

**A Tutorial on NIMROD
Physics Kernel
Code Development**

Carl R. Sovinec
Department of Engineering Physics
University of Wisconsin-Madison, Madison, Wisconsin 53706-1609

August 2001

UW-CPTC 01-3

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by tradename, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

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 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 Sect. I.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 3_0_2 is the advance of temperature instead of pressure in Sect. I.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 orderings are different than what is described in Sect. II. B.

If you find errors or have comments, please send them to me at sovinec@engr.wisc.edu. For the online version, I should be able to make changes, rescan and replace individual pages.

NIMROD Kernel Tutorial Outline

I. Framework

A. Brief reviews

1. Equations
2. Conditions of interest
3. Geometric requirements

B. Spatial discretization

1. Fourier representation
2. Finite element representation
3. Grid blocks

C. Temporal discretization

1. Semi-implicit aspects
2. Predictor-corrector aspects
3. Dissipation
4. O.B constraint
5. Complete time-advance

D. Basic functions of the NIMROD Kernel

II. Fortran implementation

A. Implementation map

B. Data storage and manipulation

1. Solution fields
2. Interpolation
3. Vector types
4. Matrix types
5. Data partitioning
6. Seams

C. Finite element computations

1. Management routines
2. Finite element module
3. Numerical integration

4. Integrands

5. Surface computations

6. Dirichlet boundary conditions

7. Regularity conditions

D. Matrix solution

1. 3D
2. 3D

E. Start-up routines

F. Utilities

G. Extrap-mod

H. History module

I. I/O

III. Parallelization

A. Parallel communication introduction

B. Grid block decomposition

C. Fourier 'layer' decomposition

D. Global-line preconditioning

I. Framework of NIMROD Kernel

Brief reviews — spatial discretization —
temporal discretization — basic functions
of the kernel.

I. A. Brief reviews

I. A. I. 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 Krall & Trivelpiece) of velocity moments of the electron and ion distribution functions, neglecting terms of order m_e/m_i smaller than other terms.

Maxwell w/o displacement current:

Ampere's Law

$$\nabla \times \vec{J} = \sigma \vec{E}$$

Gauss' Law

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

Faraday's Law

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

Fluid moments \rightarrow Single Fluid Form

Quasineutrality is assumed for the time and spatial scales of interest, i.e.

- $Z n_i \approx n_e \rightarrow n$
- $e(z_{ni} - n_e) \vec{E}$ is negligible compared to other forces.
- However $\nabla \cdot \vec{E} \neq 0$.

Continuity

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

Center-of-mass Velocity Evolution

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

• $\underline{\underline{\Pi}}$ often set to $\rho v \nabla \vec{V}$

• kinetic effects, e.g. neoclassical, would appear through $\underline{\underline{\Pi}}$.

Temperature Evolution (3-D-2 & earlier evolve P, P_e)

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

• $\vec{q}_\alpha = -n_\alpha \underline{\underline{\chi}}_\alpha \cdot \nabla T_\alpha$ in collisional limit

• Q_e includes $\alpha n J^2$

• $\alpha = i, e$

$$p_\alpha = n_\alpha k T_\alpha$$

Generalized Ohms (\sim electron velocity moment)

$$\vec{E} = -\vec{V} \times \vec{B} + \sigma \vec{J} + \frac{1}{ne} \vec{J} \times \vec{B} + \frac{m_e}{ne^2} \left[\frac{\partial \vec{J}}{\partial t} + \vec{D} \cdot (\vec{J} \vec{V} + \vec{V} \vec{J}) \right] - \frac{e}{m_e} (D p_e + D \cdot \underline{\underline{\Pi}}_e)$$

input
parameter
 $\sigma_{\text{ohms}} =$

'mhd'
'mhd&hall'
'2f1'

$\text{elecd} > 0$

advect='all' neoclassical

A steady-state part of the solution is separated from the solution fields in NIMROD.

For resistive MHD for example, $\frac{\partial}{\partial t} \rightarrow 0$ implies

$$1. \quad \rho_s \vec{V}_s \cdot \nabla \vec{V}_s = \vec{J}_s \times \vec{B}_s - \nabla P_s + \nabla \cdot \rho_s \nu \nabla \vec{V}_s$$

$$2. \quad \nabla \cdot (n_s \vec{V}_s) = 0$$

$$3. \quad \nabla \times \vec{E}_s = \nabla \times (n_s \vec{J}_s - \vec{V}_s \times \vec{B}_s) = 0$$

$$4. \quad \frac{n_s}{\gamma-1} \vec{V}_s \cdot \nabla T_s = -\rho_s \nabla \cdot \vec{V}_s + \nabla \cdot n_s \underline{\chi} \cdot \nabla T_s + Q_s$$

Notes:

- An equilibrium (Grad-Shafranov) solution is a solution of 1. ($\vec{J}_s \times \vec{B}_s = \nabla P_s$); 2.-4. may or may not be satisfied.
- 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 a vacuum field.

Decomposing each field into steady and evolving parts and cancelling steady terms gives
 (for MHD) [$A \rightarrow A_s + A_e$]

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

$$(\rho_s + \rho) \frac{\partial \vec{V}}{\partial t} + \rho [(\vec{V}_s + \vec{V}) \cdot \nabla (\vec{V}_s + \vec{V})] + \rho_s [\vec{V}_s \cdot \nabla \vec{V} + \vec{V} \cdot \nabla \vec{V}_s + \vec{V} \cdot \nabla \vec{V}]$$

$$= \vec{J}_s \times \vec{B} + \vec{J} \times \vec{B}_s + \vec{J} \times \vec{B} - \nabla P + D \cdot \nu [\rho \nabla \vec{V}_s + \rho_s \nabla \vec{V} + \rho \nabla \vec{V}]$$

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

$$\frac{(n+n_s)}{\sigma-1} \frac{\partial T}{\partial t} + \frac{n}{\sigma-1} [(\vec{V}_s + \vec{V}) \cdot \nabla (T_s + T)] + \frac{n_s}{\sigma-1} [\vec{V}_s \cdot \nabla T + \vec{V} \cdot \nabla T_s + \vec{V} \cdot \nabla T]$$

$$= -\rho_s D \cdot \vec{V} - \rho D \cdot \vec{V}_s - \rho D \cdot \vec{V}$$

$$+ D \cdot [n_s \underline{\chi} \cdot \nabla T + n \underline{\chi} \cdot \nabla T_s + n \underline{\chi} \cdot \nabla T]$$

Notes:

- The code further separates $\underline{\chi}$ (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 decomposition 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.

I. 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.

I.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

I. B. Spatial Discretization

I. 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:

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

Or

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

All physical fields are real functions of space, so we solve for the complex coefficients, A_n ($n \geq 0$) 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: $\vec{E} = -\vec{V} \times \vec{B}$

- Start with $\vec{V}_n, \vec{B}_n, 0 \leq n \leq N$

- FFT gives

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

- Multiply (collocation)

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

- FFT returns Fourier coefficients

$$\hat{E}_n, 0 \leq n \leq N$$

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

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

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

- Collocation is equivalent to spectral convolution if aliasing errors are prevented.

- One can show that setting $\vec{V}_n, \vec{B}_n, \text{etc.} = 0$ for $n > \frac{2N}{3}$ prevents aliasing from quadratic nonlinearities (products of two functions). $[\text{nmodes_total} = \frac{2^{lphi}}{3} + 1]$

- 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 $\vec{f}(\phi)$, 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

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

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

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

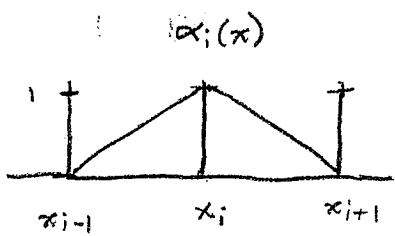
$$2\pi \frac{\partial \vec{B}_{n'}}{\partial t} = - \int d\phi \left(\nabla_{\phi} \times \vec{E}_{n'} + \frac{i n'}{R} \times \vec{E}_{n'} \right), \quad 0 \leq n' \leq N$$

I. 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.

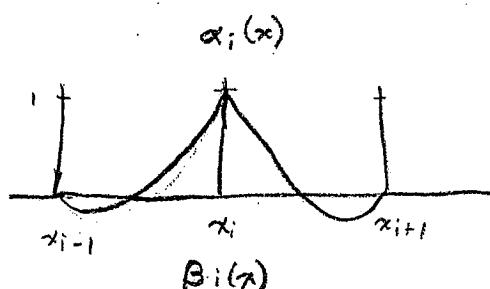
The basis functions of finite elements are low-order polynomials over limited regions of the domain.

