Corsica Users' Manual - DRAFT -

James A. Crotinger
Scott W. Haney
Thomas B. Kaiser

April 16, 2014

Preface

History

The *Corsica Project* began as a Lawrence Livermore National Laboratory (LLNL) project, funded by LLNL's Laboratory Directed Research and Development (LDRD), to develop algorithms for doing efficient and comprehensive simulations of toroidal magnetic fusion devices by coupling together existing simulations of disparate physical processes, including such effects as 2D edge physics and 3D micro-turbulence. The Corsica simulation code is a flexible and extensible transport code developed as a result of this research.

Credits

Many people have contributed to the Corsica code. The LDRD project was headed by James A. Crotinger and Ronald H. Cohen, and staffed by Michael Brown, Peter Brown, Gary G. Craddock, Scott W. Haney, Thomas B. Kaiser, Lynda L. LoDestro, Nathan Mattor, Jeff Moller, L. Donald Pearlstein, Thomas D. Rognlien, Aleksei I. Shestakov, Gary R. Smith, Alfonso G. Tarditi, and Xuequio Xu. Dick Bulmer, Bruce I. Cohen, Brian Yang and Raynard A. Jong have also made valuable contributions.

The Corsica project had three main thrusts: (1) quasi-static evolution of the magnetic geometry subject to evolving core-transport and external magnetic fields, (2) evolution of the core coupled to the edge, and (3) evolution of the core-transport coupled to micro-scale simulations of the turbulence driving the transport. These sub-projects, and the corresponding versions of the code, became knows as Corsica 1, Corsica 2, and Corsica 3. The principle team members for these sub-projects were:

- Corsica 1: Crotinger, Haney, LoDestro, Kaiser, Pearlstein.
- Corsica 2: Tarditi, Crotinger, Rognlien.
- Corsica 3: LoDestro, R. Cohen, Shestakov, Xu.

The Corsica code was largely an exercise in coupling existing codes together with efficient coupling algorithms. These previous code efforts deserve mention. The

free-boundary MHD-equilibrium code, TEQ, was written by Don Pearlstein, Lynda LoDestro, Dick Bulmer, Scott Haney, and Tom Kaiser. The edge code used by the Corsica 2 core-edge coupling project was UEDGE, developed by Tom Rognlien, et al. [1, 2] The turbulence simulation codes used by the Corsica 3 team included the Hawc code, written by James A. Crotinger [3], the HawcX code, written by Xuequio Xu [4], and finally the Gryffin code, written by Greg Hammett, Bill Dorland, and Michael Beer [5]. The core-transport code was written primarily by Scott Haney and James Crotinger, and is based in part on the SuperCode, developed by Scott Haney, et al. [6]

Many valuable contributions were made by our collaborators and customers, including Tom Casper, Bick Hooper, Dave Humphreys, and others.

Finally, we acknowledge the contributions of the Basis group. We continually pushed the limits of what Basis could do and the Basis group was quite helpful in fixing bugs and in making sure that new Basis releases were compatible with Corsica.

Chapter 1

Introduction

This chapter is meant to cover CORSICA basics. In practice, this introduction is spotty, being overly detailed in some areas and blank in others. As the later chapters are written, this will be corrected.

1.1 Basis

CORSICA is a Basis program. Basis is a software framework for developing interactive and programmable (or *steerable*) scientific programs. Basis provides CORSICA with its user interface (an embedded, Fortran-like, interpreted language), with several key software packages (graphics, binary I/O, and time history), and with a number of other *system* capabilities. This document assumes that the user is familiar with the Basis system. Refer to the Basis System documentation [7–12] for more information.[1] The Corsica Project added certain extensions to Basis to support the C++ programming language, and these will be documented in the Corsica developers manual.

1.2 The CORSICA Installation

The CORSICA system can be built from the source or installed as binary. To install from source, see the `README` file in the top directory of the source distribution. To install the binary distribution, copy the desired executable to a directory in your UNIX search path and create a link to the executable named "`corsica`":[2]

```
% mv corsica.20B9n $HOME/bin
% cd $HOME/bin
% ln -s corsica.20B9n corsica
```

[1]These documents are also available via the World Wide Web at
`http://www-phys.llnl.gov/X_Div/htdocs/basis.html`.

