A Tutorial on NIMROD Physics Kernel Code Development

Carl Sovinec
University of Wisconsin–Madison
Department of Engineering Physics

August 1–3, 2001
Madison, Wisconsin

Contents

List of Figures

Preface

The notes contained in this report were written for a tutorial given to members of the NIMROD Code Development Team and other users of NIMROD. The presentation was intended to provide a substantive introduction to the `FORTRAN` coding used by the kernel, so that attendees would learn enough to be able to further develop NIMROD or to modify it for their own applications.

This written version of the tutorial contains considerably more information than what was given in the three half-day sessions. It is the author's hope that these notes will serve as a manual for the kernel for some time. Some of the information will inevitably become dated as development continues. For example, we are already considering changes to the time-advance that will alter the predictor-corrector approach discussed in Section C.2. In addition, the 3.0.3 version described here is not complete at this time, but the only difference reflected in this report from version 3.0.2 is the advance of temperature instead of pressure in Section C.5. For those using earlier versions of NIMROD (prior to and including 2.3.4), note that basis function flexibility was added at 3.0 and data structures and array ordering are different than what is described in Section B.

If you find errors or have comments, please send them to me at **sovinec@engr.wisc.edu**.

Chapter I

Framework of NIMROD Kernel

A Brief Reviews

A.1 Equations

NIMROD started from two general-purpose PDE solvers, Proto and Proteus. Though remnants of these solvers are a small minority of the present code, most of NIMROD is modular and not specific to our applications. Therefore, what NIMROD solves can and does change, providing flexibility to developers and users.

Nonetheless, the algorithm implemented in NIMROD is tailored to solve fluid-based models of fusion plasmas. The equations are the Maxwell equations without displacement current and the single fluid form (see [1]) of velocity moments of the electron and ion distribution functions, neglecting terms of order m_e/m_i smaller than other terms.

Maxwell without displacement current:
Ampere's Law:

$$\mu_0 \mathbf{J} = \nabla \times \mathbf{B}$$

Gauss's Law:

$$\nabla \cdot \mathbf{B} = 0$$

Faraday's Law:

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

Fluid Moments → Single Fluid Form

Quasi-neutrality is assumed for the time and spatial scales of interest, *i.e.*

- $Zn_i \simeq n_e \rightarrow n$

- $e(Zn_i - n_e)\mathbf{E}$ is negligible compared to other forces
- However $\nabla \cdot \mathbf{E} \neq 0$

Continuity:

$$\frac{\partial n}{\partial t} + \mathbf{V} \cdot \nabla n = -n \nabla \cdot \mathbf{V}$$

Center-of-mass Velocity Evolution:

$$\rho \frac{\partial \mathbf{V}}{\partial t} + \rho \mathbf{V} \cdot \nabla \mathbf{V} = \mathbf{J} \times \mathbf{B} - \nabla p - \nabla \cdot \mathbf{\Pi}$$

- $\mathbf{\Pi}$ often set to $\rho \nu \nabla \mathbf{V}$
- Kinetic effects, *e.g.,* neoclassical, would appear through $\mathbf{\Pi}$

Temperature Evolution (3.0.2 & earlier evolve p, p_e)

$$\frac{n_\alpha}{\gamma - 1} \left(\frac{\partial}{\partial t} + \mathbf{V}_\alpha \cdot \nabla \right) T_\alpha = -p_\alpha \nabla \cdot \mathbf{V}_\alpha - \mathbf{\Pi}_\alpha : \nabla \mathbf{V}_\alpha - \nabla \cdot \mathbf{q}_\alpha + Q_\alpha$$

- $\mathbf{q}_\alpha = -n_\alpha \chi_\alpha \cdot \nabla T_\alpha$ in collisional limit
- Q_e includes ηJ^2
- $\alpha = i,\ e$
- $p_\alpha = n_\alpha k T_\alpha$

Generalized Ohm's ($\sim$ electron velocity moment)(underbrace denotes input parameters, unless otherwise noted `ohms=`)

$$\mathbf{E} = \underbrace{-\mathbf{V} \times \mathbf{B}}_{\substack{\texttt{'mhd'} \\ \texttt{''mhd \& hall'} \\ \texttt{'2fl'}}} \underbrace{+\eta \mathbf{J}}_{\texttt{elecd>0}} \underbrace{+\frac{1}{ne} \mathbf{J} \times \mathbf{B}}_{\substack{\texttt{'hall'} \\ \texttt{'mhd \& hall'} \\ \texttt{'2fl'}}} + \frac{m_e}{ne^2} \left[\underbrace{\frac{\partial \mathbf{J}}{\partial t} \underbrace{+\nabla \cdot (\mathbf{JV} + \mathbf{VJ})}_{\texttt{advect='all'}} - \frac{e}{m_e} (\nabla p_e}_{\texttt{'2fl'}} \underbrace{+\nabla \cdot \mathbf{\Pi}}_{\texttt{neoclassical}}) \right]$$

A steady-state part of the solution is separated from the solution fields in NIMROD. For resistive MHD for example, $\partial / \partial t \rightarrow 0$ implies

1. $\rho_s \mathbf{V}_s \cdot \nabla \mathbf{V}_s = \mathbf{J}_s \times \mathbf{B}_s - \nabla p_s + \nabla \cdot \rho_s \nu \nabla \mathbf{V}_s$
2. $\nabla \cdot (n_s \mathbf{V}_s) = 0$
3. $\nabla \times \mathbf{E}_s = \nabla \times (\eta \mathbf{J}_s - \mathbf{V}_s \times \mathbf{B}_s) = 0$
4. $\frac{n_s}{\gamma - 1} \mathbf{V}_s \cdot \nabla T_s = -p_s \nabla \cdot \mathbf{V}_s + \nabla \cdot n_s \chi \cdot \nabla T_s + Q_s$

Notes:

- An equilibrium (Grad-Shafranov) solution is a solution of 1 ($\mathbf{J}_s \times \mathbf{B}_s = \nabla p_s$); 2 – 4
- Loading an 'equilibrium' into NIMROD means that 1 – 4 are assumed. This is appropriate and convenient in many cases, but it's not appropriate when the 'equilibrium' already contains expected contributions from electromagnetic activity.
- The steady-state part may be set to 0 or to the vacuum field.

Decomposing each field into steady and evolving parts and canceling steady terms gives (for MHD)

$$[A \rightarrow A_s + A]$$

$$\frac{\partial n}{\partial t} + \mathbf{V}_s \cdot \nabla n + \mathbf{V} \cdot \nabla n_s + \mathbf{V} \cdot \nabla n = -n_s \nabla \cdot \mathbf{V} - n \nabla \cdot \mathbf{V}_s - n \nabla \cdot \mathbf{V}$$

$$(\rho_s + \rho) \frac{\partial \mathbf{V}}{\partial t} + \rho \left[(\mathbf{V}_s + \mathbf{V}) \cdot \nabla (\mathbf{V}_s + \mathbf{V}) \right] + \rho_s \left[\mathbf{V}_s \cdot \nabla \mathbf{V} + \mathbf{V} \cdot \nabla \mathbf{V}_s + \mathbf{V} \cdot \nabla \mathbf{V} \right]$$
$$= \mathbf{J}_s \times \mathbf{B} + \mathbf{J} \times \mathbf{B}_s + \mathbf{J} \times \mathbf{B} - \nabla p + \nabla \cdot \nu \left[\rho \nabla \mathbf{V}_s + \rho_s \nabla \mathbf{V} + \rho \nabla \mathbf{V} \right]$$

$$\frac{\partial \mathbf{B}}{\partial t} = \nabla \times (\mathbf{V}_s \times \mathbf{B} + \mathbf{V} \times \mathbf{B}_s + \mathbf{V} \times \mathbf{B} - \eta \mathbf{J})$$

$$\frac{(n + n_s)}{\gamma - 1} \frac{\partial T}{\partial t} + \frac{n}{\gamma - 1} \left[(\mathbf{V}_s + \mathbf{V}) \cdot \nabla (T_s + T) \right] + \frac{n_s}{\gamma - 1} \left[\mathbf{V}_s \cdot \nabla T + \mathbf{V} \cdot \nabla T_s + \mathbf{V} \cdot \nabla T \right]$$
$$= -p_s \nabla \cdot \mathbf{V} - p \nabla \cdot \mathbf{V}_s - p \nabla \cdot \mathbf{V}$$
$$+ \nabla \cdot \left[n_s \chi \cdot \nabla T + n \chi \cdot T_s + n \chi \cdot \nabla T \right]$$

Notes:

- The code further separates χ (and other dissipation coefficients ?) into steady and evolving parts.
- The motivation is that time-evolving parts of fields can be orders of magnitude smaller than the steady part, and linear terms tend to cancel, so skipping decompositions would require tremendous accuracy in the large steady part. It would also require complete source terms.
- Nonetheless, decomposition adds to computations per step and code complexity.

General Notes:

1. Collisional closures lead to local spatial derivatives only. Kinetic closures lead to integro-differential equations.
2. $\frac{\partial}{\partial t}$ equations are the basis for the NIMROD time-advance.

A.2 Conditions of Interest

The objective of the project is to simulate the electromagnetic behavior of magnetic confinement fusion plasmas. Realistic conditions make the fluid equations stiff.

- Global magnetic changes occur over the global resistive diffusion time.
- MHD waves propagate 10^4 – 10^{10} times faster.
- Topology-changing modes grow on intermediate time scales.

* All effects are present in equations sufficiently complete to model behavior in experiments.

A.3 Geometric Requirements

The ability to accurately model the geometry of specific tokamak experiments has always been a requirement for the project.

- Poloidal cross section may be complicated.
- Geometric symmetry of the toroidal direction was assumed.
- The Team added:
  - Periodic linear geometry
  - Simply-connected domains

B Spatial Discretization

B.1 Fourier Representation

A pseudospectral method is used to represent the periodic direction of the domain. A truncated Fourier series yields a set of coefficients suitable for numerical computation:

$$A(\phi) = A_0 + \sum_{n=1}^{N} \left(A_n e^{in\phi} + A_n^* e^{-in\phi} \right)$$

or

$$A(z) = A_0 + \sum_{n=1}^{N} \left(A_n e^{i\frac{2\pi n}{L}z} + A_n^* e^{-i\frac{2\pi n}{L}z} \right)$$

All physical fields are real functions of space, so we solve for the complex coefficients, A_n, $n > 0$, and the real coefficient A_0.

- Linear computations find $A_n(t)$ (typically) for a single n-value. (Input '`lin_nmax`', '`lin_nmodes`')
- Nonlinear simulations use FFTs to compute products on a uniform mesh in ϕ and z.

Example of nonlinear product: $\mathbf{E} = -\mathbf{V} \times \mathbf{B}$

- Start with $\mathbf{V}_n$, $\mathbf{B}_n$, $0 \le n \le N$
- FFT gives

$$\mathbf{V}\left(m\frac{\pi}{N}\right),\ \mathbf{B}\left(m\frac{\pi}{N}\right),\ 0 \le m \le 2N-1$$

- Multiply (colocation)

$$\mathbf{E}\left(m\frac{\pi}{N}\right) = -\mathbf{V}\left(m\frac{\pi}{N}\right) \times \mathbf{B}\left(m\frac{\pi}{N}\right),\ 0 \le m \le 2N-1$$

- FFT returns Fourier coefficients

$$\mathbf{E}_n,\ 0 \le n \le N$$

Standard FFTs require $2N$ to be a power of 2.

- Input parameter '`lphi`' determines $2N$.

$$2N = 2^{\texttt{lphi}}$$

- Collocation is equivalent to spectral convolution if aliasing errors are prevented.
- One can show that setting $\mathbf{V}_n, \mathbf{B}_n, \textit{etc.} = 0$ for $n > \frac{2N}{3}$ prevents aliasing from quadratic nonlinearities (products of two functions). $\left[\texttt{nmodes\_total} = \frac{2^{\texttt{lphi}}}{3} + 1\right]$
- Aliasing does not prevent numerical convergence if the expansion converges.

As a prelude to the finite element discretization, Fourier expansions:

1. Substituting a truncated series for $f(\phi)$ changes the PDE problem into a search for the best solution on a restricted solution space.
2. Orthogonality of basis functions is used to find a system of discrete equations for the coefficients.
   e.g. Faraday's law

$$\frac{\partial \mathbf{B}(\phi)}{\partial t} = -\nabla \times \mathbf{E}(\phi)$$

$$\frac{\partial}{\partial t}\left[\mathbf{B}_0 + \sum_{n=1}^{N} \mathbf{B}_n e^{in\phi} + \text{c.c.}\right] = -\nabla \times \left[\mathbf{E}_0 + \sum_{n=1}^{N} \mathbf{E}_n e^{in\phi} + \text{c.c.}\right]$$

$$\text{apply } \int d\phi e^{-in'\phi} \text{ for } 0 \le n' \le N$$

$$2\pi \frac{\partial \mathbf{B}_{n'}}{\partial t} = -\int d\phi \left(\nabla_{\text{pol}} \times \mathbf{E}_{n'} + \frac{in'\hat{\phi}}{R} \times \mathbf{E}_{n'}\right), \ 0 \le n' \le N$$

B.2 Finite Element Representation

Finite elements achieve discretization in a manner similar to the Fourier series. Both are basis function expansions that impose a restriction on the solution space, the restricted spaces become better approximations of the continuous solution space as more basis functions are used, and the discrete equations describe the evolution of the coefficients of the basis functions.

The basis functions of finite elements are low-order polynomials over limited regions of the domain.

$$\alpha_i(x) = \begin{cases} 0 & x < x_{i-1} \\ \frac{x - x_{i-1}}{x_i - x_{i-1}} & x_{i-1} \le x \le x_i \\ \frac{x_{i+1} - x}{x_{i+1} - x_i} & x_i \le x \le x_{i+1} \\ 0 & x > x_{i+1} \end{cases}$$

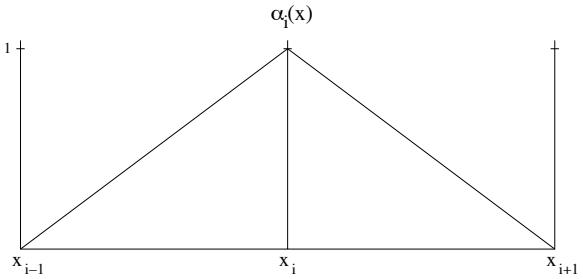


Figure I.1: continuous linear basis function

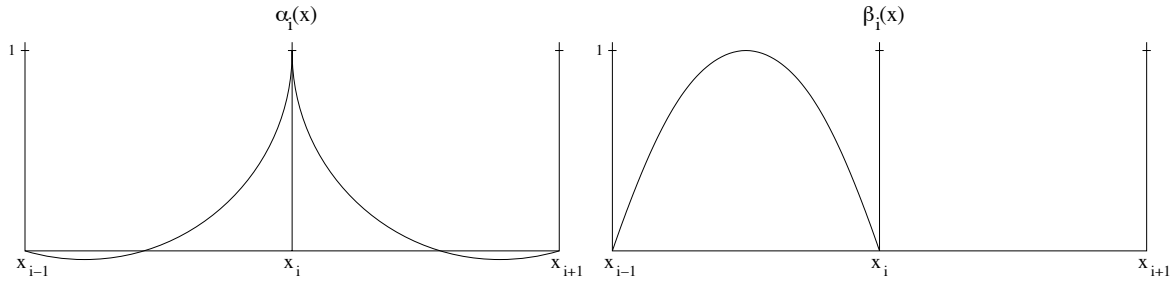


Figure I.2: continuous quadratic basis function

$$\alpha_i(x) = \begin{cases} 0 & x < x_{i-1} \\ \frac{(x-x_{i-1})(x-x_{i-\frac{1}{2}})}{(x_i-x_{i-1})(x_i-x_{i-\frac{1}{2}})} & x_{i-1} \le x \le x_i \\ \frac{(x-x_{i+1})(x-x_{i+\frac{1}{2}})}{(x_i-x_{i+1})(x_i-x_{i+\frac{1}{2}})} & x_i \le x \le x_{i+1} \\ 0 & x > x_{i+1} \end{cases} \qquad \beta_i(x) = \begin{cases} \frac{(x-x_{i-1})(x-x_i)}{(x_{i-\frac{1}{2}}-x_{i-1})(x_{i-\frac{1}{2}}-x_i)} & x_{i-1} \le x \le x_i \\ 0 & \text{otherwise} \end{cases}$$

In a conforming approximation, continuity requirements for the restricted space are the same as those for the solution space for an analytic variational form of the problem.

- Terms in the weak form must be integrable after any integration-by-parts; step functions are admissible, but delta functions are not.
- Our fluid equations require a continuous solution space in this sense, but derivatives need to be piecewise continuous only.

Using the restricted space implies that we seek solutions of the form

$$A(R,Z) = \sum_j A_j \alpha_j (R,Z)$$

- The j-summation is over the nodes of the space.
- $\alpha_j\,(R,Z)$ here may represent different polynomials. [$\alpha_j \& \beta_j$ in 1D quadratic example (Figure I.2)]
- 2D basis functions are formed as products of 1D basis functions for R and Z.