Continuous Linear

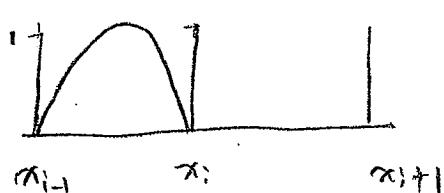


$$\alpha_i(x) = \begin{cases} 0, & x < x_{i-1} \\ \frac{x - x_{i-1}}{x_i - x_{i-1}}, & x_{i-1} \leq x \leq x_i \\ \frac{x_{i+1} - x}{x_{i+1} - x_i}, & x_i \leq x \leq x_{i+1} \\ 0, & x > x_{i+1} \end{cases}$$

Continuous Quadratic



$$\alpha_i(x) = \begin{cases} 0, & x < x_{i-1} \\ \frac{(x - x_{i-1})(x - x_{i-2})}{(x_i - x_{i-1})(x_i - x_{i-2})}, & x_{i-1} \leq x \leq x_i \\ \frac{(x - x_{i+1})(x - x_{i+2})}{(x_{i+1} - x_i)(x_{i+2} - x_i)}, & x_i \leq x \leq x_{i+1} \\ 0, & x > x_{i+1} \end{cases}$$



$$\beta_i(x) = \begin{cases} \frac{(x - x_{i-1})(x - x_i)}{(x_{i+2} - x_{i-1})(x_{i+2} - x_i)}, & x_{i-1} \leq x \leq x_i \\ 0, & otherwise \end{cases}$$

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.
[α_j & β_j in 1D quadratic example]
- 2D basis functions are all possible products of 1D basis functions for R and Z.
[Bilinear, Biquadratic, etc.]

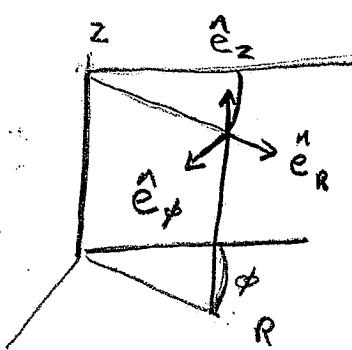
() Nonuniform meshing can be handled by a mapping from logical coordinates to physical coordinates. To preserve convergence properties, i.e. lowest order polynomials of physical coordinates are represented, the mapping should be no higher in polynomial degree than the basis functions in logical variable [Strang & Fix].

() The present version of NIMROD allows the user to choose the polynomial degree of the basis functions ('poly-degree') and of 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 Strang & Fix for analysis of self-adjoint problems.] Error bounds follow from the error bounds of a truncated Taylor series expansion.

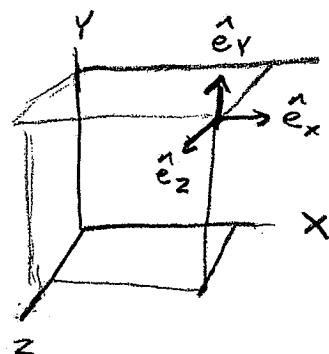
For scalar fields, we now have enough basis functions.

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.



input,
'geom='

'toroidal'



'linear'

For toroidal geometries, we must use the fact that \hat{e}_r, \hat{e}_ϕ 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

$$\vec{V}(R, Z, \phi) \rightarrow \sum_j \sum_l \alpha_j(R, Z) \hat{e}_l \left[V_{j0} + \sum_{n=1}^N V_{jn} e^{in\phi} + \text{c.c.} \right]$$

- * 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}_z , their finite extent means that

$$\iint dR dz D^s(\alpha_i(R,z)) D^s(\alpha_j(R,z)) \neq 0,$$

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 $-\vec{\nabla} \times \vec{B}$ field (after pseudospectral calculations):

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

$$\iiint dR dz d\phi (\bar{\alpha}_{j'e} e^{-in'\phi}) \cdot \sum_{j \neq n} \frac{\partial B_{jen}}{\partial t} \bar{\alpha}_{je} e^{in\phi}$$

$$= - \iiint dR dz d\phi \nabla \times (\bar{\alpha}_{j'e} e^{-in'\phi}) \cdot$$

$$\left[\frac{m}{\mu_0} \sum_{j \neq n} B_{jen} \nabla \times \bar{\alpha}_{je} e^{in\phi} \right.$$

$$\left. + \sum_{j \neq n} E_{jen} \bar{\alpha}_{je} e^{in\phi} \right]$$

$$+ \oint (\bar{\alpha}_{j'e} e^{-in'\phi}) \cdot \sum_{j \neq n} \vec{E}_{jen} \bar{\alpha}_{je} e^{in\phi} \times d\vec{s}$$

Using orthogonality to simplify, and rearranging the Lhs,

$$2\pi \sum_j \frac{\partial B_{jen'}}{\partial t} \iint dR dz \bar{\alpha}_{j'e} \bar{\alpha}_{je}$$

$$+ \sum_{j \neq l} B_{jen'} \iiint dR dz d\phi \frac{m}{\mu_0} \nabla \times (\bar{\alpha}_{j'e} e^{-in'\phi}) \cdot \nabla \times (\bar{\alpha}_{je} e^{in\phi})$$

$$= - \iiint dR dz d\phi \nabla \times (\bar{\alpha}_{j'e} e^{-in'\phi}) \cdot \sum_{j \neq l} E_{jen'} \bar{\alpha}_{je} e^{in\phi}$$

$$+ \oint \bar{\alpha}_{j'e} \cdot \sum_{j \neq l} \vec{E}_{jen'} \bar{\alpha}_{je} \times d\vec{s}$$

for all $\{j', e, n'\}$

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 $\iint dR dz f(\bar{x}_{je}) g(\bar{x}_{je})$ integrals defining a matrix element for row (j', e') and column (j, e) . These elements are diagonal in n .
- $\{b_{jen}\}$ constitutes the solution vector at an advanced time (see Sect. I.C.).
- The rhs is left as a set of integrals over functions of space.
- For self-adjoint operators, such as $\frac{\partial}{\partial t} + \nabla \cdot \frac{1}{\mu_0} \nabla \times$, using the same functions for test and basis functions (Galerkin) leads to the same equations as minimizing the corresponding variational [Strang & Fix], i.e. Rayleigh-Ritz-Galerkin problem.

I, 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 block. 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 extend of the basis functions leads to sparse matrices.
- The mapping for non uniform meshing has not been expressed explicitly. It entails

$$1) \iint dR dz \rightarrow \iint J ds dn ,$$

where s, n are logical coordinates, and J is the Jacobian,

$$2) \frac{\partial \alpha}{\partial R} \rightarrow \frac{\partial \alpha}{\partial s} \frac{\partial s}{\partial R} + \frac{\partial \alpha}{\partial n} \frac{\partial n}{\partial R} \text{ and}$$

similarly for z , where α is a low order polynomial of s and n with uniform meshing in (s, n) .

- The integrals are computed numerically with Gaussian quadrature.

$$\int_0^1 f(s) ds \rightarrow \sum_i w_i f(s_i) \text{ where}$$

$\{w_i, s_i\}, i=1, \dots, n_g$ are prescribed by the quadrature method.

[Abramowitz and Stegun]

I. 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.

I. 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 Schnack, JCP '87 and Lerlinger & Luciani, JCP]

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. [Schnack]. 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. [Lerlinger]

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, \vec{B} is first advanced with the MHD electric field, then with the Hall electric field.

$$B^{MHD} = B^n - \Delta t \nabla E_{MHD}$$

$$B^{n+1} = B^{MHD} + \Delta t \nabla E_{Hall}$$

Though it precludes 2nd order temporal convergence, this splitting reduces errors at large time-step [Harned & Mikic]. A separate semi-implicit operator is then used in the Hall advance of \vec{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 \vec{B} due to E_{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 \rightarrow 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

$\sim 1 + \omega_k^2 At^2$ where ω_k are the frequencies of the MHD waves supported by the spatial discretization.

- We have run tokamak simulations accurately at $\max |\omega_k| At \sim 10^{-4} - 10^{-5}$, and $\min |\omega_k| \approx 0$.
- Leads to very ill-conditioned matrices.
- Anisotropic terms use steady + $n=0$ fields \rightarrow operators are diagonal in n .

I. C. 2. Predictor - corrector aspects

The semi-implicit approach does not ensure numerical stability for advection, even for $\vec{V}_s \cdot \nabla$ 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 [Lionello, JCP].
- NIMROD does not have separate centering parameters for wave and advective terms, so we set centerings to $\frac{t_2}{2} + \delta$ 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.]