[2]My examples assume use of the UNIX C Shell and may require adjustment for other shells.

(The Basis runtime system gets the code's version number from the suffix of the actual executable, "`20B9n`" in this example. Basis's method for doing this requires that `corsica` be a soft link to the fully named executable, which is why it is installed in the above manner.)

Various parts of the CORSICA program are written in the Basis language. The source files for these parts of the code go in the *scripts* directory, along with some data files and help files. In order to run CORSICA, you need to install these scripts in a location of your choice (`<somewhere>` in the example below) and set the `CORSICA_SCRIPTS` environment variable accordingly:

```
% cd <somewhere>
% tar xf scripts.20B9.tar
% setenv CORSICA_SCRIPTS <somewhere>/scripts
```

The last line should probably be added to the user's `.cshrc` file (or the equivalent, if another shell is used).[3]

Finally, create a working directory, put any job-specific scripts and save files in that directory, and execute the `corsica` command. (If it is not found, you may need to run `rehash` to rebuild your shell's search-path hash table.)

1.3 The `corsica` Command

Unlike most Basis programs, CORSICA provides its own main program and has a custom input line and a variety of command line options. To learn about these options, type `corsica -help`:

```
% corsica -help
Usage: corsica [[option] [filename]]*
Options:
  -help:           Prints this message.
  -probname string Set 'probname' variable to string.
                   If this option is not specified,
                   'probname' is set using the name of
                   first input file. If no input files
                   are specified, 'probname' = 'problem'.
  -restore-hst name Restores the dump files and save files
                   saved with the save_transport command.
  -r name          Short version of -restore-hst.
  -plot:           Read 'ploteq.ezn' (default).
  -noplot:         Don't read 'ploteq.ezn.'
```

[3]Users and developers who maintain multiple versions of the code may want to have `corsica` be a shell script that sets the the appropriate environment variables and then `execs` the actual CORSICA executable. See the `scripts/corsica.sh` file for an example of how to do this.

```
    -log:             Create a log file 'probname.log' (default).
    -nolog:           Don't create a log file.
    -quiet            Keep Basis output to a minimum (default).
    -verbose          Print out excruciating amounts of output.
    -nostartup        Skip reading the startup files.
    -debug            Set debug=yes before starting.
    -exec string      Execute the Basis commands in string after
                      all other options and input files are
                      processed
  Up to 10 filenames can be specified. If the filename ends
  with the suffix 'sav', corsica will assume the
  file has been saved with saveq/saveqtr and will attempt to
  restore the equilibrium/transport case. If the suffix is
  'dat', corsica will execute 'readb filename'.
  Otherwise, the file is assumed to be a text file and
  corsica will execute 'read filename'. Files are read
  In the order specified on the command line.
```

Typical examples include running Corsica in fully interactive mode, running interactive but reading initialization scripts, and running in batch mode, reading all command from one or more input files. To run in fully interactive mode, starting from a binary save file, do:

```
% corsica -probname test1 d3d.sav
```

In this case, Corsica reads the saved state from the file `d3d.sav`, initializes itself, and then presents the user with a prompt, at which point he is free to do as he chooses. To read one or more Basis scripts (for example, to install custom sources, feedback control laws, etc), do:

```
% corsica -probname test2 d3d.sav mysource.bas
```

in which case the code will first restore the save file, then read the commands in `mysource.bas`, and then present the user with a prompt. To set up and run an entire problem in batch mode some additional preparations are required. If Corsica is to be run without user interaction, the input script should include the command (preferably at the beginning):

```
errortrp(off)
```

This turns off Basis's signal trapping, preventing Basis from prompting for user input if an error occurs. The script should also end with the `end` command. Then the batch run can be made as follows:

```
% corsica -probname test3 d3d.sav batch.bas >&batch.out &
```

The output can be monitored by running `tail -f batch.out` if desired.