Nonuniform meshing can be handled by a mapping from logical coordinates to physical coordinates. To preserve convergence properties, *i.e.* lowest order polynomials of physical coordinates, the order of the mapping should be no higher than polynomial degree of the basis functions[2].

The present version of NIMROD allows the user to choose the polynomial degree of the basis functions (`poly_degree`) and the mapping (‘`met_spl`’).

Numerical analysis shows that a correct application of the finite element approach ties convergence of the numerical solution to the best possible approximation by the restricted space. (See [2] for analysis of self-adjoint problems.) Error bounds follow from the error bounds of a truncated Taylor series expansion.

For scalar fields, we now have enough basis functions. For vector fields, we also need direction vectors. All recent versions of NIMROD have equations expressed in (R,Z,ϕ) coordinates for toroidal geometries and (X,Y,Z) for linear geometries.

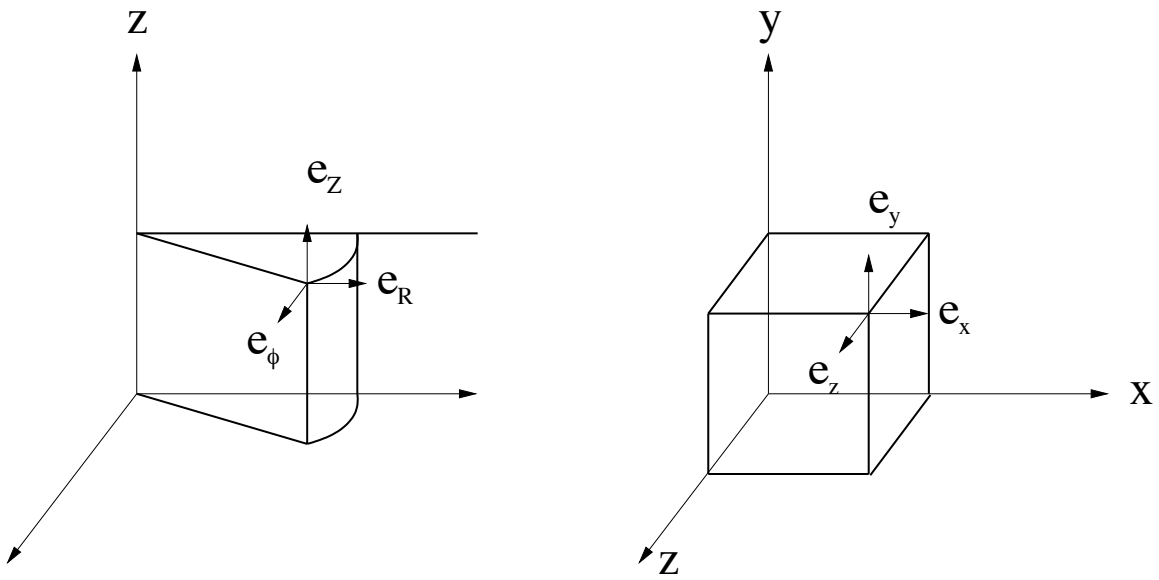


Figure I.3: input `geom=‘toroidal’ & ‘linear’`

For toroidal geometries, we must use the fact that $\hat{e}_R$, $\hat{e}_\phi$ are functions of ϕ, but we always have $\hat{e}_i \cdot \hat{e}_j = \delta_{ij}$.

- Expanding scalars

$$S(R,Z,\phi) \rightarrow \sum_j \alpha_j(R,Z)\left[S_{j0} + \sum_{n=1}^{N} S_{jn}e^{in\phi} + \text{c.c.}\right]$$

- Expanding vectors

$$\mathbf{V}(R,Z,\phi) \rightarrow \sum_j \sum_l \alpha_j(R,Z)\,\hat{e}_l\left[V_{jl0} + \sum_{n=1}^{N} V_{jln}e^{in\phi} + \text{c.c.}\right]$$

  * For convenience $\bar{\alpha}_{jl} \equiv \alpha_j \hat{e}_l$ and I'll suppress c.c. and make n sums from 0 to N, where N here is `nmodes_total`-1.

While our poloidal basis functions do not have orthogonality properties like $e^{in\phi}$ and $\hat{e}_R$, their finite extent means that

$$\iint dRdZ D^s(\alpha_i(R,Z))\, D^t(\alpha_j(R,Z)) \neq 0,$$

where D^s is a spatial derivative operator of allowable order s (here 0 or 1), only where nodes i and j are in the same element (inclusive of element borders).

Integrands of this form arise from a procedure analogous to that used with Fourier series alone; we create a weak form of the PDEs by multiplying by a conjugated basis function.

Returning to Faraday's law for resistive MHD, and using $\vec{\mathcal{E}}$ to represent the ideal $-\mathbf{V}\times\mathbf{B}$ field (after pseudospectral calculations):

- Expand $\mathbf{B}(R,Z,\phi,t)$ and $\vec{\mathcal{E}}(R,Z,\phi,t)$
- Apply $\iiint dRdZd\phi\ \bar{\alpha}_{j'l'}(R,Z)\,e^{-in'\phi}$ for $j' = 1 : \#$ nodes, $l' \epsilon \{R,Z,\phi\}$, $0 \leq n' \leq N$
- Integrate by parts to symmetrize the diffusion term and to avoid inadmissible derivatives of our solution space.

$$\iiint dRdZd\phi\ \left(\bar{\alpha}_{j'l'}e^{-in'\phi}\right)\cdot\sum_{jln}\frac{\partial B_{jln}}{\partial t}\bar{\alpha}_{jl}e^{in\phi}$$

$$= -\iiint dRdZd\phi\nabla\times\left(\bar{\alpha}_{j'l'}e^{-in'\phi}\right)\cdot$$

$$\left[\frac{\eta}{\mu_0}\sum_{jln}B_{jln}\nabla\times\bar{\alpha}_{jl}e^{in\phi} + \sum_{jln}\mathcal{E}_{jln}\bar{\alpha}_{jl}e^{in\phi}\right]$$

$$+\oint\!\!\oint\left(\bar{\alpha}_{j'l'}e^{-in'\phi}\right)\cdot\sum_{jln}\mathbf{E}_{jln}\bar{\alpha}_{jl}e^{in\phi}\times d\mathbf{S}$$

Using orthogonality to simplify, and rearranging the lhs,

$$2\pi\sum_{j}\frac{\partial B_{jl'n'}}{\partial t}\iint dRdZ\ \alpha_{j'}\alpha_{j}$$

$$+\sum_{jl}B_{jln'}\iiint dRdZd\phi\ \frac{\eta}{\mu_0}\nabla\times\left(\bar{\alpha}_{j'l'}e^{-in'\phi}\right)\cdot\nabla\times\left(\bar{\alpha}_{jl}e^{in'\phi}\right)$$

$$=-\iiint dRdZd\phi\ \nabla\times\left(\bar{\alpha}_{j'l'}e^{-in'\phi}\right)\cdot\sum_{jl}\mathcal{E}_{jl}\bar{\alpha}_{jl}e^{in'\phi}$$

$$+\oint\!\!\oint\bar{\alpha}_{j'l'}\cdot\sum_{jl}\mathbf{E}_{jln'}\bar{\alpha}_{jl}\times d\mathbf{S}$$

$$\text{for all}\,\{j',\ l',\ n'\}$$

Notes on this weak form of Faraday's law:

- Surface integrals for inter-element boundaries cancel in conforming approximations.
- The remaining surface integral is at the domain boundary.
- The lhs is written as a matrix-vector product with the $\iint dRdZ\ f\left(\bar{\alpha}_{j'l'}\right)g\left(\bar{\alpha}_{jl}\right)$ integrals defining a matrix element for row (j',l') and column (j,l). These elements are diagonal in n.
- $\{B_{jln'}\}$ constitutes the solution vector at an advanced time (see C).
- The rhs is left as a set of integrals over functions of space.
- For self-adjoint operators, such as $\underline{\mathbf{I}}+\Delta t\nabla\times\frac{\eta}{\mu_0}\nabla\times$, using the same functions for test and basis functions (Galerkin) leads to the same equations as minimizing the corresponding variational [2], *i.e.* Rayleigh-Ritz-Galerkin problem.

B.3 Grid Blocks

For geometric flexibility and parallel domain decomposition, the poloidal mesh can be constructed from multiple grid-blocks. NIMROD has two types of blocks: Logically rectangular blocks of quadrilateral elements are structured. They are called 'rblocks'. Unstructured blocks of triangular elements are called 'tblocks'. Both may be used in the same simulation.

The kernel requires conformance of elements along block boundaries, but blocks don't have to conform.
More notes on weak forms:

- The finite extent of the basis functions leads to sparse matrices
- The mapping for non-uniform meshing has not been expressed explicitly. It entails:
  1. $\iint dRdZ \rightarrow \iint \mathcal{J} d\xi d\eta$, where ξ, η are logical coordinates, and $\mathcal{J}$ is the Jacobian,
  2. $\frac{\partial \alpha}{\partial R} \rightarrow \frac{\partial \alpha}{\partial \xi}\frac{\partial \xi}{\partial R} + \frac{\partial \alpha}{\partial \eta}\frac{\partial \eta}{\partial R}$ and similarly for Z, where α is a low order polynomial of ξ and η with uniform meshing in (ξ, η).
  3. The integrals are computed numerically with Gaussian quadrature $\int_0^1 f(\xi)\, d\xi \rightarrow \sum_i w_i f(\xi_i)$ where $\{w_i, \xi_i\}$, $i = 1, \ldots n_g$ are prescribed by the quadrature method [3].

B.4 Boundary Conditions

Dirichlet conditions are normally enforced for $\mathbf{B}_{normal}$ and $\mathbf{V}$. If thermal conduction is used, Dirichlet conditions are also imposed on T_α. Boundary conditions can be modified for the needs of particular applications, however.

Formally, Dirichlet conditions are enforced on the restricted solution space ("essential conditions"). In NIMROD we simply substitute the Dirichlet boundary condition equations in the linear system at the appropriate rows for the boundary nodes.

Neuman conditions are not set explicitly. The variational aspects enforce Neuman conditions - through the appearance (or non-appearance) of surface integrals - in the limit of infinite spatial resolution ([2] has an excellent description). These "natural" conditions preserve the spatial convergence rate without using ghost cells, as in finite difference or finite volume methods.

C Temporal Discretization

NIMROD uses finite difference methods to discretize time. The solution field is marched in a sequence of time-steps from initial conditions.

C.1 Semi-implicit Aspects

Without going into an analysis, the semi-implicit method as applied to hyperbolic systems of PDEs can be described as a leap-frog approach that uses a self-adjoint differential operator to extend the range of numerical stability to arbitrarily large Δt. It is the tool we use to address the stiffness of the fluid model as applied to fusion plasmas. (See [4, 5].)

Though any self-adjoint operator can be used to achieve numerical stability, accuracy at large Δt is determined by how well the operator represents the physical behavior of the system [4]. The operator used in NIMROD to stabilize the MHD advance is the linear ideal-MHD force operator (no flow) plus a Laplacian with a relatively small coefficient [5].

Like the leap-frog method, the semi-implicit method does not introduce numerical dissipation. Truncation errors are purely dispersive. This aspect makes semi-implicit methods attractive for stiff, low dissipation systems like plasmas.

NIMROD also uses time-splitting. For example, when the Hall electric field is included in a computation, $\mathbf{B}$ is first advanced with the MHD electric field, then with the Hall electric field.

$$\mathbf{B}^{\mathrm{MHD}} = \mathbf{B}^n - \Delta t \nabla \times \mathbf{E}_{\mathrm{MHD}}$$

$$\mathbf{B}^{n+1} = \mathbf{B}^{\mathrm{MHD}} - \Delta t \nabla \times \mathbf{E}_{\mathrm{Hall}}$$

Though it precludes 2^{nd} order temporal convergence, this splitting reduces errors of large time-step [6]. A separate semi-implicit operator is then used in the Hall advance of $\mathbf{B}$. At present, the operator implemented in NIMROD often gives inaccuracies at Δt much larger than the explicit stability limit for a predictor/corrector advance of $\mathbf{B}$ due to $\mathbf{E}_{\mathrm{Hall}}$. Hall terms must be used with great care until further development is pursued.

Why not an implicit approach?

A true implicit method for nonlinear systems converges on nonlinear terms at advanced times, requiring nonlinear system solves at each time step. This may lead to 'the ultimate' time-advance scheme. However, we have chosen to take advantage of the smallness of the solution field with respect to steady parts → the dominant stiffness arises in linear terms.

Although the semi-implicit approach eliminates time-step restrictions from wave propagation, the operators lead to linear matrix equations.

- Eigenvalues for the MHD operator are $\sim 1 + \omega_k^2 \Delta t^2$ where ω_k are the frequencies of the MHD waves supported by the spatial discretization.
- We have run tokamak simulations accurately at $\max|\omega_k|\Delta t \sim 10^4 - 10^5$, and $\min|\omega_k| \simeq 0$.
- Leads to very ill-conditioned matrices.
- Anisotropic terms use steady + $(n = 0)$ fields → operators are diagonal in n.

C.2 Predictor-corrector Aspects

The semi-implicit approach does not ensure numerical stability for advection, even for $\mathbf{V}_s \cdot \nabla$ terms. A predictor-corrector approach can be made numerically stable.

- Nonlinear and linear computations require two solves per equation.
- Synergistic wave-flow effects require having the semi-implicit operator in the predictor step and centering wave-producing terms in the corrector step when there are no dissipation terms [7]
- NIMROD does not have separate centering parameters for wave and advective terms, so we set centerings to $1/2 + \delta$ and rely on having a little dissipation.
- Numerical stability still requires $\Delta t \leq C \frac{\Delta x}{V}$, where C is $O(1)$ and depends on centering. [See `input.f` in the `nimset` directory.]

C.3 Dissipative Terms

Dissipation terms are coded with time-centerings specified through input parameters. We typically use fully forward centering.

C.4 $\nabla \cdot \mathbf{B}$ Constraint

Like many MHD codes capable of using nonuniform, nonorthogonal meshes, $\nabla \cdot \mathbf{B}$ is not identically 0 in NIMROD. Left uncontrolled, the error will grow until it crashes the simulation. We have used an error-diffusion approach [8] that adds the unphysical diffusive term $\kappa_{\nabla \cdot B} \nabla \nabla \cdot \mathbf{B}$ to Faraday's law.

$$\frac{\partial \mathbf{B}}{\partial t} = -\nabla \times \mathbf{E} + \kappa_{\nabla \cdot B} \nabla \nabla \cdot \mathbf{B} \tag{I.1}$$

Applying $\nabla \cdot$ gives

$$\frac{\partial \nabla \cdot \mathbf{B}}{\partial t} = \nabla \cdot \kappa_{\nabla \cdot B} \nabla \left(\nabla \cdot \mathbf{B} \right) \tag{I.2}$$

so that the error is diffused if the boundary conditions maintain constant $\oint \oint d\mathbf{S} \cdot \mathbf{B}$[9].