I. C. 3. Dissipative terms

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

I. C. 4. $\nabla \cdot \vec{B}$ constraint

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

$$\frac{\partial \vec{B}}{\partial t} = -\nabla \times \vec{E} + K_{\nabla \cdot \vec{B}} \nabla \nabla \cdot \vec{B}$$

Applying $\nabla \cdot$ gives

$$\frac{\partial (\nabla \cdot \vec{B})}{\partial t} = \nabla \cdot K_{\nabla \cdot \vec{B}} \nabla (\nabla \cdot \vec{B})$$

so that the error is diffused if the boundary conditions maintain $\oint d\vec{s} \cdot \vec{B} = 0$.

[See my Sherwood 2000 poster on our web site, <http://nimrodteam.org>]

I. 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.

$$\left\{ \rho^n + \Delta t f_v \nabla \cdot \rho^n \nabla - \frac{C_{sm} \Delta t^2}{4} \left[\vec{B}_0 \times \nabla \times \nabla \times (\vec{B}_0 \times \vec{I}^*) - \vec{J}_0 \times \nabla \times (\vec{B}_0 \times \vec{I}^*) \right] \right. \\ \left. - \frac{C_{sp} \Delta t^2}{4} \left[\nabla \cdot \rho_0 \nabla + \nabla [\nabla \cdot (\rho_0) \cdot \vec{I}^*] \right] - \frac{C_{sn} \Delta t^2}{4} \nabla^2 \right\} (\Delta \vec{V})_{\text{pass}} \\ = - \Delta t \left[\vec{\rho} \vec{V}^* \cdot \nabla \vec{V}^* + \vec{J}^n \times \vec{B}^n - \nabla \rho^n + \nabla \cdot \rho^n \nabla \vec{V}^* + \nabla \cdot \vec{I}^*_{\text{neo}} \right]$$

- For pass = predict, $\vec{V}^* = \vec{V}^n$.

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

- $\vec{V}^{n+1} = \vec{V}^n + (\Delta \vec{V})_{\text{correct}}$

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

- pass = predict $\rightarrow n^* = n^n$

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

- $n^{n+1} = n^n + (\Delta n)_{\text{correct}}$

Note: f's are centering coefficients, C's are semi-implicit coefficients.

$$\left\{ 1 + \nabla \times \left(\frac{me}{ne^2} + \Delta t f_m \frac{m}{M_0} \right) \nabla \times - \Delta t f_{v,B} \nabla \cdot \nabla \cdot \right\} (\Delta \vec{B})_{\text{pass}}$$

$$= \Delta t \nabla \left[\vec{V}^{n+1} \times \vec{B}^* - \frac{\eta}{\mu_0} \nabla \times \vec{B}^n + \frac{1}{ne} \nabla \cdot \vec{B}_{\text{geo}} \right]$$

$$+ \Delta t \chi_{v,B} \nabla \cdot \vec{B}^n$$

- pass = predict $\rightarrow \vec{B}^* = \vec{B}^n$

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

- $\vec{B}^{\text{MHD}} = \vec{B}^n + (\Delta \vec{B})_{\text{correct}}$

$$\left\{ 1 + \nabla \times \left(\frac{me}{ne^2} + C_{SB} (\Delta t)_b \frac{|B_0|}{ne} \right) \nabla \times - \Delta t f_{v,B} \chi_{v,B} \nabla \cdot \right\} (\Delta \vec{B})_{\text{pass}}$$

$$= - (\Delta t)_b \nabla \left[\frac{1}{ne} (\vec{j}^* \times \vec{B}^* - \nabla p_e^n) + \frac{me}{ne^2} \nabla \cdot (\vec{j}^* \vec{V}^{n+1} + \vec{V}^{n+1} \vec{j}^*) \right]$$

$$+ (\Delta t)_b \chi_{v,B} \nabla \cdot \vec{B}^{\text{MHD}}$$

- pass = predict $\rightarrow (\Delta t)_b = \frac{\Delta t}{2}, \vec{B}^* = \vec{B}^{\text{MHD}}$

- pass = correct $\rightarrow (\Delta t)_b = \Delta t, \vec{B}^* = \vec{B}^{\text{MHD}} + (\Delta \vec{B})_{\text{pred}}$

- $\vec{B}^{n+1} = \vec{B}^{\text{MHD}} + (\Delta \vec{B})_{\text{correct}}$

$$\left\{ \frac{n^{n+1}}{\gamma-1} + \Delta t f_\alpha \nabla \cdot n^{\text{Hd}} \chi_\alpha \cdot \nabla \right\} (\Delta T_\alpha)_{\text{pass}}$$

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

- pass = predict $\rightarrow T_\alpha^* = T_\alpha^n$

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

- $T_\alpha^{n+1} = T_\alpha^n + (\Delta T_\alpha)_{\text{correct}}$

- $\alpha = e, i$

Boundary Conditions

Dirichlet conditions are normally enforced for \vec{B}_{normal} and \vec{V} . If thermal conduction is used,

Dirichlet conditions are also imposed on T_{∞} .

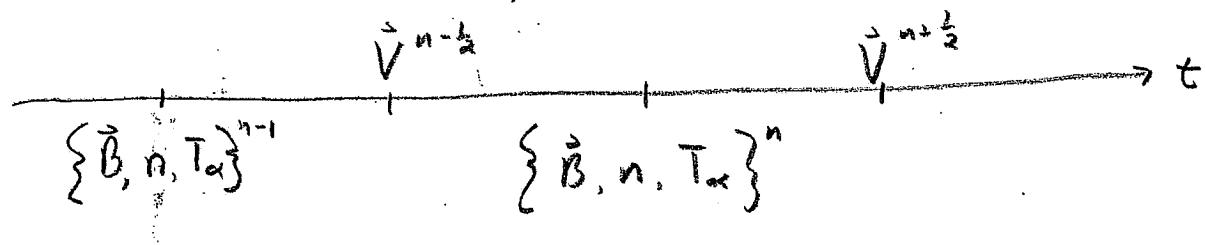
Boundary conditions can be modified for the needs of particular applications, however.

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. [Strang & Fix has an excellent description.] These "natural" conditions preserve the spatial convergence rate without using ghost cells, as in finite difference or finite volume methods.

Notes on time-advance in 3-D-3

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



- The different centering of the Hall time split is required for numerical stability.

[Nobel notes]

- The $\frac{\partial \vec{J}}{\partial t}$ electron inertia term necessarily appears in both time-splits for \vec{B} when it is included in \vec{E} .
- Fields used in semi-implicit operators are update only after changing more than a tolerated level to avoid needlessly frequent matrix element computation.

I. D. Basic functions of the NIMROD kernel

1. Startup

- Read input parameters
- Initialize parallel communication as needed
- Read grid, equilibrium, and initial fields from a 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

2. b) Advance each eq. (continued)

- Complete regularity conditions if needed.
- Update stored solution.
- Deallocate temporary storage.

3. Output

- Write global, energy, and probe diagnostics to binary or text files at desired frequency in time-step.
- Write complete dump files at desired frequency in time-step.

4. Check stop conditions.

- Problems like lack of iterative solver convergence can stop simulations at any point.

II. Fortran Implementation

Implementation map — Data storage & manipulation —
finite element computations — matrix
solution

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.
[Reading Cooper Redwine, "Upgrading to Fortran 90" is recommended.]

II. 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 sub programs for 1). The data and operation needs of sub programs at a level in the hierarchy tend to be similar, so fortran module grouping of sub programs on a level helps orchestrate the hierarchy.

Implementation Map of Primary NIMROD Operations

Finite-element hierarchy

- adiv-x routines
Manage an advection of one field.

FINITE-ELEMENT

- matrix-create
- get-rhs
- Organize f.e. computations.

REGULARITY

BOUNDARY

Essential conditions

ITER-3D-CG-MOD

- Perform 3D matrix solves.

ITER-CG-COM

- Perform complex 2D matrix solve.

TBLOCK

- tblock-matrix
- tblock-get-rhs
- Perform numerical integration in tblocks.

SURFACE

- surface-computations
- Perform numerical Surface integrals.

VECTOR-TYPE-MOD

- Perform ops on vectors of coefficients.

EDGE

- edge-network
- carry out communication across block borders.

MATRIX-MOD

- Complete 2D matrix-vector products.

SURFACE-EVENTS

- Perform surface-specific computations.

NEOCLASS-EVT

- Perform computations for neoclassical-specific equations.

INTEGRANDS

- Perform most equation-specific point-wise computations.