The CORSICA variable `probname` is used to derive the output file names for the log file, plot files, and the files saved by the `save_transport` command. Setting this variable (with the `-probname` option) allows multiple runs to be done in the same directory without overwriting the output files of previous runs.

1.4 Corsica Basics

The basic CORSICA code includes both free and fixed boundary ideal MHD equilibrium solvers and a 1D transport code that solves the flux-surface averaged plasma transport equations in toroidal magnetic geometry (as specified by the MHD equilibrium solver). The details of these modules will be described in later chapters. In this section we describe the basic commands used in starting up, taking timesteps, and examining and saving results.

1.4.1 Starting up

As mentioned above, the usual way to start CORSICA is from a previously saved state. This is done by listing a *save file* (with a `.sav` extension) on the command line. There are two types of save files: *equilibrium save files* (`saveFileType == 1`), which contain only the saved MHD equilibrium state, and *transport save files* (`saveFileType == 2`), which contain both the saved MHD equilibrium state and the saved transport state[4]. The equilibrium save files are used primarily when doing MHD equilibrium work (machine design, stability analysis, etc.), and in the process of getting a transport run started for the first time (this process will be described later). Once transport has been run, the state is usually saved in a transport save file, which we'll assume as a starting point for now.

1.4.2 First steps: the `trans` function

Taking timesteps is accomplished with the function `trans`. The `trans` function accepts from 0 to 10 arguments, although only the first 4 are usually used. These are

```
ddt     -  the initial timestep.
ddtmax  -  the maximum allowable timestep.
tadv    -  the total amount of time to advance the equations.
niter   -  the maximum number of iterations per timestep.
```

If any of these are omitted, default values are used. (Furthermore, specifying a value for any of these resets the default value.) Here is the log from a complete session:

[4]A simple way to check the type of a save file is to run `basis open myfile.sav` and then type `saveFileType` at the Basis prompt to examine its value.

```
% corsica -probname foo d3d.sav
Beginning CGM File foo.001.ncgm
Beginning CGM Log foo.001.cgmlog
corsica> trans(0.001,0.001,0.001)
-----------------------------------------------------------------------
T_e(0)  =   1.791, T_i(0)  =   1.510, <T_e>_n =   0.913, <T_i>_n =   0.825
n_e(0)  =   0.397, <n_e>   =   0.292, c_alph  =   0.000, z_eff   =   1.000
pFus    =   0.000, pOhmic  =   0.804, pAux    =   2.107, nFuel   =  22.000
Ip      =   1.362, beta    =   0.746, beta_N  =   0.689, tau_E   =   0.077
q(0)    =   0.875, q_edge  =   4.732, li(3)   =   1.092, betap(1)=   0.284
-----------------------------------------------------------------------
res_psi(1)= 3.49D-05; res_f= 1.75D-04; resi(2)= 4.25D-04; res_mu_e=2.77D-05
res_psi(2)=-1.69D-04; res_f= 4.87D-04; resi(4)= 4.38D-04; res_mu_e=4.95D-05
res_psi(3)=-3.45D-04; res_f= 2.87D-04; resi(6)= 2.75D-04; res_n_d=3.71D-05
res_psi(4)=-4.07D-04; res_f= 4.41D-05; resi(7)= 8.37D-05; res_n_d=7.18D-05
res_psi(5)=-3.66D-04; res_f= 5.75D-06; resi(9)= 1.07D-04; res_n_d=5.98D-05
res_psi(6)=-3.98D-04; res_f=-8.29D-05; resi(11)= 0.00D+00; res_n_d=2.93D-05
-----------------------------------------------------------------------
T_e(0)  =   1.791, T_i(0)  =   1.510, <T_e>_n =   0.913, <T_i>_n =   0.825
n_e(0)  =   0.397, <n_e>   =   0.292, c_alph  =   0.000, z_eff   =   1.000
pFus    =   0.000, pOhmic  =   0.805, pAux    =   2.107, nFuel   =  22.000
Ip      =   1.362, beta    =   0.747, beta_N  =   0.689, tau_E   =   0.077
q(0)    =   0.875, q_edge  =   4.727, li(3)   =   1.091, betap(1)=   0.284
-----------------------------------------------------------------------
    t = 1.00000D-03;   CPU time(6)= 7.23334D+00
Total CPU time: 1.62167D+01
corsica> saveqtr("new.sav")
corsica> quit
Output files:
/home/gandalf/jac/corsica2/doc/new.sav: PFB File
/Closed CGM File foo.001.ncgm,     1 frames.
Closed CGM Log File foo.001.cgmlog

   CPU (sec)   SYS (sec)
      37.433       2.183
```