C.5 Complete time advance

A complete time-step in version 3.0.3 will solve coefficients of the basis functions used in the following time-discrete equations.

$$\left\{\rho^n - \Delta t f_\nu \nabla\cdot\rho^n\nu\nabla - \frac{C_{sm}\Delta t^2}{4}\left[\mathbf{B}_0\times\nabla\times\nabla\times(\mathbf{B}_0\times\mathbf{I}\cdot) - \mathbf{J}_0\times\nabla\times(\mathbf{B}_0\times\mathbf{I}\cdot)\right]\right.$$
$$\left. - \frac{C_{sp}\Delta t^2}{4}\left[\nabla\gamma P_0\nabla\cdot + \nabla\left[\nabla(p_0)\cdot\mathbf{I}\cdot\right]\right] - \frac{C_{sn}\Delta t^2}{4}\nabla^2\right\}(\Delta\mathbf{V})_{pass}$$
$$= -\Delta t\left[\rho^n\mathbf{V}^*\cdot\nabla\mathbf{V}^* + \mathbf{J}^n\times\mathbf{B}^n - \nabla p^n + \nabla\cdot\rho^n\nu\nabla\mathbf{V}^n + \nabla\cdot\mathbf{\Pi}_{neo}\right]$$

- For pass = predict, $\mathbf{V}^* = \mathbf{V}^n$.
- For pass = correct, $\mathbf{V}^* = \mathbf{V}^n + f_v(\Delta\mathbf{V})_{predict}$, and
- $\mathbf{V}^{n+1} = \mathbf{V}^n + (\Delta\mathbf{V})_{correct}$

$$(\Delta n)_{pass} = -\Delta t\left[\mathbf{V}^{n+1}\cdot\nabla n^* + n^*\nabla\cdot\mathbf{V}^{n+1}\right]$$

- pass = predict $\rightarrow n^* = n^n$
- pass = correct $\rightarrow n^* = n^n + f_n(\Delta n)_{predict}$
- $n^{n+1} = n^n + (\Delta n)_{correct}$

Note: f_*'s are centering coefficients, C_{s*}'s are semi-implicit coefficients.

$$\left\{1 + \nabla\times\left(\frac{m_e}{ne^2} + (\Delta t)f_\eta\frac{eta}{\mu_0}\right)\nabla\times - (\Delta t)f_{\nabla\cdot\mathbf{B}}\kappa_{\nabla\cdot\mathbf{B}}\nabla\nabla\cdot\right\}(\Delta\mathbf{B})_{pass}$$
$$= (\Delta t)\nabla\times\left[\mathbf{V}^{n+1}\times\mathbf{B}^* - \frac{n}{\mu_0}\nabla\times\mathbf{B}^n + \frac{1}{ne}\nabla\cdot\mathbf{\Pi}_{e_{neo}}\right] + (\Delta t)\kappa_{\nabla\cdot\mathbf{B}}\nabla\nabla\cdot\mathbf{B}^n$$

- pass = predict $\rightarrow \mathbf{B}^* = \mathbf{B}^n$
- pass = correct $\rightarrow \mathbf{B}^* = \mathbf{B}^n + f_b(\Delta\mathbf{B})_{predict}$
- $\mathbf{B}^{MHD} = \mathbf{B}^n + \Delta\mathbf{B}_{correct}$

$$\left\{1 + \nabla\times\left(\frac{m_e}{ne^2} + C_{sb}(\Delta t)_b^2\frac{|\mathbf{B}_0|}{ne}\right)\nabla\times - \Delta t f_{\nabla\cdot\mathbf{B}}\kappa_{\nabla\cdot\mathbf{B}}\nabla\nabla\cdot\right\}(\Delta\mathbf{B})_{pass}$$
$$= -(\Delta t)_b\nabla\times\left[\frac{1}{ne}(\mathbf{J}^*\times\mathbf{B}^* - \nabla p_e^n) + \frac{m_e}{ne^2}\nabla\cdot(\mathbf{J}^*\mathbf{V}^{n+1} + \mathbf{V}^{n+1}\mathbf{J}^*)\right]$$
$$+(\Delta t)_b\kappa_{\nabla\cdot\mathbf{B}}\nabla\nabla\cdot\mathbf{B}^{MHD}$$

- pass = predict $\rightarrow (\Delta t)_b = \Delta t/2,\ \mathbf{B}^* = \mathbf{B}^{MHD}$
- pass = correct $\rightarrow (\Delta t)_b = \Delta t,\ \mathbf{B}^* = \mathbf{B}^{MHD} + (\Delta \mathbf{B})_{predict}$
- $\mathbf{B}^{n+1} = \mathbf{B}^{MHD} + (\Delta B)_{correct}$

$$\left\{ \frac{n^{n+1}}{\gamma - 1} + (\Delta t) f_\chi \nabla \cdot n^{n+1} \chi_\alpha \cdot \nabla \right\} (\Delta T_\alpha)_{pass}$$
$$= -\Delta t \left[\frac{n^{n+1}}{\gamma - 1} \mathbf{V}_\alpha^{n+1} \cdot \nabla T_\alpha^* + p_\alpha^n \nabla \cdot \mathbf{V}_\alpha^{n+1} - \nabla \cdot n^{n+1} \chi_\alpha \cdot \nabla T_\alpha^n \right] + Q$$

- pass = predict $\rightarrow T_\alpha^* = T_\alpha^n$
- pass = correct $\rightarrow T_\alpha^* = T_\alpha^n + f_T (\Delta T_\alpha)_{predict}$
- $T_\alpha^{n+1} = T_\alpha^n + (\Delta T_\alpha)_{correct}$
- $\alpha = e, i$

Notes on time-advance in 3.0.3

- Without the Hall time-split, approximate time-centered leap-frogging could be achieved by using $\frac{1}{2}\left(n^n + n^{n+1}\right)$ in the temperature equation

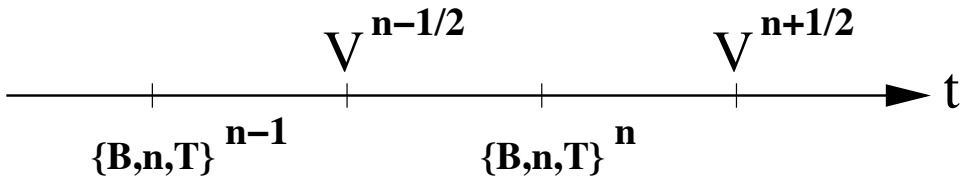


Figure I.4: schematic of leap-frog

- The different centering of the Hall time split is required for numerical stability (see Nebel notes).
- the $\frac{\partial \mathbf{J}}{\partial t}$ electron inertia term necessarily appears in both time-splits for $\mathbf{B}$ when it is included in $\mathbf{E}$.
- Fields used in semi-implicit operators are updated only after changing more than a set level to reduce unnecessary compution of the matrix elements (which are costly).

D Basic functions of the NIMROD kernel

1. Startup
   - Read input parameters
   - Initialize parallel communication as needed
   - Read grid, equilibrium, and initial fields from `dump` file
   - Allocate storage for work arrays, matrices, etc.
2. Time Advance
   (a) Preliminary Computations
      - Revise Δt to satisfy `CFL` if needed
      - Check how much fields have changed since matrices were last updated. Set flags if matrices need to be recomputed
      - Update neoclassical coefficients, voltages, etc. as needed

   (b) Advance each equation
      - Allocate `rhs` and solution vector storage
      - Extrapolate in time previous solutions to provide an initial guess for the linear solve(s).
      - Update matrix and preconditioner if needed
      - Create `rhs` vector
      - Solve linear system(s)
      - Reinforce Dirichlet conditions
      - Complete regularity conditions if needed
      - Update stored solution
      - Deallocate temporary storage
3. Output
   - Write global, energy, and probe diagnostics to binary and/or text files at desired frequency in time-step (`nhist`)
   - Write complete dump files at desired frequency in time-step (`ndump`)
4. Check stop condition
   - Problems such as lack of iterative solver convergence can stop simulations at any point

Chapter II

FORTRAN Implementation

Preliminary remarks

As implied in the Framework section, the NIMROD kernel must complete a variety of tasks each time step. Modularity has been essential in keeping the code manageable as the physics model has grown in complexity.

FORTRAN 90 extensions are also an important ingredient. Arrays of data structures provide a clean way to store similar sets of data that have differing numbers of elements. FORTRAN 90 modules organize different types of data and subprograms. Array syntax leads to cleaner code. Use of KIND when defining data settles platform-specific issues of precision. ([10] is recommended.)

A Implementation map

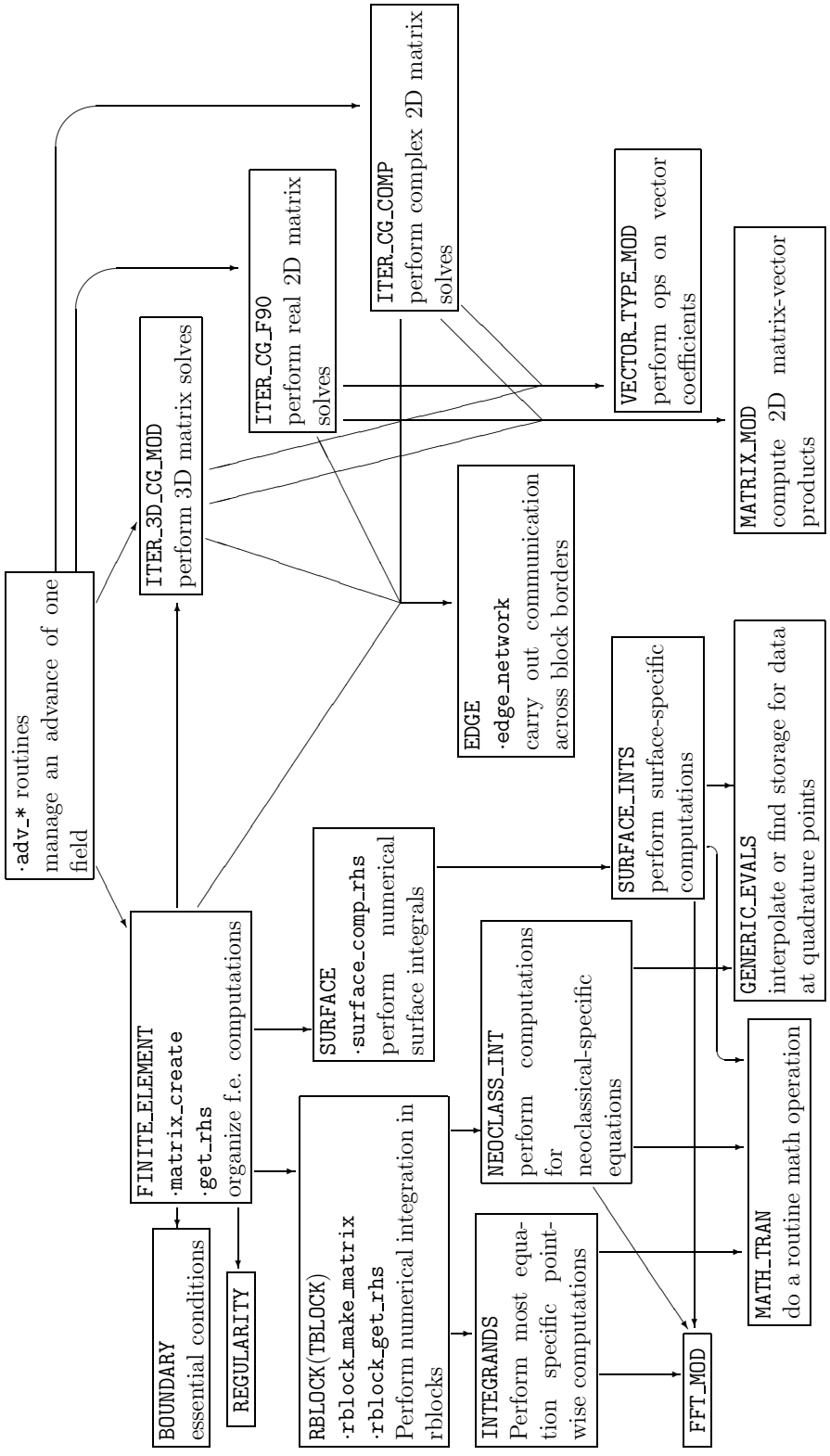
All computationally intensive operations arise during either 1) the finite element calculations to find rhs vectors and matrices, and 2) solution of the linear systems. Most of the operations are repeated for each advance, and equation-specific operations occur at only two levels in a hierarchy of subprograms for 1). The data and operation needs of subprograms at a level in the hierarchy tend to be similar, so FORTRAN module grouping of subprograms on a level helps orchestrate the hierarchy.

B Data storage and manipulation

B.1 Solution Fields

The data of a NIMROD simulation are the sets of basis functions coefficients for each solution field plus those for the grid and steady-state fields. This information is stored in FORTRAN 90 structures that are located in the fields module.

Implementation Map of Primary NIMROD Operations



NOTE:1. all caps indicate a module name. 2.'·' indicates a subroutine name or interface to module procedure

- `rb` is a 1D pointer array of defined-type `rblock_type`. At the start of a simulation, it is dimensioned `1:nrbl`, where `nrbl` is the number of rblocks.
- The type `rblock_type` (analogy:integer), is defined in module `rblock_type_mod`. This is a structure holding information such as the cell-dimensions of the block(`mx,my`), arrays of basis function information at Gaussian quadrature nodes (`alpha,dalpdr,dalpdz`), numerical integration information(`wg,xg,yg`), and two sets of structures for each solution field:
  1. A `lagr_quad_type` structure holds the coefficients and information for evaluating the field at any (ξ, η).
  2. An `rb_comp_qp_type`, just a pointer array, allocated to save the interpolates of the solution spaces at the numerical quadrature points.
- `tb` is a 1D pointer array, similar to `rb` but of type `tblock_type_mod`, dimensioned `nrbl+1:nbl` so that `tblocks` have numbers distinct from the `rblocks`.

Before delving into the element-specific structures, let's backup to the `fields` module level. Here, there are also 1D pointer arrays of `cvector_type` and `vector_type` for each solution and steady-state field, respectively. These structures are used to give block-type-independent access to coefficients. They are allocated `1:nbl`. These structures hold pointer arrays that are used as pointers (as opposed to using them as `ALLOCATABLE` arrays in structures). There are pointer assignments from these arrays to the `lagr_quad_type` structure arrays that have allocated storage space. (See [10])

In general, the element-specific structures(`lagr_quad_type`, here) are used in conjunction with the basis functions (as occurs in getting field values at quadrature point locations). The element-independent structure(`vector_types`) are used in linear algebra operations with the coefficients.

The `lagr_quad_types` are defined in the `lagr_quad_mod` module, along with the interpolation routines that use them. [The `tri_linear_types` are the analogous structures for triangular elements in the `tri_linear` module.] The `lagr_quad_2D_type` is used for ϕ-independent fields.

The structures are self-describing, containing array dimensions and character variables for names. There are also some arrays used for interpolation. The main storage locations are the `fs`, `fsh`, `fsv`, and `fsi` arrays.

- `fs(iv,ix,iy,im)` $\rightarrow$ coefficients for grid-vertex nodes.
  - `iv` = direction vector component (`1:nqty`)
  - `ix` = horizontal logical coordinate index (`0:mx`)
  - `iy` = vertical logical coordinate index (`0:my`)

  - `im` = Fourier component ("mode") index (`1:nfour`)

- `fsh(iv,ib,ix,iy,im)` → coefficients for horizontal side nodes.
  - `iv` = direction vector component (`*:*`)
  - `ib` = basis index (`1:n_side`)
  - `ix` = horizontal logical coordinate index (`1:mx`)
  - `iy` = vertical logical coordinate index (`0:my`)
  - `im` = Fourier component ("mode") index (`1:nfour`)
- `fsv(iv,ib,ix,iy,im)` → coefficients for vertical side nodes.
- `fsi(iv,ib,ix,iy,im)` → coefficients for internal nodes [`ib` limit is (`1:n_int`)]

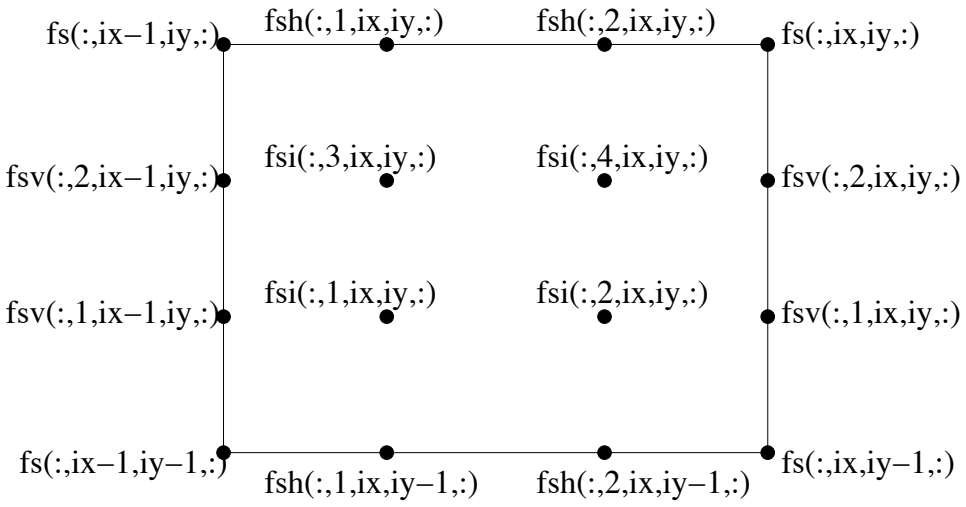


Figure II.1: Basic Element Example. (`n_side=2`, `n_int=4`)

B.2 Interpolation

The `lagr_quad_mod` module also contains subroutines to carry out the interpolation of fields to arbitrary positions. This is a critical aspect of the finite element algorithm, since interpolations are needed to carry out the numerical integrations.

For both real and complex data structures, there are single point evaluation routines (`*_eval`) and routines for finding the field at the same offset in every cell of a block (`*_all_eval` routines). All routines start by finding $\alpha_i|_{(\xi',\eta')}$ [and $\frac{\partial}{\partial\xi}\alpha_i|_{(\xi',\eta')}$ and $\frac{\partial}{\partial\eta}\alpha_i|_{(\xi',\eta')}$

if requested] for the (element–independent) basis function values at the requested offset (ξ', η'). The `lagr_quad_bases` routine is used for this operation. [It's also used at startup to find the basis function information at quadrature point locations, stored in the `rb%alpha`, `dalpdr`, `dalpdz` arrays for use as test functions in the finite element computations.] The interpolation routines then multiply basis function values with respective coefficients.

The data structure, logical positions for the interpolation point, and desired derivative order are passed into the single-point interpolation routine. Results for the interpolation are returned in the structure arrays `f`, `fx`, and `fy`.

The `all_eval` routines need logical offsets (from the lower-left corner of each cell) instead of logical coordinates, and they need arrays for the returned values and logical derivatives in each cell.

See routine `struct_set_lay` in `diagnose.f` for an example of a single point interpolation call. See `generic_3D_all_eval` for an example of an `all_eval` call.