FFT-MOD

- No a routine math operation.

GENERIC-EVNTS

- Interpolate or find storage for data at quadrature points.

- Notes:
1. All-caps indicates a module name.
 2. " " indicates a subroutine name or interface to a module procedure.

II.B. Data storage and manipulation

II.B.1. Solution fields

The data of a NIMROD simulation are the sets of basis function 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.

- rb is a 10 pointer array of defined-type rblock-type. At the start of a simulation, it is dimensioned $1 : \text{nrbl}$, where nrbl is the # 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 the Gaussian quadrature nodes (α , dalphdr , dalphdz), 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 (s, m) .
2. An nb-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 nb , but of type `tblock-type-mod`, dimensioned $\text{nrbl}+1 : \text{nbl}$ so that blocks have numbers distinct from the rblocks.

Before delving into the element-specific structures, let's back up 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 $1 : \text{nbl}$. These structures hold pointer arrays that are used as pointers. There are pointer C as opposed to using them for allocatable arrays in structures.

assignments from these arrays to the lagr-quad-type structure arrays that have allocated storage space. [See Redwine]

In general, the element-specific structures (lagr-quad-types, here) are used in conjunction with the basis functions (as occurs in getting field values at quadrature point locations). The element-independent structures (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:nx)

iy = vertical logical coordinate index (0:ny)

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

$fsh(iv, ib, ix, iy, im)$ \rightarrow coefficients for horizontal side nodes.

iv, ix, iy, im \rightarrow same as for fs ; fsh limits are $(:, :, 1:nx, 0:ny, 1:nfo)$

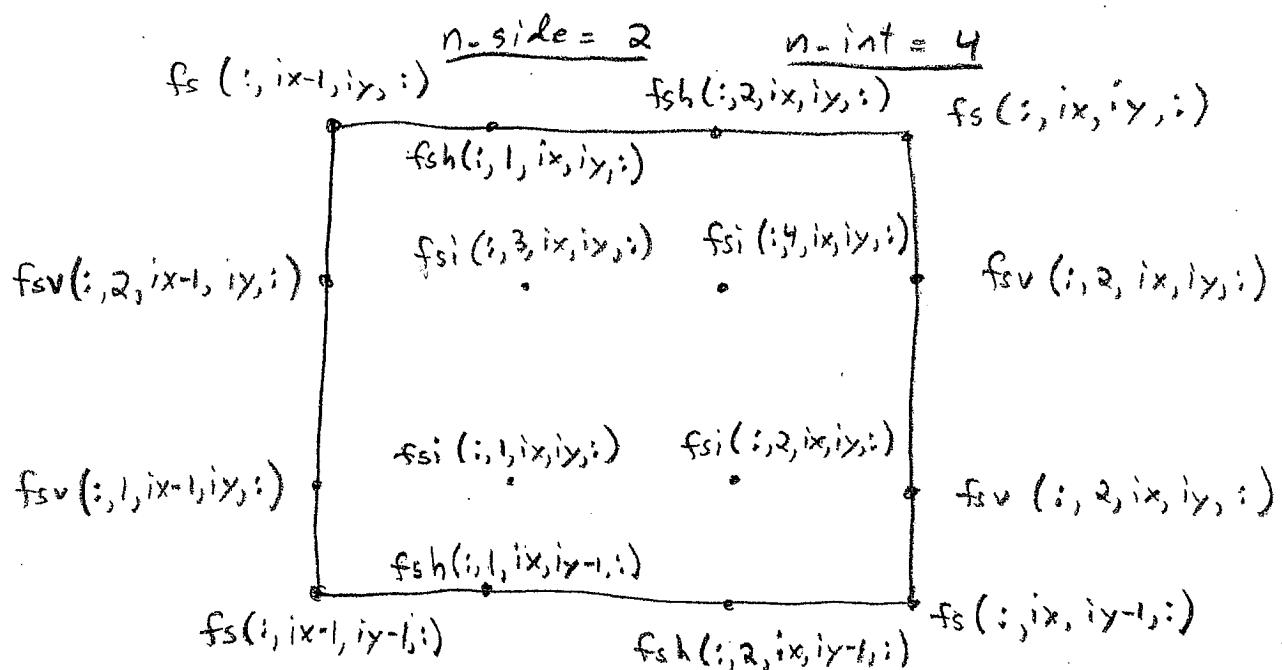
ib = basis index (1:n-side)

$fsv(iv, ib, ix, iy, im)$ \rightarrow coefficients for vertical side node

$fsi(iv, ib, ix, iy, im)$ \rightarrow coefficients for internal nodes,

For fsi ib is (1:n-int).

Bicubic Element Example



II. 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 $\alpha_i|_{(s,m)}$ [and $\frac{\partial}{\partial s} \alpha_i|_{(s,m)}$ and $\frac{\partial}{\partial m} \alpha_i|_{(s,m)}$ if requested] for the (element-independent) basis function values at the requested offset (s',m') . The lagr-quad-bases routine is used for this operation. [It's also used at start up to find the basis function information at quadrature point locations, stored in the rb% alpha, dalpha_s, dalpha_m 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 structset-lag 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.

II. 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 $\text{Re}\{B_R\}$, $\text{Re}\{B_z\}$, and $-\text{Im}\{B_\phi\}$ for the F-comp., and the other solves for $\text{Im}\{B_R\}$, $\text{Im}\{B_z\}$, and $\text{Re}\{B_\phi\}$. After a solve in a predictor step (b -pass = 'b-mhd pre' or 'b-hall pre'), we need to create a linear combination of the old field and the predicted solution. At this point,

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

What we want is $\vec{B}_{\text{inode}}^* = f_b \vec{B}_{\text{inode}}^{\text{pre}} + (1-f_b) \vec{B}_{\text{inode}}^*$

so we have

CALL vector-assign-cvec (vectr(ibl), be(ibl), flag, inode)

↳ Transfers the flag-indicated components such as 'r12m13' for $\text{Re}\{B_1\}, \text{Re}\{B_2\}, -\text{Im}\{B_3\}$ for inode to the 20 vectr.

CALL vector-add (sln(ibl), vectr(ibl), v1 face=center,
v2 face=1-center)

↳ Adds $(1-\text{center}) * \text{vectr}$ to center $* \text{sln}$ and saves result in sln.

CALL cvector-assign-vec (work1(ibl), sln(ibl), flag, inode)

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

Examining vector-type-mod at the end of the vector-type-mod.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-mod
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.

II. 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 gradients 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 $0 \leq iy \leq my$, ig and jg
 correspond to the vector index dimension of
 the vector-type. [simply 1:3 for A_R, A_2, A_4
 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.

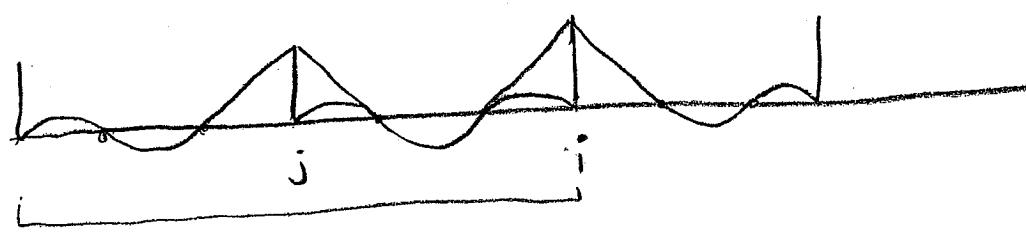
The column coordinates are simply

$$jx = ix + jx\text{off} \quad \text{and}$$

$$jy = iy + jy\text{off}$$

For each dimension, vertex to vertex offsets
 are $-1 \leq j*\text{off} \leq 1$, 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.

Cubics Example along 1D, #1



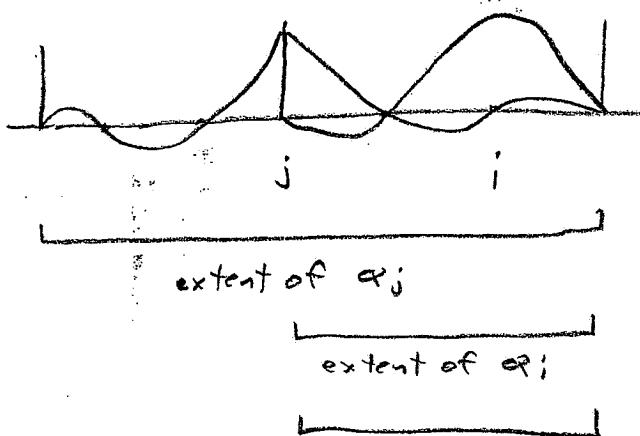
finite extent of α_j

finite extent of α_i

Integration here gives $j\text{off} = -1$ matrix elements.