There are several things to note. After the `corsica` command, which was described above, there are two messages that indicate that a CGM and CGM log file have been started. These are issued when `corsica` initializes the Basis graphics package. Next there is the "`corsica> `" prompt, at which the `trans` command was typed:

```
      corsica> trans(0.001,0.001,0.001)
```

This command instructs CORSICA to take a single 1 ms. timestep. Following this command is a table of physical quantities, the *diagnostic table*. This table is printed at the beginning of the first timestep (on the first call to `trans` only), and at the end of every successful timestep. Next are six lines listing *residuals*—iterate-to-iterate changes of various quantities:

- `resi` - a measure of the equilibrium code convergence (in free-boundary mode this is replaced by `resj`).
- `res_psi` and `res_f` - related to the convergence of the coupled equilibrium-transport system.
- `res_X`, where `X = {y, mu, nd, ...}` - is the maximum residual from the transport solutions.

The printing of these residuals can be turned off or made more verbose with the variable `nprt59`. (*Unfortunately the documentation for this variable is not up-to-date in* `eq.v`*, so "*`list nprt59`*" does not currently give the correct information, and I'm not even sure what all of the options are. I do know that* `nprt59 = -3` *turns virtually everything off, and* `nprt59 = 1` *turns on tons of equilibrium output.*) Once the timestep iteration converges, the diagnostic table is printed again, followed by the simulation time and the CPU time for the step. After all steps finish, the total CPU time for the `trans` call is printed.

At this point `saveqtr("new.sav")` was called to save the state of the code to a new transport save file, and `quit` was typed to end the run. The binary I/O package then printed a list of all binary files written, the plotting software issued messages when it closed its files, and Basis printed the total CPU and system time consumed by the run.

The transport equations are generally quite nonlinear and each timestep must be done iteratively. If too large of a timestep is attempted, this iteration may fail to converge, and occasionally it may diverge and cause a floating point error. (This can also happen if the MHD equilibrium module starts to have trouble finding the equilibrium, in which case decreasing the timestep may not help.) The code does not automatically recover from these errors. Instead it returns to the prompt and waits for user input. At this point the user may wish to examine the data to determine the problem. The user can back up and try to continue taking timesteps by calling the `redo_timestep` function. This should restore the state to that which existed after the last successful timestep. This mechanism is not bulletproof, though, so it is useful to occasionally save dumps using `saveqtr` (this can even be automated using the history packages, discussed below).

1.4.3 Time-stepping algorithm in the `trans` function

The time stepping algorithm is defined in the `adjust_algorithms` function. The algorithm is used to vary the size of the current time step (dt) between the initial time step value (ddt) and the maximum time step value (ddtmax) for use in the function `trans`. After each successful time step, a check is made to vary the time step. The decision to increase or lower the time step depends on the value of the variable iter, the number of iterations used for the previous time step to converge. If iter is less than iter_timered (default value is 8), then the code attempts to increase the time step. If iter is greater then iter_timeinc (default value is 20). the code attempts to decrease the time step. If the value of iter is in the range (8,20), the time step size is left unchanged.

The method of adjusting the timestep is determined by the parameter adj_dt_meth. If adj_dt_meth=0 (the default), then the current time step dt is adjusted by adding or subtracting the value of ddt. If the value of adj_dt_meth=1, then the current time step dt is adjusted by multiplying or dividing by a factor named dt_mult which has a default value of 2. The value of dt_mult is user settable, but the logic requires that dt_mult be greater than unity.

1.5 Looking at the Data

Corsica has over 3000 functions and variables that can be accessed from the Basis parser (including those that belong to Basis). In this section we'll introduce the more commonly accessed ones.