Interpolation operations are also needed in `tblocks`. The computations are analogous but somewhat more complicated due to the unstructured nature of these blocks. I won't cover `tblocks` in any detail, since they are due for a substantial upgrade to allow arbitrary polynomial degree basis functions, like `rblocks`. However, development for released versions must be suitable for simulations with `tblocks`.

B.3 `Vector_types`

We often perform linear algebra operations on vectors of coefficients. [Management routines, iterative solves, etc.] `Vector_type` structures simplify these operations, putting block and basis-specific array idiosyncrasies into module routines. The developer writes one call to a subroutine interface in a loop over all blocks.

Consider an example from the `adv_b_iso` management routine. There is a loop over Fourier components that calls two real 2D matrix solves per component. One solves α for $\mathrm{Re}\{B_R\}$, $\mathrm{Re}\{B_Z\}$, and $-\mathrm{Im}\{B_\phi\}$ for the `F-comp`, and the other solves for $\mathrm{Im}\{B_R\}$, $\mathrm{Im}\{B_Z\}$, and $\mathrm{Re}\{B_\phi\}$. After a solve in a predictor step (`b_pass`=`bmhd pre` or `bhall pre`), we need to create a linear combination of the old field and the predicted solution. At this point,

- `vectr` (`1:nbl`) → work space, 2D real `vector_type`
- `sln` (`1:nbl`) → holds linear system solution, also a 2D `vector_type`
- `be` (`1:nbl`) → pointers to $\mathbf{B}^n$ coefficients, 3D (complex) `vector_type`
- `work1` (`1:nbl`) → pointers to storage that will be $\mathbf{B}^*$ in corrector step, 3D `vector_type`

What we want is $\mathbf{B}^*_{\mathrm{imode}} = f_b \mathbf{B}^{\mathrm{pre}}_{\mathrm{imode}} + (1 - f_b)\mathbf{B}^n_{\mathrm{imode}}$. So we have

- `CALL vector_assign_cvec(vectr(ibl), be(ibl), flag, imode)`

  - Transfers the `flag`-indicated components such as `r12mi3` for $\mathrm{Re}\{B_R\}$, $\mathrm{Re}\{B_Z\}$, and $-\mathrm{Im}\{B_\phi\}$ for imode to the 2D vector.

- `CALL vector_add(sln(ibl), vectr(ibl), v1fac='center', v2fac='1-center')`

  - Adds `'1-center'*vectr` to `'center'*sln` and saves results in `sln`.

- `CALL cvector_assign_vec(work1(ibl), sln(ibl), flag, imode)`

  - Transfer linear combination to appropriate vector components and Fourier component of `work1`.

Examining `vector_type_mod` at the end of the `vector_type_mode.f` file, 'vector_add is defined as an interface label to three subroutines that perform the linear combination for each `vector_type`. Here we interface `vector_add_vec`, as determined by the type definitions of the passed arrays. This subroutine multiplies each coefficient array and sums them.

Note that we did not have to do anything different for `tblocks`, and coding for coefficients in different arrays is removed from the calling subroutine.

Going back to the end of the `vector_type_mode.f` file, note that '=' is overloaded, so we can write statements like

```
work1(ibl)  =  be(ibl)     if arrays conform
   be(ibl)  =  0
```

which perform the assignment operations component array to component array or scalar to each component array element.

B.4 Matrix_types

The `matrix_type_mod` module has definitions for structures used to store matrix elements and for structures that store data for preconditioning iterative solves. The structures are somewhat complicated in that there are levels containing pointer arrays of other structures. The necessity of such complication arises from varying array sizes associated with different blocks and basis types.

At the bottom of the structure, we have arrays for matrix elements that will be used in matrix-vector product routines. Thus, array ordering has been chosen to optimize these product routines which are called during every iteration of a conjugate gradient solve. Furthermore, each lowest-level array contains all matrix elements that are multiplied with a grid-vertex, horizontal-side, vertical-side, or cell-interior vector_type array (`arr`, `arrh`, `arrv`, and `arri` respectively) i.e. all coefficients for a single "basis-type" within a block. The matrices are 2D, so there are no Fourier component indices at this level, and our basis functions do not produce matrix couplings from the interior of one grid block to the interior of another.

The 6D matrix is unimaginatively named `arr`. Referencing one element appears as

```
arr(jq,jxoff,jyoff,iq,ix,iy)
```

where

`iq`	=	'quantity' row index
`ix`	=	horizontal-coordinate row index
`iy`	=	vertical-coordinate row index
`jq`	=	'quantity' column index
`jxoff`	=	horizontal-coordinate column offset
`jyoff`	=	vertical-coordinate column offset

For grid-vertex to grid-vertex connections $0 \leq ix \leq mx$ and $0 \leq iy \leq my$, `iq` and `jq` correspond to the vector index dimension of the vector_type. (simply $1:3$ for A_R, A_Z, A_ϕ or just 1 for a scalar). The structuring of rblocks permits offset storage for the remaining indices, giving dense storage for a sparse matrix without indirect addressing. The column coordinates are simply

```
jx = ix + jxoff
jy = iy + jyoff
```

For each dimension, vertex to vertex offsets are $-1 \leq \mathtt{j} * \mathtt{off} \leq 1$, with storage zeroed out where the offset extends beyond the block border. The offset range is set by the extent of the corresponding basis functions.

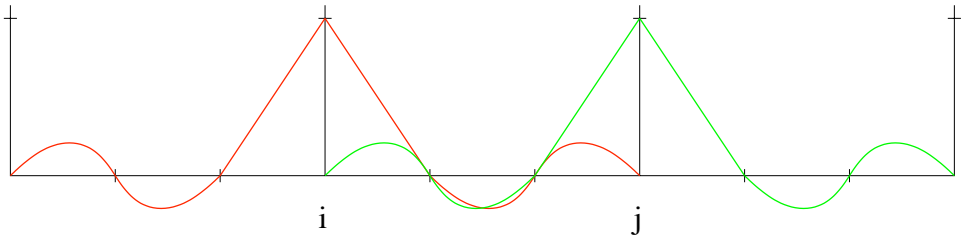


Figure II.2: 1-D Cubic : extent of α_i in red and α_j in green, integration gives `joff`$= -1$

Offsets are more limited for basis functions that have their nonzero node in a cell interior.

Integration here gives `joff`=-1 matrix elements, since cells are numbered from 1, while vertices are numbered from 0. Thus for grid-vertex-to-cell connections $-1 \leq$ `jxoff` ≤ 0, while for cell-to-grid-vertex connections $0 \leq$ `jxoff` ≤ 1. Finally, `j*off`=0 for cell-to-cell. The 'outer-product' nature implies unique offset ranges for grid-vertex, horizontal-side, vertical-side, and cell-interior centered coefficients.

Though offset requirements are the same, we also need to distinguish different bases of the same type, such as the two sketched here.

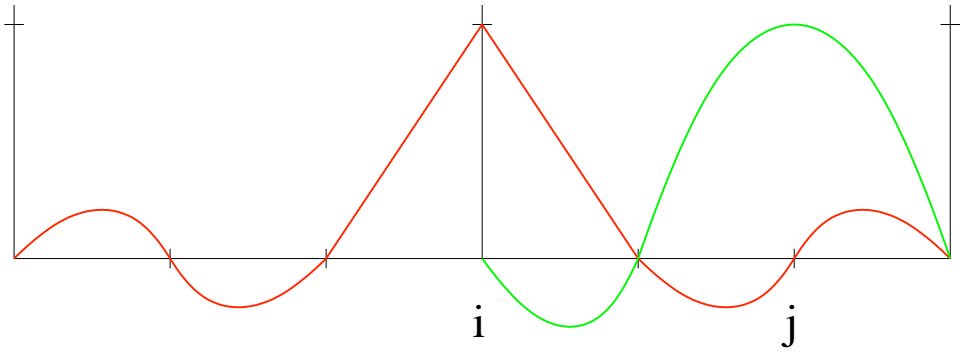


Figure II.3: 1-D Cubic : extent of α_i in red and β_j in green, note j is an internal node

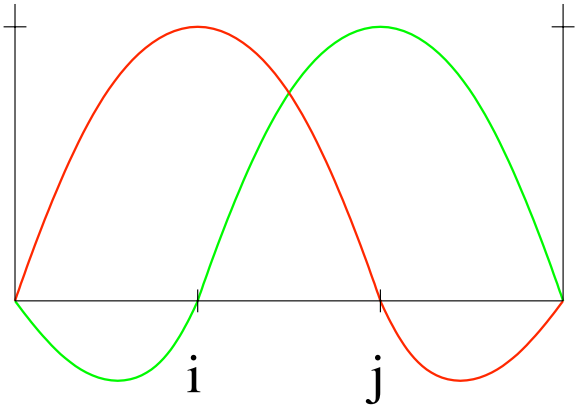


Figure II.4: 1-D Cubic : extent of β_i in red and β_j in green, note both i, j are internal nodes

For `lagr_quad_type` storage and `vector_type` pointers, this is accomplished with an extra array dimension for side and interior centered data. For matrix-vector multiplication it is convenient and more efficient to collapse the vector and basis indices into a single index with the vector index running faster.

Examples:

- Bi-cubic grid vertex to vertical side for P
  `jq`=1, $1 \leq$ `iq` ≤ 2
- Bi-quartic cell interior to horizontal side for $\vec{B}$
  $1 \leq$ `jq` ≤ 27, $1 \leq$ `iq` ≤ 9

Storing separate arrays for matrix elements connecting each of the basis types therefore lets us have contiguous storage for nonzero elements despite different offset and basis dimensions.

Instead of giving the arrays 16 different names, we place `arr` in a structure contained by a 4×4 array `mat`. One of the matrix element arrays is then referenced as

```
mat(jb,ib)%arr
```

where

`ib`=basis-type row index

`jb`=basis-type column index

and $1 \leq$ `ib` ≤ 4, $1 \leq$ `jb` ≤ 4. The numeral coding is

1≡grid-vertex type

2≡horizontal-vertex type

3≡vertical-vertex type

4≡cell-interior type

Bilinear elements are an exception. There are only grid-vertex bases, so `mat` is a 1×1 array.

This level of the matrix structure also contains self-descriptions, such as `nbtype`=1 or 4; starting horizontal and vertical coordinates (=0 or 1) for each of the different types; `nb_type(1:4)` is equal to the number of bases for each type (`poly_degree`-1 for types 2-3 and (`poly_degree` $-1)^2$ for type 4); and `nq_type` is equal to the quantity-index dimension for each type.

All of this is placed in a structure, `rbl_mat`, so that we can have on array of `rbl_mat`'s with one component component per grid block. A similar `tbl_mat` structure is also set at this level, along with self-description.

Collecting all `block_mats` in the `global_matrix_type` allows one to define an array of these structures with one array component for each Fourier component.

This is a `1D` array since the only stored matrices are diagonal in the Fourier component index, i.e. `2D`.

`matrix_type_mod` also contains definitions of structures used to hold data for the pre-conditioning step of the iterative solves. NIMROD has several options, and each has its own storage needs.

So far, only `matrix_type` definitions have been discussed. The storage is located in the `matrix_storage_mod` module. For most cases, each equation has its own matrix storage that holds matrix elements from time-step to time-step with updates as needed. Anisotropic operators require complex arrays, while isotropic operators use the same real matrix for two sets of real and imaginary components, as in the **B** advance example in II.B.3.

The `matrix_vector` product subroutines are available in the `matrix_mod` module. Interface `matvec` allows one to use the same name for different data types. The large number of low-level routines called by the interfaced `matvec`s arose from optimization. The operations were initially written into one subroutine with adjustable offsets and starting indices. What's there now is ugly but faster due to having fixed index limits for each basis type resulting in better compiler optimization.

Note that tblock `matrix_vector` routines are also kept in `matrix_mod`. They will undergo major changes when tblock basis functions are generalized.

The `matelim` routines serve a related purpose - reducing the number of unknowns in a linear system prior to calling an iterative solve. This amounts to a direct application of matrix partitioning. For the linear system equation,

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{II.1}$$

where A_{ij} are sub-matrices, x_j and b_i are the parts of vectors, and A and A_{22} are invertible,

$$x2 = A_{22}^{-1}(b_2 - A_{21}x_1) \tag{II.2}$$

$$\left[\underbrace{A_{11} - A_{12}A_{22}^{-1}A_{21}}_{Schur\ component}\right] x_1 = b_1 - A_{12}A_{22}^{-1}b_2 \tag{II.3}$$

gives a reduced system. If A_{11} is sparse but $A_{11} - A_{12}A_{22}^{-1}A_{21}$ is not, partitioning may slow the computation. However, if $A_{11} - A_{12}A_{22}^{-1}A_{21}$ has the same structure as A_{11}, it can help. For cell-interior to cell-interior sub-matrices, this is true due to the single-cell extent of interior basis functions. Elimination is therefore used for `2D` linear systems when `poly_degree`> 1.

$\star\star$**Additional notes regarding matrix storage:** the stored coefficients are not summed across block borders. Instead, vectors are summed, so that $\mathbf{A}x$ is found through operations separated by block, followed by a communication step to sum the product vector across block borders.

B.5 Data Partitioning

Data partitioning is part of a philosophy of how NIMROD has been - and should continue to be - developed. NIMROD's kernel in isolation is a complicated code, due to the objectives of the project and chosen algorithm. Development isn't trivial, but it would be far worse without modularity.

NIMROD's modularity follows from the separation of tasks described by the implementation map. However, data partitioning is the linchpin in the scheme. Early versions of the code had all data storage together. [*see the discussion of version 2.1.8 changes in the kernel's* ***README*** *file.*]. While this made it easy to get some parts of the code working, it

encouraged repeated coding with slight modifications instead of taking the time to develop general purpose tools for each task. Eventually it became too cumbersome, and the changes made to achieve modularity at 2.1.8 were extensive.

Despite many changes since, the modularity revision has stood the test of time. Changing terms or adding equations requires relatively minimal amount of coding. But, even major changes, like generalizing finite element basis functions, are tractable. The scheme will need to evolve as new strides are taken. [Particle closures come to mind.] However, thought and planning for modularity and data partitioning reward in the long term.

NOTE

- These comments relate to coding for release. If you are only concerned about one application, do what is necessary to get a result
- However, in many cases, a little extra effort leads to a lot more usage of development work.

B.6 Seams

The seam structure holds information needed at the borders of grid blocks, including the connectivity of adjacent blocks and geometric data and flags for setting boundary conditions. The format is independent of block type to avoid block-specific coding at the step where communication occurs.

The `seam_storage_mod` module hold the seam array, a list of blocks that touch the boundary of the domain (`exblock_list`), and `seam0`. The `seam0` structure is of `edge_type`, like the seam array itself. `seam0` was intended as a functionally equivalent seam for all space beyond the domain, but the approach hampered parallelization. Instead, it is only used during initialization to determine which vertices and cell sides lie along the boundary. We still require `nimset` to create `seam0`.

As defined in the `edge_type_mod` module, an `edge_type` holds vertex and segment structures. Separation of the border data into a vertex structure and segment structure arises from the different connectivity requirements; only two blocks can share a cell side, while many blocks can share a vertex. The three logical arrays: `expoint`, `excorner`, and `r0point`, are flags to indicate whether a boundary condition should be applied. Additional arrays could be added to indicate where different boundary conditions should be applied.

Aside *For flux injection boundary conditions, we have added the logical array* `applV0(1:2,:)`. *This logical is set at nimrod start up and is used to determine where to apply injection current and the appropriate boundary condition. The first component applies to the vertex and the second component to the associated segment.*

The seam array is dimensioned from one to the number of blocks. For each block the vertex and segment arrays have the same size, the number of border vertices equals the

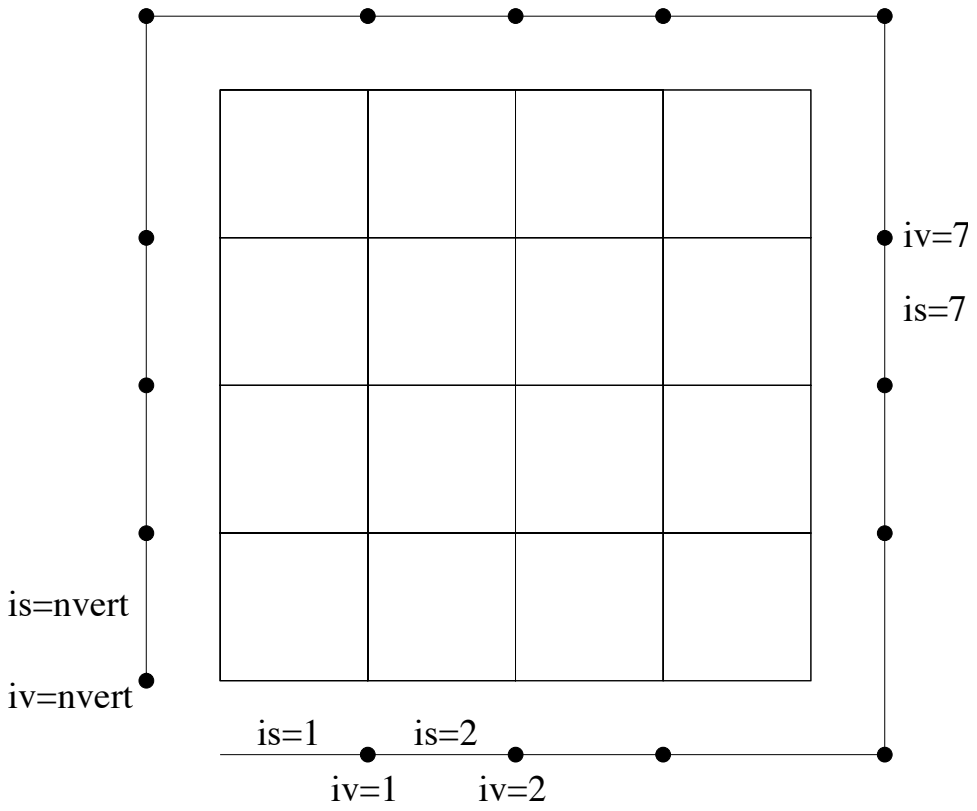


Figure II.5: `iv`=vertex index, `is`=segment index, `nvert`=count of segment and vertices

number of border segments, but the starting locations are offset (see Figure II.5). For rblocks, the ordering is unique.

