=000 hhtb=10.  hhtbl=10.

ibeaneq=0 rfrank=0.  resc3=+0.01807 resc4=1.e-09 iresc5=10 resc6=0.001
iresc7=0500 resc8=1.0e+0 resc9=1.0 resc10=1.0 resc11=0.e-9
resc20=2.1 resc21=0.0 resc22=-.00 resc23=0.3 resc24=0.5
resc25=000.1 resc26=0.000 resc27=9.e-3 resc28=0.8 resc29=2.
ichaald=0 ichacbr=0 ichapre=0 ratioq=1.0
$
$HALL
ihall=0 ihallt=0 ielecp=0 iden=0 idengd=1 ivisc=0
pefac=0.5 xmie0=1.0e-2 hallf=1.0
ietal=0 idench=0
$
$HYBN

im=32 jm=32 km=08 r0=200.  l9z=0.  a9=050.  twod=0
dt9=005.  nm9=0001 nsm=1 sx=0.  sy9=0.  sz=0.  load=2 width=.50 xshift=.0
modem=0 contu=8 pskip=0040 wmax=0.05
nsrc=00000 mmm=000000 iavg=0 ismth=0 ipj=1 iters=4 err9=.0010
bt0=1.  q0=1.  c1=1.  c2=1.  c3=1.  c4=0.
rkk1=050.  irkk2=10 rkk3=0.  rkk4=0001.  rkk5=0.

$
$HYB2

mm1=500000 tste1=.25e-19 msmi1=1.0e-00 q1=1.  lr1=1
```

```
kappan=0.  kappat=0.  vpar=0.  vper=0.0 tpartp=1.
$
$HYB3

trtype=1 ntrm=00 tste1=1.  msmi1=1.  q1=1.  lr1=1
kappan=0.  kappat=0.