1.5.1 Magnetic geometry and flux surface labels

Corsica makes the fundamental assumption that all transport-timescale phenomena are axisymmetric. Thus the slow-timescale behavior can be modeled as 2D. Furthermore, inside the confined region of the plasma (inside the *last closed flux surface*), fast parallel transport processes are assumed to quickly make the important transported quantities constant on a flux surface. Thus the evolution of these quantities can be modeled with 1D flux-surface averaged transport equations, where the 1D coordinate system is found by solving the 2D Grad-Shafranov equation for the ideal magnetohydrodynamic (MHD) equilibrium.

For now we'll take the magnetic geometry as a given and simply state that when Corsica is run, an ideal MHD equilibrium is calculated consistent with the profiles. The result of this calculation is the poloidal magnetic flux function, $\psi(R,Z)$, where R is the radial coordinate and Z is the vertical coordinate. This equilibrium can be viewed with the `layout` command; for example:

```
    corsica> layout(0,0)
```

produces the plot shown in Fig. (1.1). The `layout` command plots cross-sections of the poloidal field coils, the vacuum vessel (and other passive structure), and the flux surfaces (in a free-boundary run these are plotted both inside and outside the separatrix).

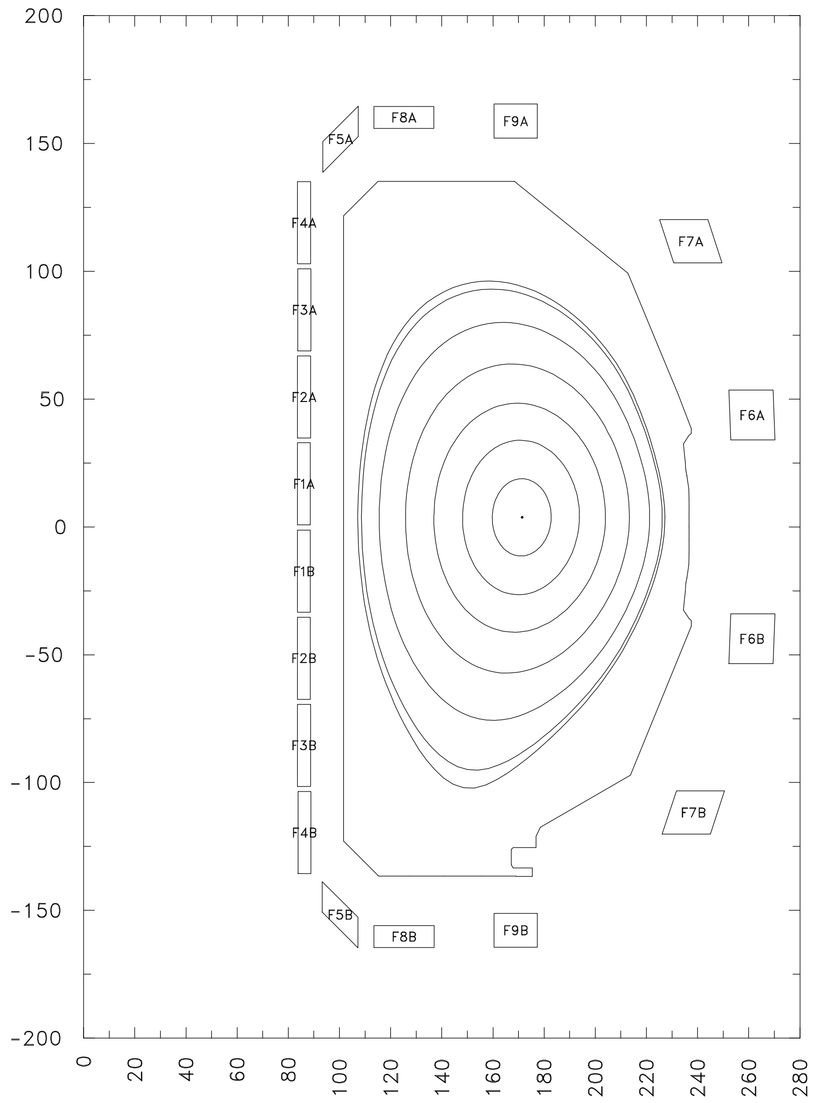


Figure 1.1: The `layout` plot.