Both indices proceed counter clockwise around the block. For tblocks, there is no requirement that the scan start at a particular internal vertex number, but the seam indexing always progresses counter clockwise around its block. (entire domain for `seam0`)

Within the `vertex_type` structure, the `ptr` and `ptr2` arrays hold the connectivity information. The `ptr` array is in the original format with connections to `seam0`. It gets defined in `nimset` and is saved in dump files. The `ptr2` array is a duplicate of `ptr`, but references to `seam0` are removed. It is defined during the startup phase of a nimrod simulation and is not saved to dump files. While `ptr2` is the one used during simulations, `ptr` is the one that must be defined during pre-processing. Both arrays have two dimensions. The second index selects a particular connection, and the first index is either: 1 to select (global) block numbers (running from 1 to `nbl_total`, not 1 to `nbl`, see information on parallelization), or 2 to select seam vertex number. (see Figure II.6)

Connections for the one vertex common to all blocks are stored as:

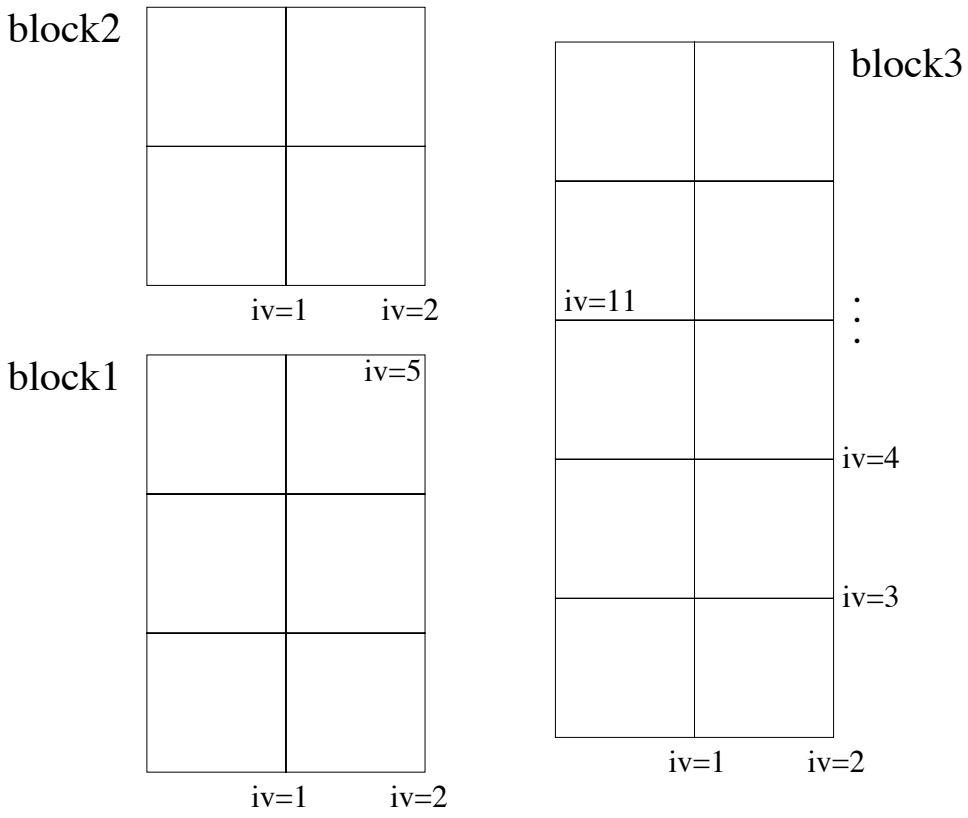


Figure II.6: 3 blocks with corresponding vertices

- seam(1)%vertex(5)
  - seam(1)%vertex(5)% ptr(1,1) = 3
  - seam(1)%vertex(5)% ptr(2,1) = 11
  - seam(1)%vertex(5)% ptr(1,2) = 2
  - seam(1)%vertex(5)% ptr(2,2) = 2
- seam(2)%vertex(2)
  - seam(2)%vertex(2)%ptr(1,1) = 1
  - seam(2)%vertex(2)%ptr(2,1) = 5
  - seam(2)%vertex(2)%ptr(1,2) = 3
  - seam(2)%vertex(2)%ptr(2,2) = 11
- seam(3)%vertex(11)

  - seam(3)%vertex(11)%ptr(1,1) = 2
  - seam(3)%vertex(11)%ptr(2,1) = 2
  - seam(3)%vertex(11)%ptr(1,2) = 1
  - seam(3)%vertex(11)%ptr(2,2) = 5

Notes on `ptr`:

- It does not hold a self-reference, e.g. there are no references like:
  - seam(3)%vertex(11)%ptr(1,3) = 3
  - seam(3)%vertex(11)%ptr(2,3) = 11
- The order of the connections is arbitrary

The `vertex_type` stores interior vertex labels for its block in the `intxy` array. In rblocks `vertex(iv)%intxy(1:2)` saves `(ix,iy)` for seam index `iv`. In `tblocks`, there is only one block-interior vertex label. `intxy(1)` holds its value, and `intxy(2)` is set to 0.

Aside: Though discussion of tblocks has been avoided due to expected revisions, the `vector_type` section should have discussed their index ranges for data storage. Triangle vertices are numbered from 0 to `mvert`, analogous to the logical coordinates in rblocks. The list is 1D in tblocks, but an extra array dimension is carried so that `vector_type` pointers can be used. For vertex information, the extra index follows the rblock vertex index and has the ranges 0:0. The extra dimension for cell information has the range 1:1, also consistent with rblock cell numbering.

The various `%seam_*` arrays in `vertex_type` are temporary storage locations for the data that is communicated across block boundaries. The `%order` array holds a unique prescription (determined during NIMROD startup) of the summation order for the different contributions at a border vertex, ensuring identical results for the different blocks. The `%ave_factor` and `%ave_factor_pre` arrays hold weights that are used during the iterative solves.

For vertices along a domain boundary, `tang` and `norm` are allocated (1:2) to hold the R and Z components of unit vectors in the boundary tangent and normal directions, respectively, with respect to the poloidal plane. These unit vectors are used for setting Dirichlet boundary conditions. The scalar `rgeom` is set to R for all vertices in the seam. It is used to find vertices located at $R = 0$, where regularity conditions are applied.

The `%segment` array (belonging to `edge_type` and of type `segment_type`) holds similar information; however, segment communication differers from vertex communication three ways.

- First, as mentioned above, only two blocks can share a cell side.

- Second, segments are used to communicate off-diagonal matrix elements that extend along the cell side.

- Third, there may be no `element_side_centered` data (`poly_degree` = 1), or there may be many nodes along a given element side.

The 1D `ptr` array reflects the limited connectivity. Matrix element communication makes use of `intxyp` and `inxyn` internal vertex indices for the previous and next vertices along the seam (`intxys` holds the internal cell indices). Finally, the `tang` and `norm` arrays have n extra dimension corresponding to the different element side nodes if `poly_degree` > 1.

Before describing the routines that handle block border communication, it is worth noting what is written to the dump files for each seam. The subroutine `dump_write_seam` in the `dump` module shows that very little is written. Only the `ptr`, `intxy`, and `excorner` arrays plus descriptor information are saved for the vertices, but `seam0` is also written. None of the segment information is saved. Thus, NIMROD creates much of the vertex and all of the segment information at startup. This ensures self-consistency among the different seam structures and with the grid.

The routines called to perform block border communication are located in the `edge` module. Central among these routines is `edge_network`, which invokes serial or parallel operations to accumulate vertex and segment data. The `edge_load` and `edge_unload` routines are used to transfer block vertex-type data to or from the seam storage. There are three sets of routines for the three different vector types.

A typical example occurs at the end of the `get_rhs` routine in `finite_element_mod`. The code segment starts with

```
DO ibl=1,nbl
   CALL edge_load_carr(rhsdum(ibl),nqty,1_i4,nfour,nside,seam(ibl))
ENDDO
```

which loads vector component indices `1:nqty` (starting location assumed) and Fourier component indices `1:nfour` (starting location set by the third parameter in the `CALL` statement) for the vertex nodes and ‘`nside` cell-side nodes from the `cvector_type rhsdum` to `seam_cin` arrays in seam. The single `ibl` loop includes both block types.

The next step is to perform the communication.

```
CALL edge_network(nqty,nfour,nside,.false.)
```

There are a few subtleties in the passed parameters. Were the operation for a real (necessarily 2D) `vector_type`, nfour must be 0 (‘`0_i4` in the statement to match the dummy argument’s kind). Were the operation for a `cvector_2D_type`, as occurs in `iter_cg_comp`, nfour must be 1 (‘`1_i4`). Finally, the fourth argument is a flag to perform the load and

unload steps within edge_network. If true, it communicates the crhs and rhs vector_types in the computation_pointers module, which is something of an archaic remnant of the pre-data-partitioning days.

The final step is to unload the border-summed data.

```
DO ibl=1,nbl
   CALL edge_unload_carr(rhsdum(ibl),nqty,1_i4,nfour,nside,seam(ibl))
ENDDO
```

The pardata module also has structures for the seams that are used for parallel communication only. They are accessed indirectly through edge_network and do not appear in the finite element or linear algebra coding.

C Finite Element Computations

Creating a linear system of equations for the advance of a solution field is a nontrivial task with 2D finite elements. The FORTRAN modules listed on the left side of the implementation map (Sec. A) have been developed to minimize the programming burden associated with adding new terms and equations. In fact, normal physics capability development occurs at only two levels. Near the bottom of the map, the integrand modules hold coding that determines what appears in the volume integrals like those on pg11 for Faraday's law. In many instances, adding terms to existing equations requires modifications at this level only.

The other level subject to frequent development is the management level at the top of the map. The management routines control which integrand routines are called for setting-up an advance. They also need to call the iterative solver that is appropriate for a particular linear system. Adding a new equation to the physics model usually requires adding a new management routine as well as new integrand routines. This section of the tutorial is intended to provide enough information to guide such development.

C.1 Management Routines

The adv_* routines in file nimrod.f manage the advance of a physical field. They are called once per advance in the case of linear computations without flow, or they are called twice to predict then correct the effects of advection. A character pass-flag informs the management routine which step to take.

The sequence of operations performed by a management routine is outlined in Sec. D.2b. This section describes more of the details needed to create a management routine.

To use the most efficient linear solver for each equation, there are now three different types of management routines. The simplest matrices are diagonal in Fourier component and have decoupling between various real and imaginary vector components. These advances require

two matrix solves (using the same matrix) for each Fourier component. The `adv_v_clap` is an example; the matrix is the sum of a mass matrix and the discretized Laplacian. The second type is diagonal in Fouries component, but all real and imaginary vector components must be solved simultaneously, usually due to anisotropy. The full semi-implicit operator has such anisotropy, hence `adv_v_aniso` is this type of management routine. The third type of management routine deals with coupling among Fourier components, i.e. 3D matrices. Full use of an evolving number density field requires a 3D matrix solve for velocity, and this advance is managed by `adv_v_3d`.

All these management routine types are similar with respect to creating a set of 2D matrices (used only for preconditioning in the 3D matrix-systems) and with respect to finding the rhs vector. The call to `matrix_create` passes an array of `global_matrix_type` and an array of `matrix_factor_type`, so that operators for each Fourier components can be saved. Subroutine names for the integrand and essential boundary conditions are the next arguments, followed by a b.c. flag and the '`pass`' character variable. If elimination of cell-interior data is appropriate (see Sec. B.4), matrix elements will be modified within `matrix_create`, and interior storage [`rbl_mat(ibl)%mat(4,4)%arr`] is then used to hold the inverse of the interior submatrix (A_{22}^{-1}).

The `get_rhs` calls are similar, except the parameter list is (integrand routine name, `cvector_type` array, essential b.c. routine name, b.c. flags (2), logical switch for using a surface integral, surface integrand name, `global_matrix_type`). The last argument is optional and its presence indicates that interior elimination is used. [Use `rmat_elim=` or `cmat_elim=` to signify real or complex matrix types.

Once the matrix and rhs are formed, there are vector and matrix operations to find the product of the matrix and the old solution field, and the result is added to the rhs vector before calling the iterative solve. This step allows us to write integrand routines for the change of a solution field rather than its new value, but then solve for the new field so that the relative tolerance of the iterative solver is not applied to the change, since $|\Delta x| \ll |x|$ is often true.

Summarizing:

- Integrands are written for $\mathbf{A}$ and b with $\mathbf{A}\Delta x = b$ to avoid coding semi-implicit terms twice.
- Before the iterative solve, find $b - \mathbf{A}x^{\text{old}}$.
- The iterative method solves $\mathbf{A}x^{\text{new}} = (b - x^{\text{old}})$.

The 3D management routines are similar to the 2D management routines. They differ in the following way:

- Cell-interior elimination is not used.
- Loops over Fourier components are within the iterative solve.

- The iterative solve used a 'matrix-free' approach, where the full 3D matrix is never formed. Instead, an rhs integrand name is passed.

See Sec. D for more information on the iterative solves.

After an iterative solve is complete, there are some vector operations, including the completion of regularity conditions and an enforcement of Dirichlet boundary conditions. The latter prevents an accumulation of error from the iterative solver, but it is probably superfluous. Data is saved (with time-centering in predictor steps) as basis function coefficients and at quadrature points after interpolating through the `*block_qp_update` calls.

C.2 `Finite_element` Module

Routines in the `finite_element` module also serve managerial functions, but the operations are those repeated for every field advance, allowing for a few options.

The `real_` and `comp_matrix_create` routines first call the separate numerical integration routines for `rblocks` and `tblocks`, passing the same integrand routine name. Next, they call the appropriate `matelim` routine to reduce the number of unknowns when `poly_degree`> 1. Then, connections for a degenerate `rblock` (one with a circular-polar origin vertex) are collected. Finally, `iter_factor` finds some type of approximate factorization to use as a preconditioner for the iterative solve.

The `get_rhs` routine also starts with block-specific numerical integration, and it may perform an optional surface integration. The next step is to modify grid-vertex and cell-side coefficients through cell-interior elimination ($b_1 - A_{12}A_{22}^{-1}b_2$ from p.53). The block-border seaming then completes a scatter process:

The final steps in `get_rhs` apply regularity and essential boundary conditions.

Aside on the programming:

- Although the names of the integrand, surface integrand, and boundary condition routines are passed through `matrix_create` and `get_rhs`, `finite_element` does not use the `integrands`, `surface_ints`, or `boundary` modules. `finite_element` just needs parameter-list information to call any routine out of the three classes. The parameter list information, including assumed-shape array definitions, are provided by the interface blocks. Note that all integrand routines suitable for `comp_matrix_create`, for example, must use the same argument list as that provided by the interface block for '`integrand`' in `comp_matrix_create`.