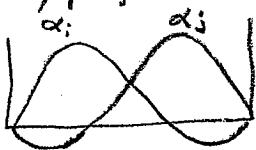
Offsets are more limited for basis functions that have their nonzero node in a cell interior.

Cubic Example along 1D, #2



Integration here gives $j_{\text{off}} = -1$ matrix elements, since cells are numbered from 1, while vertices are numbered from 0. Thus for grid vertex to cell connections $-1 \leq j_{\text{off}} \leq 0$, while for cell to grid vertex connections, $0 \leq j_{\text{off}} \leq 1$. Finally, $j_{\text{off}} = 0$ 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.



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:

1) Bicubic grid vertex to vertical side for P
 $\rightarrow j_q=1, 1 \leq i_g \leq 2$

2) Biquartic cell interior to horizontal side
for $\vec{B} \rightarrow 1 \leq j_q \leq 27, 1 \leq i_g \leq 9$

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

1	= grid-vertex type
2	= horizontal-side type
3	= vertical-side 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 $nbtypes=1 or 4$; starting horizontal and vertical coordinates ($=0 or 1$) for each of the different types; $nbase$ -type ($1:4$) = the number of bases for each type [poly-degree-1 for types 2-3 and $(\text{poly-degree}-1)^2$ for type 4]; and nq -type = quantity-index dimension for each type.

All of this is placed in a structure, rbl-mat, so that we can have an array of rbl-mats with one component per grid block. A similar rbl-mat structure is also at this level, along with more 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 Fourier component index, i.e. 2D.

Matrix-type-mod also contains definitions of structures used to hold data for the preconditioning 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 \vec{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 matvecs 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,

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

where A_{ij} are submatrices; x_i and b_i are parts of vectors, and A_1 and A_{22} are invertible,

$$x_2 = A_{22}^{-1} (b_2 - A_{21}x_1)$$

$$\underbrace{[A_{11} - A_{12}A_{22}^{-1}A_{21}]}_{\text{Scher complement}} x_1 = b_1 - A_{12}A_{22}^{-1}b_2$$

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 submatrices, 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 .

* Additional note regarding matrix storage: the stored coefficients are not summed across block borders. Instead, vectors are summed, so that Ax is found through operations separated by block, followed by a communication step to sum the product vector across block borders.

II. B. 5. Data partitioning

Data partitioning is part of a philosophy on 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 the 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 rolled 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 a 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's necessary to get a result.
- However, in many cases, a little extra effort leads to a lot more usage of development work.

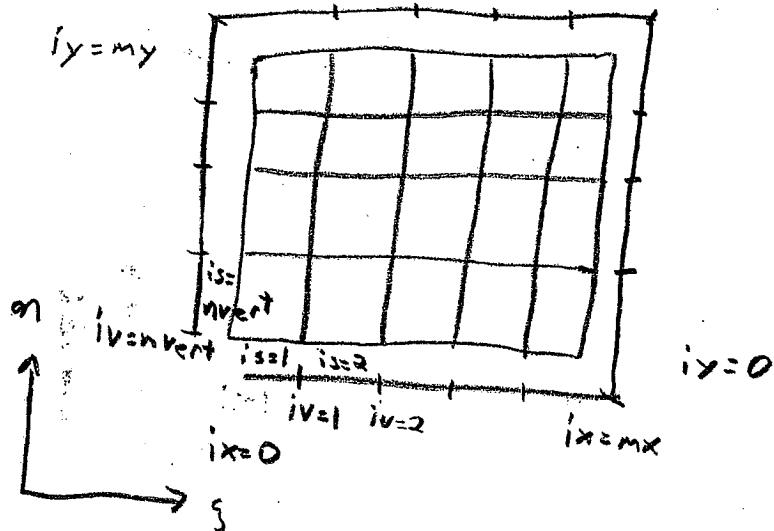
II. B. 6. Seams

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 holds the seam array, a list of blocks that touch the boundary of the domain (exblock-list), and seam \emptyset . The seam \emptyset structure is of edge-type, like the seam array itself. seam \emptyset 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 himset to create seam \emptyset .]

As defined in the edge-type-mod module, an edge-type holds vertex and segment structures. Separation of the border data 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 r \emptyset point 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.

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 number of border segments—but the starting locations are offset. For r blocks, the ordering is unique:



where $iv = \text{vertex index}$

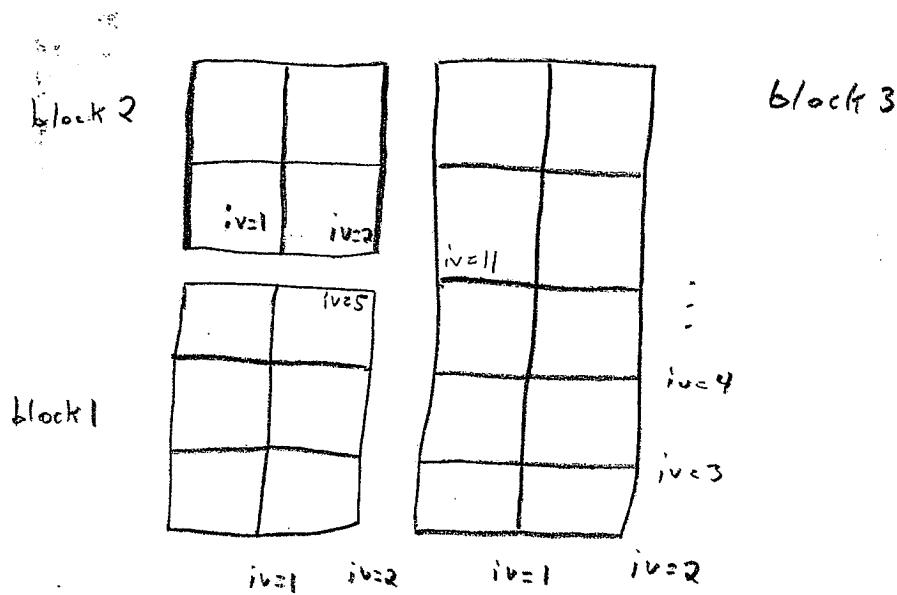
$is = \text{segment index}$

$nvert = \text{number of border vertices or segments}$

Both indices proceed counter clockwise around the block. For t blocks, there is no requirement that the seam start at a particular internal-vertex number, but the seam indexing always progresses counter clockwise around its block (entire domain for $\text{seam} \Phi$).

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 $\text{seam} \Phi$. It gets defined in nimset and is saved in dump files. The ptr2 array is a duplicate of ptr , but references to $\text{seam} \Phi$ 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 number (running from 1 to nbl-total, not 1 to nbl - see information on parallelization) or 2 to select seam vertex number. Here is an example:



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

$$\begin{aligned} \text{seam}(1) \% \text{vertex}(5) \% \text{ptr}(1,1) &= 3 \\ \text{ptr}(2,1) &= 11 \\ \text{ptr}(1,2) &= 2 \\ \text{ptr}(2,2) &= 2 \end{aligned}$$

$$\begin{aligned} \text{seam}(2) \% \text{vertex}(2) \% \text{ptr}(1,1) &= 1 \\ \text{ptr}(2,1) &= 5 \\ \text{ptr}(1,2) &= 3 \\ \text{ptr}(2,2) &= 11 \end{aligned}$$

$$\begin{aligned} \text{seam}(3) \% \text{vertex}(11) \% \text{ptr}(1,1) &= 2 \\ \text{ptr}(2,1) &= 2 \\ \text{ptr}(1,2) &= 1 \\ \text{ptr}(2,2) &= 5 \end{aligned}$$

Notes on ptr

- It does not hold a self-reference, e.g. there are no references like

$$\text{seam}(3) \% \text{vertex}(11) \% \text{ptr}(1,3) = 3 \\ \text{ptr}(2,3) = 11$$

- The order of the connections is arbitrary.

The vertex-type stores interior vertex labels for its block in the intxy array. In rblocks $\text{vertex}(iv) \% \text{intxy}(1:2)$ 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 \emptyset .

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 \emptyset to invert, 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 range $\emptyset:\emptyset$.

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, wrt 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 (of type segment-type) holds similar information; however, segment communication differs from vertex communication in 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 ($\text{poly-degree} = 1$), or there may be many nodes along a given element side. The ID_jptr -array reflects the limited connectivity. Matrix element communication makes use of intxy_p and intxyn — internal vertex indices for the previous and next vertices along the seam. [Intxy_s 's holds the internal cell indices.] Finally, the tang and norm arrays have an extra dimension corresponding to the different element side nodes if $\text{poly-degree} > 1$.