The flux surfaces can be labeled by any quantity that is constant on a flux surface and varies monotonically from surface to surface. Several common choices are listed in Table 1.1, along with their CORSICA variable names, where applicable. In our

Table 1.1: Flux surface labels.

symbol	*definition*	*variable name*	*units*
Ψ	poloidal magnetic flux	`polFlux`	Wb
Φ	toroidal magnetic flux		
$\bar{\rho} = \Phi/\Phi_m$	normalized toroidal flux ($\Phi_m \equiv \Phi_{boundary}$)	`rhobar`	—
V	flux surface volume	`vTor`	m^3
r	minor radius = (horizontal diameter) / 2	`rminTor`	m

formulation of the transport equations, the normalized toroidal flux, $\bar{\rho}$ (`rhobar`), is used as the independent variable labeling each flux surface. However `polFlux`, `rminTor`, and other variables are available for use in calculations, plots, etc.

1.5.2 Transported quantities

In this section we introduce the variables that are evolved by the transport code, and various derived quantities. Table 1.2 shows the symbols and variable names for the density, temperature, and safety factor—flux functions that are commonly used to specify the state of the plasma. Note that the `densTor` and `tempTor` ar-

Table 1.2: Basic plasma variables.

symbol	*definition*	*variable name*	*units*
n, n_j	density (of species j)	`densTor(,j)`	$10^{20}/m^3$
T_e, T_i	electron and ion temperatures	`tempTor(,j)`	keV
$q = \frac{1}{2\pi} d\Phi/d\psi$	safety factor	`qTor`	—

rays hold the densities and temperatures for all species. The CORSICA code defines symbolic selectors to use in order to specify the species index. Several of these are described in Table 1.3. Thus the electron temperature at grid-point 3 is given by `tempTor(3,iElec)`, and the ion density in the center is `densTor(1,iIon)`. To plot the deuterium density as a function of minor radius, the command

```
corsica> plot densTor(,iDeut) rminTor
```

Table 1.3: Species selectors.

variable	*description*
`iElec`	electron temperature or density
`iIon`	ion temperature or density
`iHyd`	hydrogen density
`iDeut`	deuterium density
`iTrit`	tritium density
`iAlph`	helium density
`iBery`	beryllium density
`iCarb`	carbon density

is issued. (See Ref. [7], Sec. 3.5 and Ch. 4, for an introduction to Basis array syntax and Basis graphics. Refer to Ref. [8] and Ref. [9] for details.)

Although the n, T, and q are natural state variables, they are not necessarily the most convenient variables to evolve. In our formulation of the flux-surface averaged transport equations, the dependent variables are derived from the adiabatic invariants, quantities that remain constant on flux surfaces under very rapid changes of the magnetic geometry. These invariants are the safety factor, q, the number of particles on a flux surface, nV' (where $V' = dV/d\bar{\rho}$, `vprTor` in the code) , and the *entropy density*, pV'^{γ}, where $\gamma = 5/3$. In transforming the equations to normalized flux coordinates, $(\bar{\rho})$, slightly different dependent variables are introduced in order to simplify the convective terms arising from the possible time dependence of Φ_m. These variables are summarized in Table 1.4.[5] For most purposes one can think of

Table 1.4: Normalized dependent variables.

symbol	*definition*	*variable name*
N_j	$\frac{n_j V'}{\Phi_m^2}$	`nTor(,j)`
μ_j	$\frac{p_j V'^{5/3}}{\Phi_m^{8/3}}$	`muTor(,j)`
y	$\frac{C_2 C_3 \Phi_m}{2\pi q}$	`yTor`

the transport code as evolving n, T, and q, but it is useful to keep in mind that N, μ, and y are the actual dependent variables.

[5] C_2 and C_3 are geometric coefficients that will be defined later.