C.3 Numerical integration

Modules at the next level in the finite element hierarchy perform the routine operations required for numerical integration over volumes and surfaces. The three modules, `rblock`,

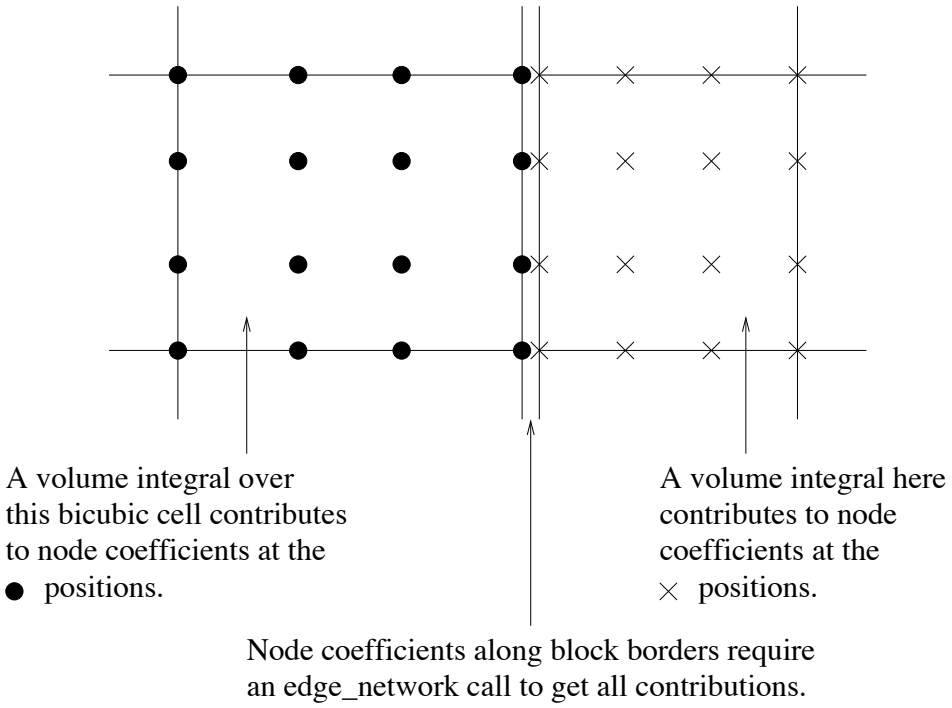


Figure II.7: Block-border seaming

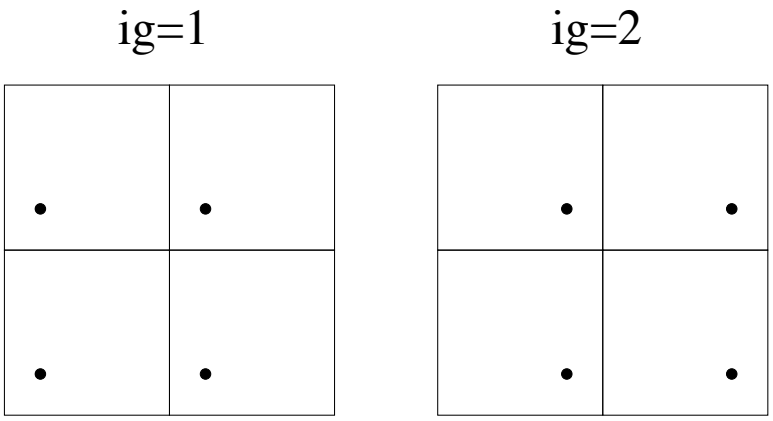


Figure II.8: 4 quad points, 4 elements in a block, dot denotes quadrature position, not node

`tblock`, and `surface`, are separated to provide a partitioning of the different geometric information and integration procedures. This approach would allow us to incorporate additional block types in a straightforward manner should the need arise.

The `rblock_get_comp_rhs` routine serves as an example of the numerical integration performed in NIMROD. It starts by setting the storage arrays in the passed `cvector_type` to $\emptyset$. It then allocates the temporary array, integrand, which is used to collect contributions for test functions that are nonzero in a cell, one quadrature point at a time. Our quadrilateral elements have (`poly_degree`+1)^2 nonzero test functions in every cell (see the Figure II.7 and observe the number of node positions).

The integration is performed as loop over quadrature points, using saved data for the logical offsets from the bottom left corner of each cell, R, and $w_i * \mathcal{J}(\xi_i, \eta_i)$. Thus each iteration of the do-loop finds the contributions for the same respective quadrature point in all elements of a block.

The call to the dummy name for the passed integrand then finds the equation and algorithm-specific information from a quadrature point. A blank tblock is passed along with the rb structure for the block, since the respective tblock integration routine calls the same integrands and one has to be able to use the same argument list. In fact, if there were only one type of block, we would not need the temporary integrand storage at all.

The final step in the integration is the scatter into each basis function coefficient storage. Note that $w_i * \mathcal{J}$ appears here, so in the integral $\int d\xi d\eta \mathcal{J}(\xi, \eta) f(\xi, \eta)$, the integrand routine finds f(ξ, η) only.

The matrix integration routines are somewhat more complicated, because contributions have to be sorted for each basis type and offset pair, as described in B.4 [NIMROD uses two different labeling schemes for the basis function nodes. The `vector_type` and `global_matrix_type` separate basis types (grid vertex, horizontal side, vertical side, cell interior) for efficiency in

linear algebra. In contrast, the integrand routines use a single list for nonzero basis functions in each element. The latter is simpler and more tractable for the block-independent integrand coding, but each node may have more than one label in this scheme. The matrix integration routines have to do the work of translating between the two labeling schemes.]

C.4 Integrands

Before delving further into the coding, let's review what is typically needed for an integrand computation. The Faraday's law example in Section A.1 suggests the following:

1. Finite element basis functions and their derivatives with respect to `R` and `Z`.
2. Wavenumbers for the toridal direction
3. Values of solution fields and their derivatives.
4. Vector operations.
5. Pseudospectral computations.
6. Input parameters and global simulation data.

Some of this information is already stored at the quadrature point locations. The basis function information depends on the grid and the selected solution space. Neither vary during a simulation, so the basis function information is evaluated and saved in `block-structure` array as described in B.1. Solution field data is interpolated to the quadrature pions and stored at the end of the equation advances. It is available through the `qp_type` storage, also described in B.1. Therefore, `generic_alpha_eval` and `generic_ptr_set` routines merely set pointers to the appropriate storage locations and do not copy the data to new locations. **It is important to remember** that integrand computations must not change any values in the pointer arrays. Doing so will lead to unexpected and confusing consequences in other integrand routines.

The input and global parameteres are satisfied by the F90 `USE` statement of modules given those names. The array `keff(1:nmodes)`, is provided to find the value of the wavenumber associated with each Fourier component. The factor `keff(1:nmodes)/bigr` is the toroidal wavenumber for the data at Fourier index `imode`. There are two things to note:

1. In the toroidal geometry, `keff`, holds the n-value, and `bigr(:,:)` holds `R`. In linear geometry, `keff` holds $k_n = \dfrac{2\pi n}{per\_length}$ and `bigr`=1.
2. Do not assume that a particular n-value is associated with some value of `imode`. Linear computations often have `nmodes`=1 and `keff(1)` equal to the input parameter `lin_nmax`. Domain decomposition also affects the values and range of `keff`. This is why there is a `keff` array.

The k2ef array holds the square of each value of keff.

Continuing with the Faraday's law example, the first term on the lhs of the second equation on pg 11 is associated with $\frac{\partial B}{\partial t}$. The spatial discretization gives the integrand $\alpha_i \alpha_j$ (recall that $\mathcal{J}$ appears at the next higher level in the hierarchy). This 'mass' matrix is so common that it has its own integrand routine, get_mass. [The matrix elements are stored in the mass_mat data structure. Operators requiring a mass matrix can have matrix_create add it to another operator.]

The int array passed to get_mass has 6 dimensions, like int arrays for other matrix computations. The first two dimensions are direction-vector-component or 'quantity' indices. The first is the column quantity (jq), and the second is the row quantity (iq). The next two dimensions have range (1:mx,1:my) for the cell indices in a grid-block. The last two indices are basis function indices (jv,iv) for the j' column basis index and the j row index. For get_mass, the orthogonality of direction vectors implies that $\hat{e}_l$ does not appear, so iq=jq=1:

$$\begin{array}{cl} \alpha_{j'}(\xi,\eta)\hat{e}_{l'}\exp(-in'\phi) & \text{test function dotted with} \\ \alpha_j(\xi,\eta)\hat{e}_l\exp(in\phi)\frac{\partial B_{jln}}{\partial t} & \text{gives} \\ \alpha_{j'}\alpha_j & \text{for all } l' \text{ and } n' \end{array}$$

All of the stored matrices are 2D, i.e. diagonal in n, so separate indices for row and column n indices are not used.

The get_mass subroutines then consists of a dimension check to determine the number of nonzero finite elemnt basis function in each cell, a pointer set for α, and a double loop over basis indices. Note that our operators have full storage, regardless of symmetry. This helps make our matrix-vector products faster and it implies that the same routines are suitable for forming nonsymmetric matrices.

The next term in the lhs (from pgB.2) arises from implicit diffusion. The adv_b_iso management routine passes the curl_de_iso integrand name, so lets examine that routine. It is more complicated than get_mass, but the general form is similar. The routine is used for creating the resistive diffusion operator and the semi-implicit operator for Hall terms, so there is coding for two different coefficients. The character variable integrand_flag, that is set in the advance subroutine of nimrod.f and saved in the global module, determines which coefficient to use. Note that calls to generic_ptr_set pass both rblock andtblock structures for a field. One of them is always a blank structure, but this keeps block specific coding out of integrands. Instead, the generic_evals module selects the appropriate data storage.

After the impedence coefficient is found, the int array is filled. The basis loops appear in a conditional statement to use or skip the $\nabla\nabla\cdot$ term for the divergence cleaning. We will skip them in this discussion. Thus we need $\nabla\times(\alpha_{j'}\hat{e}_{l'}\exp(-in'\phi))\cdot\nabla\times(\alpha_j\hat{e}_l\exp(in\phi))$ only, as on pg 11. To understand what is in the basis function loops, we need to evaluate $\nabla\times(\alpha_j\hat{e}_l\exp(in\phi))$. Though $\hat{e}_l$ is $\hat{e}_R$, $\hat{e}_Z$ or $\hat{e}_\phi$ only, it is straightforward to use a general

direction vector and then restrict attention to the three basis vectors of our coordinate system.

$$\begin{aligned}
\nabla\times(\alpha_j\hat{e}_l\exp(in\phi)) &= \nabla(\alpha_j\exp(in\phi))\times\hat{e}_l+\alpha_j\exp(in\phi)\nabla\times(\hat{e}_l)\\
\nabla(\alpha_j\exp(in\phi)) &= \left(\frac{\partial\alpha}{\partial R}\hat{R}+\frac{\partial\alpha}{\partial Z}\hat{Z}+\frac{in\alpha}{R}\alpha\hat{\phi}\right)\exp(in\phi)\\
\nabla(\alpha_j\exp(in\phi))\times\hat{e}_l &= \left[\left(\frac{\partial\alpha}{\partial Z}(\hat{e}_l)_\phi-\frac{in\alpha}{R}(\hat{e}_l)_Z\right)\hat{R}+\left(\frac{in\alpha}{R}(\hat{e}_l)_R-\frac{\partial\alpha}{\partial R}(\hat{e}_l)_\phi\right)\hat{Z}\right.\\
&\quad\left.+\left(\frac{\partial\alpha}{\partial R}(\hat{e}_l)_Z-\frac{\partial\alpha}{\partial Z}(\hat{e}_l)_R\right)\hat{\phi}\right]\exp(in\phi)
\end{aligned}$$

$$\nabla\times(\hat{e}_l)=0 \quad \text{for} \quad l=R,Z$$
$$\nabla\times(\hat{e}_l)=-\frac{1}{R}\hat{Z} \quad \text{for} \quad l=\phi\text{and toroidal geometry}$$

Thus, the scalar product $\nabla\times(\alpha_{j'}\hat{e}_{l'}\exp(-in'\phi))\cdot\nabla\times(\alpha_j\hat{e}_l\exp(in\phi))$ is

$$\begin{aligned}
&\left(\frac{\partial\alpha_{j'}}{\partial Z}(\hat{e}_{l'})_\phi+\frac{in'\alpha_{j'}}{R}(\hat{e}_{l'})_Z\right)\left(\frac{\partial\alpha_j}{\partial Z}(\hat{e}_l)_\phi-\frac{in\alpha_j}{R}(\hat{e}_l)_Z\right)\\
&+\left(-\frac{in\alpha_{j'}}{R}(\hat{e}_{l'})_R-\left(\frac{\alpha_{j'}}{R}+\frac{\partial\alpha_{j'}}{\partial R}\right)(\hat{e}_{l'})_\phi\right)\left(\frac{in\alpha_j}{R}(\hat{e}_l)_R-\left(\frac{\alpha_j}{R}+\frac{\partial\alpha}{\partial R}\right)(\hat{e}_l)_\phi\right)\\
&+\left(\frac{\partial\alpha_{j'}}{\partial R}(\hat{e}_{l'})_Z-\frac{\partial\alpha_{j'}}{\partial Z}(\hat{e}_{l'})_R\right)\left(\frac{\partial\alpha_j}{\partial R}(\hat{e}_l)_Z-\frac{\partial\alpha_j}{\partial Z}(\hat{e}_l)_R\right)
\end{aligned}$$

Finding the appropriate scalar product for a given `(jq,iq)` pair is then a matter of substituting $\hat{R},\hat{Z},\hat{\phi}$ for $\hat{e}_{l'}$ for $l'\Rightarrow iq=1,2,3$, respectively. This simplifies the scalar product greatly for each case. Note how the symmetry of the operator is preserved by construction.

For $(jq,iq=(1,1),\ (\hat{e}_{l'})_R=(\hat{e}_l)_R=1$ and $(\hat{e}_{l'})_Z=(\hat{e}_{l'})_\phi=(\hat{e}_l)_Z=(\hat{e}_l)_\phi=0$ so

$$\texttt{int(1,1,:,:,jv,iv)}=\left(\frac{n^2}{R}\alpha_{jv}\alpha iv+\frac{\partial\alpha_{jv}}{\partial Z}\frac{\alpha_{iv}}{\partial Z}\right)\times\texttt{ziso}$$

where `ziso` is an effective impedance.

Like other isotropic operators the resulting matrix elements are either purely real or purely imaginary, and the only imaginary elements are those coupling poloidal and toroidal vector components. Thus, only real poloidal coefficients are coupled to imaginary toroidal coefficients and vice versa. Furthermore, the coupling between $\texttt{Re}(B_R,B_Z)$ and $\texttt{Im}(B_\phi)$ is the same as that between $\texttt{Im}(B_R,B_Z)$ and $\texttt{Re}(B_\phi)$, making the operator phase independent. This lets us solve for these separate groups of components as two real matrix equations, instead of one complex matrix equation, saving CPU time. An example of the poloidal-toroidal coupling for real coefficients is

$$\texttt{int(1,3,:,:,jv,iv)}=\frac{n\alpha_{jv}}{R}\left(\frac{\alpha iv}{R}+\frac{\alpha_{iv}}{\partial R}\right)\times\texttt{ziso}$$

which appears through a transpose of the $(3,1)$ element. [compare with $-\dfrac{in\alpha_{jv}}{R}\left(\dfrac{\alpha iv}{R}+\dfrac{\alpha_{iv}}{\partial R}\right)$ from the scalar product on pg 41]

Notes on matrix integrands:

- The Fourier component index, `jmode`, is taken from the `global` module. It is the do-loop index set in `matrix_create` and must not be changed in integrand routines.
- Other vector-differential operations acting on $\alpha_j\hat{e}_l\exp(in\phi)$ are needed elsewhere. The computations are derived in the same manner as $\nabla\times(\alpha_j\hat{e}_l\exp(in\phi))$ given above.
- In some case (semi-implicit operators, for example) the $n=0$ component of a field is needed regardless of the n-value assigned to a processor. Separate real storage is created and saved on all processors (see Sects F& C).

The `rhs` integrands differ from matrix integrands in a few ways:

1. The vector aspect (in a linear algebra sense) of the result implies one set of basis function indices in the `int` array.
2. Coefficients for all Fourier components are created in one integrand call.
3. The `int` array has 5 dimensions, `int(iq,:,:,iv,im)`, where `im` is the Fourier index (`imode`).
4. There are FFTs and pseudospectral operations to generate nonlinear terms.

Returning to Faraday's law as an example, the `brhs_mhd` routine finds $\nabla\times(\alpha_{j'}\hat{e}_{l'}\exp(-in'\phi))\cdot\mathbf{E}(R,Z,\phi)$, where $\mathbf{E}$ may contain ideal mhd, resistive, and neoclassical contributions. As with the matrix integrand, $\mathbf{E}$ is only needed at the quadrature points.

From the first executable statement, there are a number of pointers set to make various fields available. The pointers for $\mathbf{B}$ are set to the storage for data from the end of the last time-split, and the `math_tran` routine, `math_curl` is used to find the corresponding current density. Then, $\nabla\cdot\mathbf{B}$ is found for error diffusion terms. After that, the `be`,`ber`, and `bez` arrays are reset to the predicted $\mathbf{B}$ for the ideal electric field during corrector steps (indicated by the `integrand_flag`. More pointers are then set for neoclassical calculations.

The first nonlinear computation appears after the `neoclassical_init` select block. The $\mathbf{V}$ and $\mathbf{B}$ data is transformed to functions of ϕ, where the cross product, $\mathbf{V}\times\mathbf{B}$ is determined. The resulting nonlinear ideal $\mathbf{E}$ is then transformed to Fourier components (see Sec B.1). Note that `fft_nim` concatenates the poloidal position indices into one index, so that `real_be` has dimensions `(1:3,1:mpseudo,1:nphi)` for example. The `mpseudo` parameter can be less

than the number of cells in a block due to domain decomposition (see Sec C), so one should not relate the poloidal index with the 2D poloidal indices used with Fourier components.

A loop over the Fourier index follows the pseudospectral computations. The linear ideal $\mathbf{E}$ terms are created using the data for the specified steady solution, completing - $(\mathbf{V}_s \times \mathbf{B} + (\mathbf{V} \times \mathbf{B}_s + (\mathbf{V} \times \mathbf{B})$ (see Sec A.1). Then, the resistive and neoclassical terms are computed.