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 `ex-corner` arrays plus descriptor information are saved for the vertices, but seam ϕ is also written. None of the segment information is saved. Thus, `nirrod` 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-med. The code segment starts with

```
DO ibl=1,nbl  
  CALL edge-load-carr (rhsdum(ibl),ngty,1-i4,nfour,  
                      nside,seam(ibl)).  
ENDDO
```

which loads vector component indices 1:ngty (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 'inside' cell-side nodes from the vector-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 (ngty, nfour, nside, .false.)

There are a few subtleties in the passed parameters. Were the operation for a real, (necessarily 20) vector-type, nfour must be 0 ('0-i4' in the statement to match the dummy argument's kind). Were the operation for a vector-20-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 (rhsdun(ibl), ngty, 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.

II. C. Finite element computations

Creating a linear system of equations for the advance of a solution field is a nontrivial task with 20 finite elements. The Fortran modules listed on the left side of the implementation map (sect. II 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 p. 19 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.

II. 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 Sect. I. D. 2.b). 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 Fourier 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 three 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 component 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 II.B.4.), matrix elements will be modified within matrix-create, and interior store,

[rbl.mat(rbl)%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 'imat-elim' or 'cimat-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 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 $\|Ax\| \ll \|x\|$ is often true.

Summarizing: Integrands are written for $A \cdot b$ with $A \Delta x = b$ to avoid coding semi-implicit terms twice.

- Before the iterative solve, find $b - Ax^{old}$.

The iterative method solves $Ax^{new} = (b - Ax^{old})$

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

- Cell-interior elimination is not used.
- Loops over Fourier components are within the iterative solve.
- The iterative solve uses a 'matrix-free' approach, where the full 3D matrix is never formed. Instead, a rhs integrand name is passed.

See Sect. II.D. for more information on the iterative solves.

After an iterative solve is complete, there are some vector operations, including the completion of

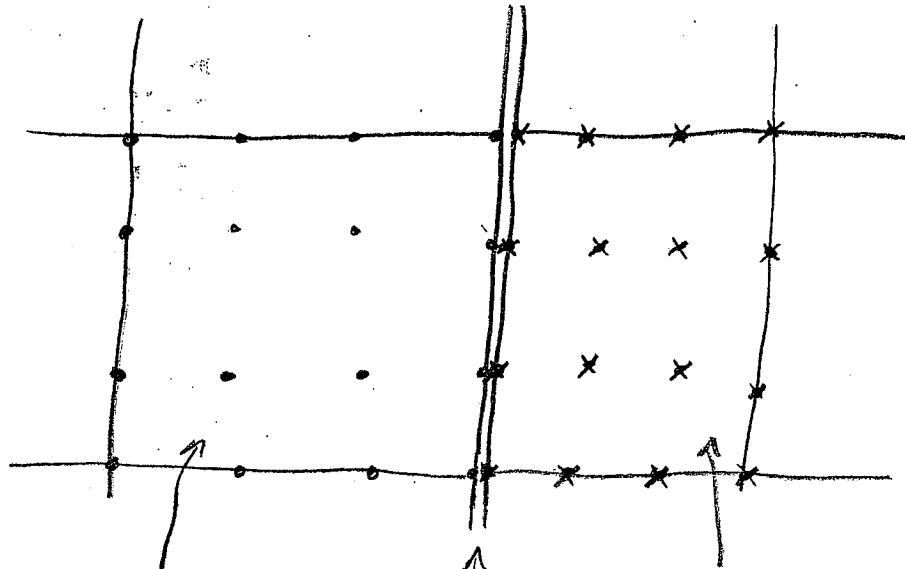
regularity conditions and an enforcement of Dirichlet boundary conditions. The latter prevents an accumulation of error from the iterative solver, but it is probably superfluous. Data is saved (with time-centering in predictor steps) as basis function coefficients and at quadrature points after interpolating through the *block-qp-update calls.

II. 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 pblock (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 a cell-interior elimination ($b_1 = A_{12} A_{22}^{-1} b_2$ from p. 53). The block-border seaming then completes a scatter process:



A volume integral over this bicubic cell contributes to node coefficients at the \circ positions.

A volume integral here contributes to node coefficients at the \times positions,

Node coefficients along block borders require an edge-network call to get all contributions.

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.

II. C. 3. Numerical integration

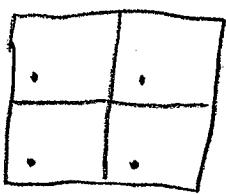
Modules at the next level in the finite element hierarchy perform the routine operations required for numerical integration over volumes and surfaces. The three modules, rblock, tblock, and surface, are separated to provide a partitioning of the different geometric information and integration procedures. This approach would allow us to incorporate additional block types in a straight forward manner should the need arise.

The `rblock-get-comps` 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 all test functions that are nonzero in a cell, one quadrature point at a time. Our quadrilateral elements have $(\text{poly-degree}+1)^2$ nonzero test functions in every cell (see the figure on p. 70 and observe the number of node positions).

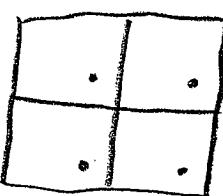
The integration is performed as a loop over quadrature points, using saved data for the logical offsets from the bottom left corner of each cell, R , and $w_i * f(s_i, n_i)$. Thus each iteration of the do-loop finds the contributions for the same respective quadrature point in all elements of a block:

Example - 4 quad points, 4 elements in a block

$ig=1$



$ig=2$



, etc.

"." shows quadrature point position - not node positions.

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 integrand, 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^* appears here, so in the integral $\int dz_m g(z,m) f(z,m)$, the integrand routine finds $f(z,m)$ 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 Sect. II.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.]

II. 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 Sections I.B.1.-2. suggests the following:

1. Finite element basis functions and their derivatives with respect to R and Z .
2. Wavenumbers for the toroidal 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 save in block-structure arrays, as described in II. B. 1. Solution field data is interpolated to the quadrature points and stored at the end of equation advances. It is available through the qp-type storage, also described in II.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's 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 parameters 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(imode)/bigr is the toroidal wavenumber for the data at Fourier index imode. There are two things to note:

1. In toroidal geometry, keff, holds the n-value, and bigr (:,:) holds R. In linear geometry, keff holds $k_n = \frac{2\pi n}{\text{per-length}}$, and bigr=1.

2. Do not assume that a particular n-value is associated with some value of imode. Linear computations often have nnodes=1, and keff(1) = the input parameter lin-nmax. Domain decomposition also affects the values and range of keff. This is why there is a keff array.

The kdef 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 p.19 is associated with $\frac{\partial \mathbf{B}}{\partial t}$. The spatial discretization gives the integrand $\alpha_j \alpha_j$ (recall that 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 (jg), and the second is the row quantity (ig). The next two dimensions have range (1:mx, 1:my) for the cells 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}_2 does not appear, so $i_2 = j_2 = 1$:

$\alpha_j(z, m) \hat{e}_2 e^{-i k_2 z}$ test function dotted with

$\alpha_j(z, m) \hat{e}_2 e^{i k_2 z} \frac{\partial b_{j, l, n}}{\partial t}$ gives

$$\alpha_j'(z, m) \alpha_j(z, m) \quad \text{for all } l' \text{ and } n'.$$

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 subroutine then consists of a dimension check to determine the number of nonzero finite element basis functions 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 p. 19) arises from implicit diffusion. The adv-b-iso management routine passes the curl-de-iso integrand name, so let's 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 and tblock 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 impedance coefficient is found, the int array is filled. The basis loops appear in a conditional statement to use or skip the $\nabla \cdot \mathbf{D} \mathbf{D}$ terms for divergence cleaning. We'll skip them in this discussion. Thus, we need $\nabla \times (\alpha_j \hat{\mathbf{e}}_e e^{i\omega t}) \cdot \nabla \times (\alpha_j \hat{\mathbf{e}}_e e^{i\omega t})$ only, as on p. 19. To understand what is in the basis function loops, we need to evaluate $\nabla \times (\alpha_j \hat{\mathbf{e}}_e e^{i\omega t})$. Though $\hat{\mathbf{e}}_e$ is $\hat{\mathbf{e}}_x, \hat{\mathbf{e}}_y$, or $\hat{\mathbf{e}}_z$ only, it's straightforward to use a general direction vector and then restrict attention to the three basis vectors of our coordinate system.