$
$HYB4
i=1.  x0=0.8 y0=0.5 z0=0.  u0=0.  vp20=10.
$
```

Index

A
accumInnerBracket(), 50, 53
AccumLocalFE(), 26
addPerturbation(), 47
ALLDOUBLE macro, 57, 59
AllocLocalFEMatrix(), 26
ALLREAL macro, 57, 59
Ampere's law, 6
AOApplicationToPetsc(), 32, 48
AOCreateBasic(), 19
application ordering, 17–18
ApplyBdyonF(), 45
area of a triangular element, 24

B
basis functions, piecewise linear, 24
boundary conditions, 42

C
chpar namelist, 57–58
cli1 file, 57
cli2 file, 57
clim file, 57
config.dat12, 12, 14
Configuration structure, 13
connectivity matrix, 24, 26–27
ConstructComm(), 15
ConstructFEMatrix(), 27
constructGrid(), 21
ConstructLmass(), 35
ConstructLocalFEMatrix(), 23
constructMDoF(), 56
constructMHDdata(), 28
constructMHDsolver(), 33
constructPData(), 22, 29
constructRPData(), 29–30
constructTmpData(), 30

D
dfdphi(), 39
differential operators, definitions, 4
dmesh(), 59
dPhi(), 37, 54
dR(), 40
dR0(), 40, 54
duplicatePData(), 22, 29–30
duplicateRPData(), 29–30
dVector1D(), 14, 26
dVector2D(), 14, 26
dXdphi(), 37, 41, 46–47, 49
dXdphi0(), 54
dXdR(), 46–47, 49
dXdR0(), 54
dXdR_PData(), 40
dXdZ(), 46–47, 49
dXdZ0(), 54
dXdZ_PData(), 40
dzset(), 61
dzsetc(), 61

E
element ordering, 20

F
F equation, solving, 41
Faraday's law, 5
FEMatrix structure, 24
feqn(), 41
findElementOrdering(), 19
findLocalColumnNum(), 27
findOrdering(), 17
findThetaIndexRange(), 19
FiniteElementMatrix structure, 28
FixBdyDR(), 40
fixBdyLocalFE(), 27

`FixBdyXPoisson()`, 52
`fixTwoNeighbors()`, 27
`fMatrix()`, 26
`Free_fMatrix()`, 28
`free_iVector1D()`, 19
Freund, R., 35

G

Galerkin method, 37
`GetArrayInPData()`, 23, 35–36, 40, 46–49, 55
`GetArrayInRPData()`, 36, 39
ghost grids, numbering, 17
Golub, G., 35
grfblk file, 57, 60

H

`hall` namelist, 58

I

`iMatrix()`, 26
index set, definition, 19
`IndexMapping` structure, 16
`indexMapping()`, 16, 18
`initMHD()`, 36
`initProfileVmec()`, 13–14
`initt()`, 59
`initt2()`, 59
`innerBracket()`, 49
`innerBracket0()`, 49
`ISCreateGeneral()`, 35
`ISGlobalToLocalMappingApply()`, 20
`ISLocaltoGlobalMappingApply()`, 22, 32–33, 48
`ISLocaltoGlobalMappingCreate()`, 19
`iVector1D()`, 14, 16, 31, 33

K

`KSPSetInitialGuessNonzero()`, 35, 57
`KSPSetTolerances()`, 35, 57

L

Laplacian operator, 4
`LineIntegralF()`, 44–45
`LocalFE` structure, 23
Longcope, D., 25

M

`main()`, 13
`MapCPUtoPartition()`, 16
`mapIndexToVmecCoordinates()`, 22, 32, 48
mars units, 14
mass matrix, definition, 24
`MatAssemblyBegin()`, 28, 45, 47
`MatAssemblyEnd()`, 28, 45, 47
`MatConvert()`, 34
`MatCreateMPIAIJ()`, 28, 56
`MatDuplicate()`, 34
`MatGetRow()`, 33–34, 45, 47
Mathematica, 4
`MatMult()`, 45–46
`matRestoreBdy()`, 47, 55
`matRestoreInterior()`, 46
`MatRestoreRow()`, 33–34, 45, 47
`MatrixAIJ` structure, 23
`MatrixAIJSetUp()`, 25
`MatSetLocalToGlobalMapping()`, 28
`MatSetValue()`, 45, 47
`MatSetValuesLocal()`, 28
`matZeroBdy()`, 47, 55
`matZeroInterior()`, 45
`MDoF` structure, 56
`Mesh` structure, 21
`mh3d()`, 57
`MHD_basic` structure, 29
`MHD_DATA` structure, 28
`MHD_dphis` structure, 30
`MHD_dRdZs` structure, 30
`MHD_extend` structure, 29
`MHD_RHS` structure, 30
`MHD_Scratch` structure, 30
`MHD_Solver` structure, 33
`MHD_work` structure, 30
`MPI_Allreduce()`, 43–44
`MPI_Comm_vector()`, 15

N

Nachtigal, N., 35
`ncarcgm()`, 58

`NodeDomain` structure, 16
`NormPhi` variable, 22–23, 44
Numerical Recipes, 21

O

Ohm's law, resistive MHD, 7
`OptionsGetDouble()`, 30–31, 47
`OptionsGetInt()`, 13
`OptionsGetString()`, 56
`OptionsHasName()`, 30

P

`par2m3d()`, 60
param1 file, 57
parameter flag, 30–32, 49–50, 54–55
`parcoefs()`, 61
`ParData` structure, 21
`pargrid()`, 60
`parparms()`, 60
Petsc ordering, 17–18
`PetscGetTime()`, 57
`PetscInitialize()`, 13
`PetscMalloc()`, 33–34, 57
`phieqn()`, 49
`phone` variable, initialization, 15
`PLogEventBegin()`, 41, 46, 49
`PLogEventEnd()`, 46–47, 55
`PLogFlops()`, 39–40, 46–47, 50, 52–53, 55
`PLogStageRegister()`, 13
Poisson bracket, definition, 50
`PoissonBracket0()`, 51
`polint()`, 21
`psieqn()`, 46

R

`RParData` structure, 29
run.12, 12

S

`SAME_PRECONDITIONER` flag, 45, 47
`setComm()`, 15
`setOptions()`, 30
`setupIJPonGrid()`, 31, 33
`SLES`, 33–34
`SLESCreate()`, 34, 57
`SLESGetKSP()`, 35, 57
`SLESSetFromOptions()`, 34, 57
`SLESSetOperators()`, 34–35, 45, 47, 55
`SLESSolve()`, 39–40, 46–47, 49, 54–55
Smith, B., 36
sparse matrix format, 26
stiffness matrix, definition, 24
Strauss, H., 25
`SurfaceIntegral0()`, 43–44
`SWEEP`, 20

T

`timeadv()`, 35
`TmpData` structure, 30
`ToroidalUpDate()`, 23

U

unit normal, to a flux surface, 41
`UpdateMassStiff()`, 26

V

`VecArray()`, 22, 29
`VecAssemblyBegin()`, 23, 32, 47–48, 55
`VecAssemblyEnd()`, 23, 32, 47–48, 55
`VecAXPY()`, 39, 50, 53–54
`VecAYPX()`, 46, 56
`VecCopy()`, 36, 39, 43, 46, 50
`VecCreateGhost()`, 22, 29
`VecCreateMPI()`, 17, 30, 56
`VecDestroy()`, 19
`VecDuplicate()`, 22, 29–30
`VecGetArray()`, 26, 30, 36, 43–45
`VecGetOwnershipRange()`, 17
`VecGhostGetLocalForm()`, 26, 43–44
`VecGhostRestoreLocalForm()`, 27, 43–44
`VecGhostUpdateBegin()`, 26, 42–43
`VecGhostUpdateEnd()`, 26, 43–44
`VecMax()`, 48
`VecNorm()`, 43
`VecPointwiseDivide()`, 46

`VecPointwiseMult()`, 32, 43, 50, 53–54
`VecReciprocal()`, 39
`VecRestoreArray()`, 27, 30, 36, 43–44
`VecScale()`, 32, 43, 50
`VecSet()`, 36, 40, 42, 46, 49
`VecSetValue()`, 55
`VecSetValues()`, 23, 32, 48
`VecShift()`, 32, 50
vector potential, 5
`VecWAXPY()`, 56
vertex orderings, 18
Vmec data file, 12, 14
`vmecBfield()`, 32
`VmecData` structure, 14
`vmecfit()`, 21
`vmecfit0()`, 21
`vmecmesh()`, 21
`vmecnorm()`, 23
`vmecpoint2()`, 23, 37
`vmecread()`, 14
`vmecscaling()`, 14

W

`wdat` namelist, 57–58
`wnllong()`, 58
`WorkSpace` structure, 30
`wread()`, 58
`wreadgk()`, 58
`wreadold()`, 58
wxi file, 58
wxo file, 58
wxy file, 58

X

`xDR()`, 37
`xDR0()`, 37
`xDZ()`, 39
`xDZ0()`, 39
`xInnerBracket0()`, 53
`xInnerBracketRPP()`, 53
`xPoissonBracket0()`, 51–52
`xPoissonBracketRPP()`, 50

Z

`zeroFlowField()`, 36