Near the end of the routine, we encounter the loop over test-function basis function indices (j', l'). The terms are similar to those in `curl_de_iso`, except $\nabla \times (\alpha_j \hat{e}_l \exp(-in\phi))$ is replaced by the local $\mathbf{E}$, and there is a sign change for $-\nabla \times (\alpha_{j'} \hat{e}_{l'} \exp(-in'\phi)) \cdot \mathbf{E}$.

The integrand routines used during iterative solves of 3D matrices are very similar to `rhs` integrand routines. They are used to find the dot product of a 3D matrix and a vector without forming matrix elements themselves (see Sec D.2). The only noteworthy difference with `rhs` integrands is that the operand vector is used only once. It is therefore interpolated to the quadrature locations in the integrand routine with a `generic_all_eval` call, and the interpolate is left in local arrays. In `p_aniso_dot`, for example, `pres`,`presr`, and `presz` are local arrays, not pointers.

C.5 Surface Computations

The `surface` and `surface_int` modules function in an analogous manner to the volume integration and `integrand` modules, respectively. One important difference is that not all border segments of a block require surface computation (when there is one). Only those on the domain boundary have contributions, and the seam data is not carried below the `finite_element` level. Thus, the surface integration is called one cell-side at a time from `get_rhs`.

At present, 1D basis function information for the cell side is generated on the fly in the surface integration routines. It is passed to a surface integrand. Further development will likely be required if more complicated operations are needed in the computations.

C.6 Dirichlet boundary conditions

The `boundary` module contains routines used for setting Dirichlet (or 'essential') boundary conditions. They work by changing the appropriate rows of a linear system to

$$(A_{jln\ jln})(x_{jln}) = 0, \tag{II.4}$$

where j is a basis function node along the boundary, and l and n are the direction-vector and Fourier component indices. In some cases, a linear combination is set to zero. For Dirichlet conditions on the normal component, for example, the rhs of $Ax = b$ is modified to

$$(\underline{\mathbf{I}} - \hat{n}_j \hat{n}_j) \cdot b \equiv \tilde{b} \tag{II.5}$$

where $\hat{n}_j$ is the unit normal direction at boundary node j. The matrix becomes

$$(\underline{\mathbf{I}} - \hat{n}_j\hat{n}_j) \cdot A \cdot (\underline{\mathbf{I}} - \hat{n}_j\hat{n}_j) + \hat{n}_j\hat{n}_j \tag{II.6}$$

Dotting $\hat{n}_j$ into the modified system gives

$$x_{jn} = \hat{n}_j \cdot x = 0 \tag{II.7}$$

Dotting $(\underline{\mathbf{I}} - \hat{n}_j\hat{n}_j)$ into the modified system gives

$$(\underline{\mathbf{I}} - \hat{n}_j\hat{n}_j) \cdot A \cdot \tilde{x} = \tilde{b} \tag{II.8}$$

The net effect is to apply the boundary condition and to remove x_{jn} from the rest of the linear system.

Notes:

- Changing boundary node equations is computationally more tractable than changing the size of the linear system, as implied by finite element theory.
- Removing coefficients from the rest of the system [through $\cdot(\underline{\mathbf{I}} - \hat{n}\hat{n})$ on the right side of A] may seem unnecessary. However, it preserves the symmetric form of the operator, which is important when using conjugate gradients.

The `dirichlet_rhs` routine in the `boundary` module modifies a `cvector_type` through operations like $(\underline{\mathbf{I}} - \hat{n}\hat{n})\cdot$. The 'component' character input is a flag describing which component(s) should be set to 0. The seam data tang, norm, and intxy(s) for each vertex (segment) are used to minimize the amount of computation.

The `dirichlet_op` and `dirichlet_comp_op` routines perform the related matrix operations described above. Though mathematically simple, the coding is involved, since it has to address the offset storage scheme and multiple basis_types for rblocks.

At present, the kernel always applies the same Dirichlet conditions to all boundary nodes of a domain. Since the `dirichlet_rhs` is called one block at a time, different component input could be provided for different blocks. However, the `dirichlet_op` routines would need modification to be consistent with the rhs. It would also be straightforward to make the 'component' information part of the seam structure, so that it could vary along the boundary independent of the block decomposition.

Since our equations are constructed for the change of a field and not its new value, the existing routines can be used for inhomogeneous Dirichlet conditions, too. If the conditions do not change in time, one only needs to comment out the calls to `dirichlet_rhs` at the end of the respective management routine and in `dirichlet_vals_init`. For time-dependent inhomogeneous conditions, one can adjust the boundary node values from the management routine level, then proceed with homogeneous conditions in the system for the change in the field. Some consideration for time-centering the boundary data may be required. However, the error is no more than $O(\Delta t)$ in any case, provided that the time rate of change does not appear explicitly.

C.7 Regularity conditions

When the input parameter geometry is set to 'tor', and the left side of the grid has points along $R = 0$, the domain is simply connected – cylindrical with axis along Z, not toroidal. With the cylindrical (R, Z, ϕ) coordinate system, our Fourier coefficients must satisfy regularity conditions so that fields and their derivatives approach ϕ-independent values at $R = 0$.

The regularity conditions may be derived by inserting the $x = R\cos\phi$, $y = R\sin\phi$ transformation and derivatives into the 2D Taylor series expansion in (x, y) about the origin. It is straightforward to show that the Fourier ϕ-coefficients of a scalar $S(R, \phi)$ must satisfy

$$S_n(\phi) \sim R^{|n|}\psi(R^2), \tag{II.9}$$

where ψ is a series of nonnegative powers of its argument. For vectors, the Z-direction component satisfies the relation for scalars, but

$$V_{R_n}, V_{\phi_n} \sim R^{|n-1|}\psi(R^2) \text{ for } n \geq 0. \tag{II.10}$$

Furthermore, at $R = 0$, phase information for V_{R_1} and V_{ϕ_1} is redundant. We take ϕ to be 0 at $R = 0$ and enforce

$$V_{\phi_1} = iV_{R_1}. \tag{II.11}$$

The entire functional form of the regularity conditions is not imposed numerically. The conditions are for the limit of $R \to 0$ and would be too restrictive if applied across the finite extent of a computational cell. Instead, we focus on the leading terms of the expansions.

The routines in the `regularity` module impose the leading-order behavior in a manner similar to what is done for Dirichlet boundary conditions. The `regular_vec`, `regular_op`, and `regular_comp_op` routines modify the linear systems to set $S_n = 0$ at $R = 0$ for $n > 0$ and to set V_{R_n} and V_{ϕ_n} to 0 for $n = 0$ and $n \geq 2$. The phase relation for V_{R_1} and V_{ϕ_1} at $R = 0$ is enforced by the following steps:

1. At $R = 0$ nodes only, change variables to $V_+ = (V_r + iV_\phi)/2$ and $V_- = (V_r - iV_\phi)/2$.

2. Set $V_+ = 0$, i.e. set all elements in a V_+ column to 0.

3. To preserve symmetry with nonzero V_- column entries, add $-i$ times the V_ϕ-row to the V_R-row and set all elements of the V_ϕ-row to 0 except for the diagonal.

After the linear system is solved, the `regular_ave` routine is used to set $V_\phi = iV_-$ at the appropriate nodes.

Returning to the power series behavior for $R \to 0$, the leading order behavior for S_0, V_{R_1}, and V_ϕ is the vanishing radial derivative. This is like a Neumann boundary condition. Unfortunately, we cannot rely on a 'natural B.C.' approach because there is no surface with

finite area along an $R = 0$ side of a computational domain. Instead, we add the penalty matrix,

$$\int dV \frac{w}{R^2} \frac{\partial \alpha_{j'}}{\partial R} \frac{\partial \alpha_j}{\partial R}, \tag{II.12}$$

saved in the `dr_penalty` matrix structure, with suitable normalization to matrix rows for S_0, V_{R_1} and V_{ϕ_1}. The weight w is nonzero in elements along $R = 0$ only. Its positive value penalizes changes leading to $\frac{\partial}{\partial R} \neq 0$ in these elements. It is not diffusive, since it is added to linear equations for the changes in physical fields. The `regular_op` and `regular_comp_op` routines add this penalty term before addressing the other regularity conditions.

[Looking through the subroutines, it appears that the penalty has been added for V_{R_1} and V_{ϕ_1} only. We should keep this in mind if a problem arises with S_0 (or V_{Z_0}) near $R = 0$.]

D Matrix Solution

At present, NIMROD relies on its own iterative solvers that are coded in `FORTRAN 90` like the rest of the algorithm. They perform the basic conjugate gradient steps in NIMROD's block decomposed `vector_type` structures, and the `matrix_type` storage arrays have been arranged to optimized matrix-vector multiplication.

D.1 2D matrices

The `iter_cg_f90` and `iter_cg_comp` modules contain the routines needed to solve symmetric-positive-definite and Hermitian-positive-definite systems, respectively. The `iter_cg` module holds interface blocks to give common names and the real and complex routines. The rest of the `iter_cg.f` file has external subroutines that address solver-specific block-border communication operations.

Within the long `iter_cg_f90.f` and `iter_cg_comp.f` files are separate modules for routines for different preconditioning schemes, a module for managing partial factorization(`iter_*_fac`), and a module for managing routines that solve a system(`iter_cg_*`). The basic cg steps are performed by `iter_solve_*`, and may be compared with textbook descriptions of cg, once one understands NIMROD's `vector_type` manipulation.

Although normal seam communication is used during `matrix_vector` multiplication, there are also solver-specific block communication operations. The partial factorization routines for preconditioning need matrix elements to be summed across block borders(including off-diagonal elements connecting border nodes), unlike `matrix_vector` product routines. The `iter_mat_com` routines execute this communication using special functionality in `edge_seg_network` for the off-diagonal elements. After the partial factors are found and stored in the `matrix_factor_type` structure, border elements of the matric storage are restored to their original values by `iter_mat_rest`.

Other seam-related oddities are the averaging factors for border elements. The scalar product of two `vector_type` is computed by `iter_dot`. The routine is simple, but it has to account for redundant storage of coefficients along block borders, hence `ave_factor`. In the preconditioning step, the solve of $\tilde{\mathbf{A}}z = r$ where $\tilde{\mathbf{A}}$ is an approximate A and r is the residual, results from block-based partial-factors(the "direct", "`bl_ilu_*`", and "`bl_diag*`" choices) are averaged at block borders. However, simply multiplying border z-value by `ave_factor` and seaming after preconditioning destroys the symmetry of the operation(and convergence). Instead, we symmetrize the averaging by multiply border elements of r by `ave_factor_pre`($\sim \sqrt{\texttt{ave\_factor}}$), inverting $\tilde{\mathbf{A}}$, then multiply z by `ave_factor_pre` before summing.

Each preconditioner has its own set of routines and factor storage, except for the global and block line-Jacobi algorithms which share 1D matrix routines. The block-direct option uses Lapack library routines. The block-incomplete factorization options are complicated but effective for mid-range condition numbers(arising roughly when $\Delta t_{explicit-limit} \ll \Delta t \ll \tau_A$). Neither of these two schemes has been updated to function with higher-order node data. The block and global line-Jacobi schemes use 1D solves of couplings along logical directions, and the latter has its own parallel communication(see D). There is also diagonal preconditioning, which inverts local direction vector couplings only. This simple scheme is the only one has functions in both rblocks and tblocks. Computations with both block types and solver $\neq$ 'diagonal' will use the specified solver in rblocks and 'diagonal' in tblocks.

NIMROD grids may have periodic rblocks, degenerate points in rblocks, and cell-interior data may or may not be eliminated. These idiosyncracies have little effect on the basic `cg` steps, but they complicate the preconditioning operations. They also make coupling to external library solver packages somewhat challenging.

D.2 3D matrices

The conjugate gradient solver for 3D systems is mathematically similar to the 2D solvers. Computationally, it is quite different. The Fourier representation leads to matrices that are dense in n-index when ϕ-dependencies appear in the lhs of an equation. Storage requirements for such a matrix would have to grow as the number of Fourier components is increased, even with parallel decomposition. Achieving parallel scaling would be very challenging.

To avoid these issues, we use a 'matrix-free' approach, where the matrix-vector products needed for cg iteration are formed directly from rhs-type finite element computation. For example, were the resistivity in our form if Faraday's law on p.? a function of ϕ, it would generate off-diagonal in n contributions:

$$\sum_{jln} B_{jln} \int\int\int dRdZd\phi \frac{\eta(R,Z,\phi)}{\mu_0} \nabla \times (\bar{\alpha}_{j'l'} e^{-in'\phi}) \cdot \nabla \times (\bar{\alpha}_{jl} e^{in\phi}) \qquad \text{(II.13)}$$

which may be nonzero for all n'.

We can switch the summation and integration orders to arrive at

$$\int\int\int dRdZd\phi\nabla\times(\bar{\alpha}_{j'l'}e^{-in'\phi})\cdot\frac{\eta(R,Z,\phi)}{\mu_0}[\sum_{jln}B_{jln}\nabla\times(\bar{\alpha}_{jl}e^{in\phi})] \tag{II.14}$$

Identifying the term in brackets as $\mu_0\mathbf{J}(R,Z,\phi)$, the curl of the interpolated and FFT'ed $\mathbf{B}$, shows how the product can be found as a rhs computation. The B_{jln} data are coefficients of the operand, but when they are associated with finite-element structures, calls to `generic_all_eval` and `math_curl` create $\mu_0 J_n$. Thus, instead of calling an explicit matrix-vector product routine, `iter_3d_solve` calls `get_rhs` in finite element, passing a dot-product integrand name. This integrand routine uses the coefficients of the operand in finite-element interpolations.

The scheme uses 2D matrix structures to generate partial factors for preconditioning that do not address $n \rightarrow n'$ coupling. It also accepts a separate 2D matrix structure to avoid repetition of diagonal in n operations in the dot-integrand; the product is then the sum of the finite element result and a 2D-matrix-vector multiplication. Further, `iter_3d_cg` solves systems for changes in fields directly. The solution field at the beginning of the time split is passed into `iter_3d_cg_solve` to scale norms appropriately for the stop condition.

E Start-up Routines

Development of the physics model often requires new `*block_type` and matrix storage. [The storage modules and structure types are described in Sec. B.] Allocation and initialization of storage are primary tasks of the start-up routines located in file `nimrod_init.f`. Adding new variables and matrices is usually a straightforward task of identifying an existing similar data structure and copying and modifying calls to allocation routines.

The `variable_alloc` routine creates quadrature-point storage with calls to `rblock_qp_alloc` and `tblock_qp_alloc`. There are also `lagr_quad_alloc` and `tri_linear_alloc` calls to create finite-element structures for nonfundamental fields (like $\mathbf{J}$ which is computed from the fundamental $\mathbf{B}$ field) and work structures for predicted fields. Finally, `variable_alloc` creates `vector_type` structures used for diagnostic computations. The `quadrature_save` routine allocates and initializes `quadrature_point` storage for the steady-state (or 'equilibrium') fields, and it initializes quadrature-point storage for dependent fields.

Additional field initialization information:

- The $\hat{e}_3$ component of `be_eq` structures use the covariant component (RB_ϕ) in toroidal geometry, and the $\hat{e}_3$ of `ja_eq` is the contravariant J_ϕ/R. The data is converted to cylindrical components after evaluating at quadrature point locations. The cylindrical components are saved in their respective `qp` structures.

- Initialization routines have conditional statements that determine what storage is created for different physics model options.

- The `pointer_init` routine links the `vector_type` and finite element structures via pointer assignment. New fields will need to be added to this list too.
- The `boundary_vals_init` routine enforces the homogeneous Dirichlet boundary conditions on the initial state. Selected calls can be commented out if inhomogeneous conditions are appropriate.

The `matrix_init` routine allocates matrix and preconditioner storage structures. Operations common to all matrix allocations are coded in the `*_matrix_init_alloc`, `iter_fac_alloc`, and `*_matrix_fac_degen` subroutines. Any new structure allocations can be coded by copying and modifying existing allocations.

F Utilities

The `utilities.f` file has subroutines for operations that are not encompassed by `finite _element` and normal `vector_type` operations. The `new_dt` and `ave_field_check` subroutines are particularly important, though not elegantly coded, subroutines. Teh `new_dt` routine computes the rate of flow through mesh cells to determine if advection should limit the time step. Velocity vectors are averaged over all nodes in an element, and rates of advection are found from $|\mathbf{v}\cdot\mathbf{x}_i/|\mathbf{x}_i|^2|$ in each cell, where $\mathbf{x}_i$ is a vector displacement across the cell in the i-th logical coordinate direction. A similar computation is performed for electron flow when the two-fluid model is used.

The `ave_field_check` routine is an important part of our semi-implicit advance. The 'linear' terms in the operators use the steady-state fields and the $n = 0$ part of the solution. The coefficient for the isotropic part of the operator is based on the maximum difference between total pressures and the `steady_plus_n=0` pressures, over the ϕ-direction. Thus, both the 'linear' and 'nonlinear' parts of the semi-implicit operator change in time. However, computing matrix elements and finding partial factors fro preconditioning are computationally intensive operations. To avoid calling these operations during every time step, we determine how much the fields have changed sicne the last matrix update, and we compute new matrices when the change exceeds a tolerance (`ave_change_limit`). [Matrices are also recomputed when Δt changes.]