$$\nabla \times (\alpha; \hat{e}_e e^{in\phi}) = \nabla(\alpha; e^{in\phi}) \times \hat{e}_e + \alpha e^{in\phi} \nabla \times (\hat{e}_e)$$

$$\nabla(\alpha; e^{in\phi}) = \left(\frac{\partial \alpha}{\partial R} \hat{R} + \frac{\partial \alpha}{\partial Z} \hat{Z} + \frac{in\alpha}{R} \alpha \hat{\phi} \right) e^{in\phi}$$

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

$$\nabla \times (\hat{e}_e) = 0 \quad \text{for } l=R, Z$$

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

Thus, the scalar product $\nabla \times (\alpha; \hat{e}_{e'} e^{-iq\phi}) \cdot \nabla \times (\alpha; \hat{e}_e e^{in\phi})$

is

$$\begin{aligned} & \left(\frac{\partial \alpha_{j'}}{\partial Z} (\hat{e}_{e'})_\phi + \frac{in\alpha_{j'}}{R} (\hat{e}_{e'})_z \right) \left(\frac{\partial \alpha_i}{\partial Z} (\hat{e}_e)_\phi - \frac{in\alpha_i}{R} (\hat{e}_e)_z \right) \\ &+ \left(-\frac{in\alpha_{j'}}{R} (\hat{e}_{e'})_R - \left(\frac{\partial \alpha_{j'}}{\partial R} + \frac{\partial \alpha_{j'}}{\partial Z} \right) (\hat{e}_{e'})_\phi \right) \left(\frac{in\alpha_i}{R} (\hat{e}_e)_R - \left(\frac{\partial \alpha_i}{\partial R} + \frac{\partial \alpha_i}{\partial Z} \right) (\hat{e}_e)_\phi \right) \\ &+ \left(\frac{\partial \alpha_{j'}}{\partial R} (\hat{e}_{e'})_z - \frac{\partial \alpha_{j'}}{\partial Z} (\hat{e}_{e'})_R \right) \left(\frac{\partial \alpha_i}{\partial R} (\hat{e}_e)_z - \frac{\partial \alpha_i}{\partial Z} (\hat{e}_e)_R \right) \end{aligned}$$

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

For $(j_\theta, i_\theta) = (1, 1)$, $(\hat{e}_\theta')_R = (\hat{e}_\theta)_R = 1$ and
 $(\hat{e}_\theta')_z = (\hat{e}_\theta')_\phi = (\hat{e}_\theta)_z = (\hat{e}_\theta)_\phi = 0$, so

$$\text{int}(1, 1, :, :, j_\theta, i_\theta) = \left(\frac{n^2}{R} \alpha_{j_\theta} \text{div} + \frac{\partial \alpha_{j_\theta}}{\partial z} \frac{\partial \text{div}}{\partial z} \right) * Z_{\text{iso}}$$

where Z_{iso} 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 $\text{Re}(B_R, B_z)$ and $-\text{Im}(B_\theta)$ is the same as that between $\text{Im}(B_R, B_z)$ and $\text{Re}(B_\theta)$, 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

$$\text{int}(1, 3, :, :, j_\theta, i_\theta) = \frac{n \alpha_{j_\theta}}{R} \left(\frac{\text{div}}{R} + \frac{\partial \text{div}}{\partial R} \right) * Z_{\text{iso}}$$

which appears through a transpose of the (3,1) element.
[Compare with $-\frac{i n \alpha_{j_\theta}}{R} \left(\frac{\text{div}}{R} + \frac{\partial \text{div}}{\partial R} \right)$ from the scalar product on p. 79.]

Notes on matrix integrands:

- The Fourier component index, j_{mode} , 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 $\alpha_j \hat{e}^{e^{\text{int}}}$ are needed elsewhere. The computations are derived in the same manner as $\nabla \times (\alpha_j \hat{e}^{e^{\text{int}}})$ given above.
- In some case (semi-implicit operators, for example) the $n=0$ component of a field is needed regardless of the n -values assigned to a processor. Separate real storage is created and saved on all processors. See Sects. II, F, and III, G.

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, $\text{int}(ig, :, :, iv, im)$, where im is the Fourier index (j_{mode}).
- 4) There are FFTs and pseudospectral operations to generate nonlinear terms.

Returning to Faraday's law as an example, the brhs_mhd routine finds $\nabla \times (\alpha_j \vec{E}_e \cdot e^{-int}) \cdot \vec{E}(r, z, \phi)$, where \vec{E} may contain ideal-mhd, resistive, and neoclassical contributions. As with the matrix integrand, \vec{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 \vec{B} are set to the storage for data from the end of the last time-split, and the math-curl routine, math-curl is used to find the corresponding current density. Then, $\nabla \cdot \vec{B}$ is found for error-diffusion terms. After that, the be, ber, and bez arrays are reset to the predicted \vec{B} for the ideal electric field during corrector steps. (indicated by integrand-flag). More pointers are then set for neoclassical calculations.

The first nonlinear computation appears after the neoclass-init select block. The \vec{V} and \vec{B} data is transformed to functions of ϕ , where the cross product, $\vec{V} \times \vec{B}$ is determined. The resulting nonlinear ideal \vec{E} is then transformed to Fourier components. [See I.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:inphi) for example. The pseudo parameter can be less than the number of cells in a block

due to domain decomposition (see III.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 \vec{E} terms are created using the data for the specified steady solution, completing $-(V_s \times B + V \times B_s + V \times B)$.

[See I.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 $\nabla \times (\alpha_j \vec{e}_x e^{i\theta})$ is replaced by the local \vec{E} , and there is a sign change for $-\nabla \times (\alpha_{j'} \vec{e}_{x'} e^{i\theta'}) \cdot \vec{E}$.

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 Sect. II.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.

II. 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, 1.0 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.

II. C. G. 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

$$(A_{jen\ jen})(x_{jen}) = 0$$

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 $(I - \hat{n}_j \hat{n}_j^T) \cdot b \equiv \tilde{b}$, where \hat{n}_j is the unit normal direction at boundary node j . The matrix becomes

$$(I - \hat{n}_j \hat{n}_j^T) \cdot A \cdot (I - \hat{n}_j \hat{n}_j^T) + \hat{n}_j \hat{n}_j^T .$$

Dotting A_j into the modified system gives

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

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

$$(I - \hat{n}_j \hat{n}_j^T) \cdot A \cdot \tilde{x} = \tilde{b}$$

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 $\cdot(I - \hat{A}_j \hat{n}_j)$ 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 vector-type through operations like $(I - \hat{n}_j \hat{n}_j)$. 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 routine 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 boundary-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.

II. C. 7. Regularity conditions

When the input parameter geom 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

$$S_n(\phi) \sim R^{lnl} \gamma(R^2)$$

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

$$V_{Rn}, V_{\phi n} \sim R^{ln-1} \gamma(R^2) \quad \text{for } n \geq 0.$$

Furthermore, at $R=0$, phase information for V_R and V_ϕ is redundant. We take ϕ to be 0 at $R=0$ and enforce $V_{\phi 1} = i V_{R1}$.

The entire functional form of the regularity conditions is not imposed numerically. The conditions are for the limit of $R \rightarrow 0$, 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_{Rn} and $V_{\phi n}$ to 0 for $n=0$ and $n \geq 2$. The phase relation for V_R and V_ϕ at $R=0$ is enforced by the following steps

- 1) At $R=0$ nodes only, change variables to $V_+ = (V_R + iV_\phi)/2$, $V_- = (V_R - iV_\phi)/2$.
- 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-vec routine is used to set $V_\phi = iV_-$ at the appropriate nodes.

Returning to the power series behavior for $R \rightarrow 0$, the leading order behavior for S_0 , V_R , and V_ϕ , is the vanishing radial derivative. This is like a Neuman 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, $\int dV \frac{w}{R^2} \frac{\partial v_i}{\partial R} \frac{\partial v_j}{\partial R}$, saved in the dr-penalty matrix structure, with suitable normalization to matrix rows for S_0 , V_R , and V_ϕ . 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 and V_ϕ , only. We should keep this in mind if problem arise with S_0 (or V_{z_0}) near $R=0$.]

II. 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 gradients steps in NIMROD's block-decomposed vector-type structures, and the matrix-type storage arrays have been arranged to optimize matrix-vector multiplication.