The `ave_field_check` subroutine uses the `vector_type` pointers to facilitate the test. It also considers the grid-vertex data only, assuming it would not remain fixed while other coefficients change. If the tolerance is exceeded, flags such as `b0_changed` in the `global` module are set to true, and the storage arrays for the $n = 0$ fields or nonlinear pressures are updated. Parallel communication is required for domain decomposition of the Fourier components (see Sec. C).

Before leaving `utilities.f`, let's take a quick look at `find_bb`. This routine is used to find the toroidally symmetric part of the $\hat{b}\hat{b}$ dyad, which is used for the 2D preconditioner matrix for anisotropic thermal conduction. The computation requires information from

all Fourier components, so it cannot occur in a matrix integrand routine. [Moving the `jmode` loop from `matrix_create` to matrix integrands is neither practical nor preferrable.] Thus, $\hat{b}\hat{b}$ is created and stored at quadrature point locations, consistent with an integrand-type of computation, but separate from the finite-element hierarchy. The routine uses the `mpi_allreduce` command to sum contributions from different Fourier 'layers'.

Routines like `find_bb` may become common if we find it necessary to use more 3D linear systems in our advances.

G Extrap_mod

The `extrap_mod` module holds data and routines for extrapolating solution fields to the new time level, providing the initial guess for an iterative linear system solve. The amount of data saved depends on the `extrap_order` input parameter.

Code development rarely requires modification of these routines, except `extrap_init`. This routine decides how much storage is required, and it creates an index for locating the data for each advance. Therefore, when adding a new equation, increase the dimension fo `extrap_q`, `extrap_nq`, and `extrap_int` and define the new values in `extrap_init`. Again, using existing code for examples is helpful.

H History Module hist_mod

The time-dependent data written to `XDRAW` binary or text files is generated by the routines in `hist_mod`. The output from `probe_hist` is presently just a collection of coefficients from a single grid vertex. Diagnostics requiring surface or volume integrals (toroidal flux and kinetic energy, for example) use `rblock` and `tblock` numerical integration of the integrands coded in the `diagnostic_ints` module.

The numerical integration and integrands differ only slightly from those used in forming systems for advancing the solution. First, the `cell_rhs` and `cell_crhs vector_types` are allocated with `poly_degree=0`. This allocates a single cell-interior basis and no grid or side bases. [The `*block_get_rhs` routines have flexibility for using different basis functions through a simulation. They 'scatter' to any `vector_type` storage provided by the calling routine.] The `diagnostic_ints` integrands differ in that there are no $\bar{\alpha}_{j'l'}e^{-in'\phi}$ test functions. The integrals are over physical densities.

I I/O

Coding new input parameters requires two changes to `input.f` (in the `nimset` directory and an addition to `parallel.f`. The first part of the `input` module defines variables and default values. A description of the parameters should be provided if other users will have

access to the modified code. A new parameter should be defined near related parameters in the file. In addition, the parameter must be added to the appropriate namelist in the `read_namelist` subroutine, so that it can be read from a `nimrod.in` file. The change required in `parallel.f` is just to add the new parameterto the list in `broadcast_input`. This sends the read values from the single processor that reads `nimrod.in` to all others. Be sure to use another `mpi_bcast` or `bcase_str` with the same data type as an example.

Changes to dump files are made infrequently to maintain compatibility across code versions to the greatest extent possible. However, data structures are written and read with generic subroutines, which makes it easy to add fields. The overall layout of a dump file is

1. global information (`dump_write` and `dump_read`)
2. seams (`dump_write_seam`, `dump_read_seam`)
3. `rblocks` (*_write_rblock, *_read_rblock)
4. `tblocks` (*_write_tblock, *_read_tblock)

Within the `rblock` and `tblock` routines are calls to structure-specific subroutines. These calls can be copied and modified to add new fields. [But, don't expect compatibility with other dump files.]

Other dump notes:

- The read routines allocate data structures to appropriate sizes just before a record is read.
- All data is written as 8-byte `real` to improve portability of the file while using `FORTRAN` binary I/O.
- Only one processor reads and writes dump files. Data is transferred to other processors via MPI communication coded in `parallel_io.f`.
- The `NIMROD` and `NIMSET` `dump.f` files are different to avoid parallel data requirements in `NIMSET`. Any changes must be made to both files.

Chapter III

Parallelization

A Parallel communication introduction

For the present and forseeable future, 'high-performance' computing implies parallel computing. This fact was readily apparent at the inception of the NIMROD project, so NIMROD has always been developed as a parallel code. The distributed memory paradigm and `MPI` communication were chosen for portability and efficiency. A well conceived implementation (thanks largely to Steve Plimpton of Sandia) keeps most parallel communication in modular coding, isolated from the physics model coding.

A detailed description of NIMROD's parallel communication is beyond the scope of this tutorial. However, a basic understanding of the domain decomposition is necessary for successful development, and all developers will probably write at least one `mpi_allreduce` call at some time.

While the idea of dividing a simulation into multiple processes is intuitive, the independence of the processes is not. When `mpirun` is used to launch a parallel simulation, it starts multiple processes that execute NIMROD. The first statements in the main program establish communication between the different processes, assign an integer label (`node`) to each process, and tells each process the total number of processes. From this information and that read from the `input` and `dump` files, processes determine what part of the simulation to perform (in `parallel_block_init`). Each process then performs its part of the simulation as if it were performing a serial computation, except when it encounters calls to communicate with other processes. The program itself is responsible for orchestration without an active director.

Before describing the different types of domain decomposition used in NIMROD, lets return to the `find_bb` routine in `utilities.f` for an `MPI` example. Each process is finding $\sum_n \hat{b}_n(R,Z)\hat{b}_n(R,Z)$ for a potentially limited range of n-values and grid blocks. What's needed in every process is the sum over all n-values, i.e. $\overline{\hat{b}\hat{b}}$, for each grid block assigned to a

process. This need is met by calling `mpi_allreduce` within `grid_block do` loops. Here an array of data is sent by each process, and the `MPI` library routine sums the data element by element with data from other processes that have difference Fourier components for the same block. The resulting array is returned to all processes that participate in the communication. Then, each process proceeds independently until its next `MPI` call.

B Grid_block decomposition

The distributed memory aspect of domain decomposition implies that each processor needs direct access to a limited amount of the simulation data. For grid_block decomposition, this is just a matter of having each processor choose a subset of the blocks. Within a `layer` of Fourier components (described in III.C.), the block subsets are disjoint. Decomposition of the data itself is facilitated by the `FORTRAN 90` data structures. Each process has an `rb` and/or `tb block_type` array but the sum of the dimensions is the number of elements in its subset of blocks.

Glancing through `finite_element.f`, one sees `do`-loops starting from 1 and ending at `nrbl`, or starting from `nrbl`+1 and ending at `nbl`. Thus, the `nrbl` and `nbl` values are the number of `rblocks` and `rblocks+tblocks` in a process. In fact, parallel simulations assign new block index labels that are local to each process. The `rb(ibl)%id` and `tb(ibl)%id` descriptors can be used to map a processor's local block index (`ibl`) to the global index (`id`) defined in `nimset`. `parallel_block_init` sets up the inverse mapping `global2local` array that is stored in `pardata`.[`pardata` has comment lines defining the function of each of its variables.] The global index limits, `nrbl_total` and `nbl_total`, are saved in the `global` module.

Block to block communication between different processes is handled within the `edge_network` and `edge_seg_network` subroutines. These routines call `parallel_seam_comm` and related routines instead of the serial `edge_accumulate`. In routines performing finite element or linear algebra computations, cross block communication is invoked with one call, regardless of parallel decomposition.

For those interested in greater details of the parallel block communication, the `parallel_seam_init` routine in `parallel.f` creates new data structures during start-up. These `send` and `recv` structures are analogous to `seam` structures, but the array at the highest level is over processes with which the local process shares block border information.

The `parallel_seam_comm` routine (and its complex version) use `MPI` 'point to point' communication to exchange information. This means that each process issues separate 'send' and 'receive' request to every other process involved in the communication.

The general sequence of steps used in NIMROD for 'point to point' communications is

1. Issue a `mpi_irecv` to every process in the exchange, which indicates readiness to accept data.

2. Issue a `mpi_send` to every process involved which sends data from the local process.

3. Perform some serial work, like seaming among different blocks of the local process to avoid wasting `CPU` time while waiting for data to arrive.

4. Use `mpi_waitany` to verify that expected data from each participating process has arrived. Once verification is complete, the process can continue on to other operations.

Caution: calls to nonblocking communication (`mpi_isend,mpi_irecv,...`) seem to have difficulty with buffers that are part of `F90` structures when the `mpich` implementation is used. The difficulty is overcome by passing the first element, e.g.

```
CALL mpi_irecv(recv(irecv)%data(1),...)
```

instead of the entire array, `%data,...`

As a prelude to the next section, block border communication only involves processes assigned to the same Fourier `layer`. This issue is addressed in the `block2proc` mapping array. An example statement

```
inode=block2proc(ibl_global
```

assigns the `node` label for the process that owns global block number `ibl_global` - for the same `layer` as the local process - to the variable `inode`.

Separating processes by some condition (e.g. same `layer`) is an important tool for NIMROD due to the two distinct types of decomposition. Process groups are established by the `mpi_comm_split` routine. There are two in `parallel_block_init`. One creates a communication tag grouping all processes in a `layer`, `comm_layer`, and the other creates a tag grouping all processes with the same subset of blocks, but different `layer`s, `comm_mode`. These tags are particularly useful for collective communication within the groups of processes. The `find_bb` `mpi_allreduce` was one example. Others appear after scalar products of vectors in 2D iterative solves: different `layer`s solve different systems simultaneously. The sum across blocks involves processes in the same `layer` only, hence calls to `mpi_allreduce` with the `comm_layer` tag. Where all processes are involved, or where 'point to point' communication (which references the default node index) is used, the `mpi_comm_world` tag appears.

C Fourier "layer" decomposition

By this point it's probably evident that Fourier components are also separated into disjoint subsets, called layers. The motivation is that a large part of the NIMROD algorithm involves no interaction among different Fourier components, As envisioned until recently, the matrices would always be 2D, and Fourier component interaction would occur in explicit pseudo-spectral operations only. The new 3D linear systems preserve layer decomposition by avoiding explicit computation of matrix elements that are not diagonal in Fourier index.

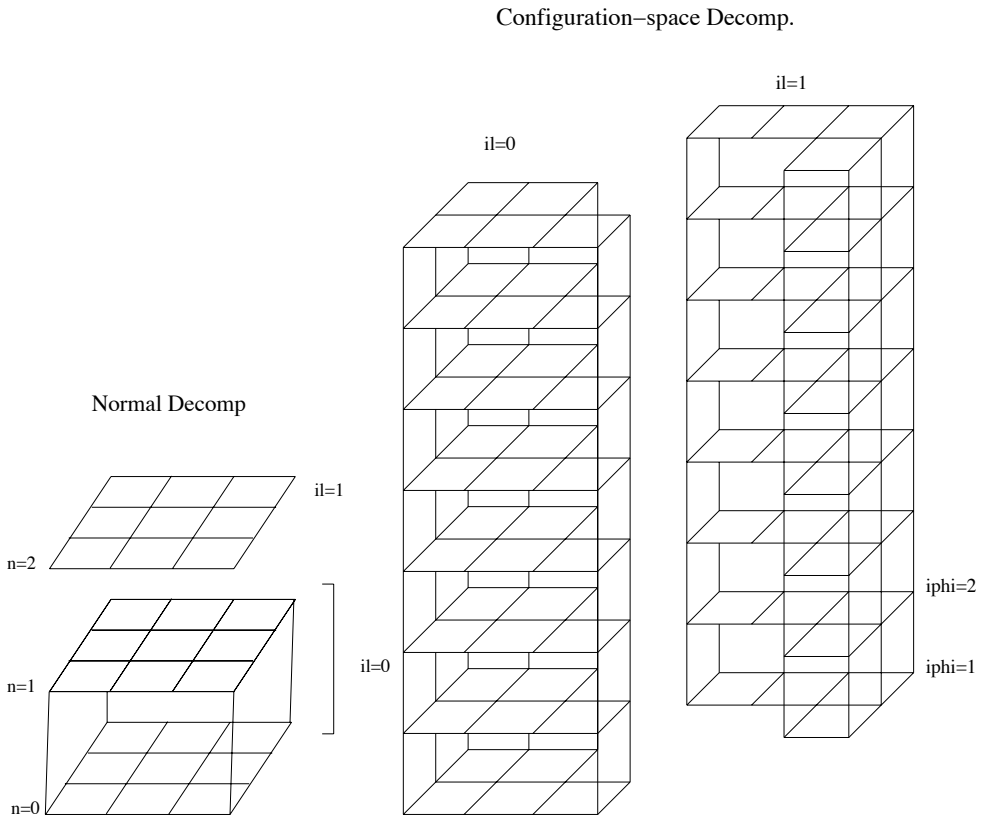


Figure III.1: Normal Decomp: `nmodes=2` on `il=0` and `nmodes=1` on `il=1`, `nf`=`mx`×`my=9`
Config-space Decomp: `nphi`=$2^{lphi} = 8$, `mpseudo=5` on `il=0` and `mpseudo=4` on `il=1`
Parameters are `mx=my=3`, `nlayer=2`, `lphi=3`

Apart from the basic `mpi_allreduce` exchanges, communication among layers is required to multiply functions of toroidal angle (see B.1). Furthermore, the decomposition of the domain is changed before an "inverse" FFT (going from Fourier coefficients to values at toroidal positions) and after a "forward" FFT. The scheme lets us use serial FFT routines, and it maintains a balanced workload among the processes, without redundancy.

For illustrative purposes, consider a poorly balanced choice of `lphi`= 3, $0 \leq n \leq$`nmodes_total`-1 with `nmodes_total`= 3, and `nlayers`= 2. (layers must be divisible by nlayers.) Since pseudo-spectral operations are performed on block at a time, we can consider a single-block problem without loss of generality. (see Figure III.1)

Notes:

- The domain swap uses the collective `mpi_alltoallv` routine.
- Communication is carried out inside `fft_nim`, isolating it from the integrand routines.

- calls to `fft_nim` with $nr = nf$ duplicate the entire block's config-space data on every layer.

D Global-line preconditioning

That ill-conditional matrices arise at large δt implies global propagation of perturbations within a single time advance. Krylov-space iterative methods require very many iterations to reach convergence in these conditions, unless the precondition step provides global "relaxation". The most effective method we have found for cg on very ill-conditioned matrices is a line-Jacobi approach. We actually invert two approximate matrices and average the result. They are defined by eliminating off-diagonal connections in the logical s-direction for one approximation and by eliminating in the n-direction for the other. Each approximate system $\tilde{A}z = r$ is solved by inverting 1D matrices,

To achieve global relaxation, lines are extended across block borders. Factoring and solving these systems uses a domain swap, not unlike that used before pseudospectral computations. However, point-to-point communication is called (from `parallel_line_*` routines) instead of collective communication. (see our 1999 APS poster, "Recent algorithmic ..." on the web site.)

Bibliography

[1] N. A. Krall and Trivelpiece. *Principles of Plasma Physics*. San Francisco Press, 1986.

[2] G. Strang and G. J. Fix. *An analysis of the finite element method*. Wesley-Cambridge Press, 1987.

[3] M. Abramowitz and I. A. Stegun, editors. *Handbook of Mathematical Functions*. Washington, D. C.: National Bureau of Standards: For sale by the Supt. of Docs., U.S. G.P.O., 1964, 1981.

[4] D. D. Schnack, D. C. Barnes, Z. Mikić, Douglas S. Harned, and E. J. Caramana. Semi-implicit magnetohydrodynamic calculations. *J. Comput. Phys.*, 70:330–354, 1987.

[5] K. Lerbinger and J. F. Luciani. A new semi-implicit approach for MHD computation. *J. Comput. Phys.*, 97:444, 1991.

[6] D. S. Harned and Z. Mikić. Accurate semi-implicit treatment of the Hall effect in MHD computations. *J. Comput. Phys.*, 83:1, 1989.

[7] R. Lionello, Z. Mikić, and J. A. Linker. *J. Comput. Phys.*, 152:346, 1999.

[8] B. Marder. A method for incorporating Gauss' law into electromagnetic PIC codes. *J. Comput. Phys.*, 68:48, 1987.

[9] C. R. Sovinec, A. H. Glasser, T. A. Gianakon, D. C. Barnes, R. A. Nebel, S. E. Kruger, D. D. Schnack, S. J. Plimpton, A. Tarditi, M. S. Chu, and the NIMROD Team. Nonlinear magnetohydrodynamics simulation using high-order finite elements. *J. Comput. Phys.*, 195:355–386, 2004.

[10] C. Redwine. *Upgrading to Fortran 90*. Springer, 1995.