II. D. I. 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 to 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 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 matrix 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-types 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 $\tilde{A}z=r$ where \tilde{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-values 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 ($\sim \sqrt{\text{ave-factor}}$), inverting \tilde{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 $\Delta t_{\text{explicit-limit}} \ll \Delta t \ll T_A$). 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 III. O.). There is also diagonal preconditioning, which inverts local direction vector couplings only. This simple scheme is the only one that functions in both rblocks and tblocks. Computations with both block types and solver ≠ '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 idiosyncrasies 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.

II. D. 2. 3D linear systems

The conjugate gradient solve for 3D systems is mathematically similar to the 2D solves. 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 phs-type finite-element computation. For example, were the resistivity in our form of Faraday's law on p.19 a function of ϕ , it would generate off-diagonal in n contributions:

$$\sum_{j,n} B_{jn} \iiint dR dZ d\phi \frac{\partial (R_j Z_j)}{\partial \phi} \nabla \times (\alpha_j e^{in\phi}) \cdot \nabla \times (\bar{\alpha}_n e^{in\phi}),$$

which may be nonzero for all n'.

We can switch the summation and integration orders to arrive at

$$\iiint dR dz d\phi \nabla \times (\bar{a}_{j,k} e^{-in\phi}) \cdot \frac{\mu_0}{\rho_0} \left[\sum_{j,k,n} B_{j,k,n} \nabla \times (\bar{a}_{j,k} e^{ind}) \right]$$

Identifying the term in brackets as $\mu_0 \tilde{J}(R, z, \phi)$, the curl of the interpolated and FFT'ed \tilde{B} , shows how the product can be found as a rhs computation. The $B_{j,k,n}$ data are coefficients of the operand, but when they are associated with finite element structures, it 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.

II. E. Start-up routines.

Development of the physics model often requires new xblock-type and matrix storage. [The storage modules and structure types are described in Sect. II.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 non fundamental fields (like J which

is computed from the fundamental \vec{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 \vec{E}_3 component of be_eq structures use the covariant component ($R B_\phi$) in toroidal geometry, and the \vec{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.

II. 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. The 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 $|\vec{v} \cdot \hat{x}_i / |\hat{x}_i|^2|$ in each cell, where \hat{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 for preconditioning are computationally intensive operations. To avoid calling these operations during every time step, we determine how much the fields have changed since the last matrix update, and we compute new matrices when the change exceeds a

tolerance (ave-change-limit). [Matrices are also recomputed when At 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 Sect. III.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 $\overset{\text{an}}{bb}$ 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 preferable.] Thus, $\overset{\text{an}}{bb}$ 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.

II. 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 of extrap-q, extrap-nq, and extrap-int and define the new

values in extrap-init. Again, using existing code for examples is helpful.

II. H. History module (hist-mod)

The time-dependent data written to XDRRAW 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

Integrand differ in that there are no ~~as~~^{int} test functions.

The integrals are over physical densities.

II. I. I 10

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 parameter 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 parameter to 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 bcast_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.

III. Parallelization

Parallel communication introduction -
Grid block decomposition - Fourier

'layer' decomposition - Global-line
preconditioning

III. A. Parallel communication introduction

For the present and the foreseeable 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 Plington 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 tell each process the total number of processes. From this information and what is 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, let's return to the find-bb routine in utilities.f, for an 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. $\hat{b}\hat{b}$, 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 different 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.

III. 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 tbl(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 requests to every other process involved in the communication.

The general sequence of steps used in NIMROD for point-to-point communication is

1) issue an mpi-irecv to every other process in the exchange, which indicates readiness to accept data.

2) issue an mpi-send to every other 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 this verification is complete, the process can continue on to other operations.

* Caution: calls to non-blocking communication (`mpi-isend`, `mpi-irecv`, ...) seem to have difficulty with buffers that are part of F90 structures, when the mpich implementation of the library is used. The difficulty is overcome by passing the first element, like `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

```
iinode = block2proc(ibl-global)
```

assigns the node label for the process that owns global block # ibl-global—for the same layer as the local process—to the variable, iinode.

Separating processes by some condition (same layer, for example) 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 layers, '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 layers solve different linear

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.

III. C. Fourier 'layer' decomposition

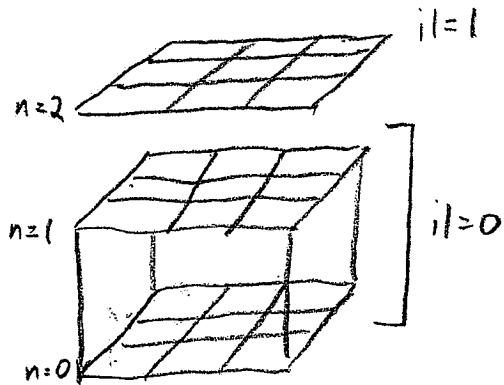
By this point it's probably evident that Fourier components are also separated into disjoint subsets, called layers. The motivation is that a large part of the NIMROD algorithm involves no interaction among different Fourier components. As envisioned until recently, the matrices would always be 2D, and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

Apart from the basic mpi_allreduce exchanges, communication among layers is required to multiply functions of toroidal angle [See I.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 $l_{\phi i} = 3$, $0 \leq n \leq \text{nmode_total} - 1$ with $\text{nmode_total} = 3$, and $n_{\text{layers}} = 2$. [n_{layers} is an input parameter, and the number of processors must be divisible by n_{layers} .] Since pseudospectral operations are performed one block at a time, we can consider a single-block problem without loss of generality. [see p. 115]

'Layer' Decomposition Swap

Normal Decomp.

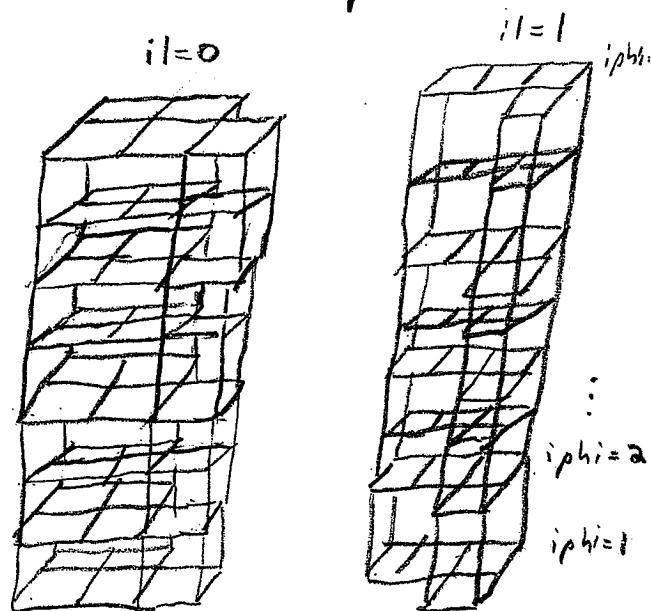


$n_{modes} = 2$ on layer $il=0$

$n_{modes} = 1$ on layer $il=1$

$$nf = mx \times my = 9$$

Configuration-space
Decomp.



$$n_{phi} = 2^{l_{phi}} = 8$$

$$m_{pseudo} = 5 \text{ on } il=0$$

$$m_{pseudo} = 4 \text{ on } il=1$$

parameters are $mx=ny=3$, $n_{layers}=2$, $l_{phi}=3$.

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.

III. C. Global-line preconditioning

That ill-conditioned 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 M -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 AIAA poster, "Recent algorithmic... on the web site.]

REFERENCES

The NIMROD web page is located at <http://nimrodteam.org/>

Abramowitz, M. and Stegun, I. A., eds., "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.

Glasser, A. H., Sovinec, C. R., Nebel, R. A., Gianakon, T. A., Plimpton, S. J., Chu, M. S., Schnack, D. D., and the NIMROD Team, "The NIMROD Code: a New Approach to Numerical Plasma Physics," *Plasma Phys. Control. Fusion* **41**, A747 (1999).

Harned, D. S. and Mikic, Z., "Accurate Semi-implicit Treatment of the Hall Effect in MHD Computations," *J. Comput. Phys.* **83**, 1 (1989).

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

Lerbinger, K. and Luciani, "A New Semi-implicit Approach for MHD Computation," *J. F., J. Comput. Phys.* **97**, 444 (1991).

Marder, B., "A Method for Incorporating Gauss' Law into Electromagnetic PIC Codes," *J. Comput. Phys.* **68**, 48 (1987).

Redwine, C., *Upgrading to Fortran 90*, Springer, 1995.

Schnack, D. D., Barnes, D. C., Mikic, Z., Harned, D. S., and Caramana, E. J., *J. Comput. Phys.* **70**, 330 (1987).

Strang, G. and Fix, G. J, *An Analysis of the Finite Element Method*, Wesley-Cambridge Press, 1987.