1.5.3 The PDE solver and boundary conditions

The transport equations are solved using a Galerkin cubic-spline finite element technique. This technique will be explained in detail in a later chapter. There are a few implications of this choice that will be mentioned here. First, the actual quantities being advanced by our time-advance algorithm are the coefficients in the B-spline representation of the cubic spline. There are two more such coefficients than there are grid points through which the spline passes (the extra two degrees of freedom are used to specify or calculate the derivatives at the end-points). In Table 1.5 we list some of the important variables related to the solver. The calculation of `nTor`

Table 1.5: B-spline coefficient arrays and dimensions.

variable	*description*
`nFP(,j)`	fitting parameters for `nTor(,j)`
`muFP(,j)`	fitting parameters for `muTor(,j)`
`yFP`	fitting parameters for `yTor`
`mtor`	number of grid points on the `rhobar` grid
`nfps`	number of fitting parameters (`mtor + 2`)

from `nFP`, etc., is handled by the function `computeTrProfiles`. The inverse can be accomplished with the function `bsplineInterp`.

The transport equations are parabolic equations and thus require boundary conditions at the outer edge (the center is not a boundary). These boundary conditions are specified using the arrays listed in Table 1.6. First note that `densBC` and `tempBC` do

Table 1.6: Boundary condition arrays.

variable	*description*
`densBC`	boundary condition for `densTor`
`tempBC`	boundary condition for `tempTor`
`yBC`	boundary condition for `yTor`

not specify the boundary conditions on `nTor` and `muTor`, but rather on `densTor` and `tempTor`. This is in keeping with our desire to allow the user to work with the more physical variables. Second, `densBC` and `tempBC` are arrays that allow the specification of a mixed boundary condition, as follows:

$$[\mathtt{densBC(1)} \cdot \langle \nabla n \rangle + \mathtt{densBC(2)} \cdot n]|_{\bar\rho=1} = \mathtt{densBC(3)}$$

where $\langle\cdot\rangle$ indicates the flux surface average. Finally, in deriving the matrix equations that are solved to advance the B-spline coefficients, we have made use of the mixed boundary condition to eliminate certain boundary terms. A consequence of this is that a purely Dirichlet boundary condition, for example `densBC(,iDeut) = [ 0.0, 1.0, 1.0 ]`, is not allowed. However this can be approximated by making `densBC(1,iDeut)` very small. (Actually, the code will accept pure Dirichlet BCs if the equations have a convective term (which all of our equations do have), but this is not completely general and the current implementation results in a loss of accuracy. However, there are instances where convergence is improved by choosing the pure Dirichlet BC over its approximation. Experimentation is required to determine whether this is worth the lost accuracy.)

1.5.4 Transport models and sources

The transport equations are 1D conservation laws, and thus have the generic form:

$$V'\frac{\partial u}{\partial t} + \frac{\partial}{\partial \bar{\rho}}\left(\Gamma_u V'\right) = S_u V'$$

where Γ_u is the flux of u and S_u is the source. Table 1.7 lists the arrays that hold flux and source data for the density and energy equations. Note that these are not input variables, but intermediate quantities. The fluxes are related to the state variables and their gradients via a *transport model*. The total sources are calculated from the various source models in the code. Also note that the `particleFlux` and `heatFlux` are in flux coordinates and thus do not have the usual units. They are defined such that multiplying by `vprTor`, the differential volume in flux geometry, gives the total flow of particles or heat across the flux surface.

Table 1.7: Flux and source variables.

variable	*description*
`particleFlux(,j)`	flux of particle species `j`
`heatFlux(,j)`	flux of heat (`j = iElec, iIon, iTot`)
`particleSrc(,j)`	source of particle species `j`
`heatSrc(,j)`	source of heat (`j = iElec, iIon, iTot`)

The CORSICA code has several transport models available, and it is easy to add others. The function `setTrModel(iModel,iDefault)` is used to change the transport model, where `iModel` is the model number and `iDefault` specifies whether the default model parameters (if the model has any) are to be reset or not. The model numbers for the current models are given in Table 1.8. The RLW and KDBH models are *critical gradient* models that can produce zero flux for certain profiles. For this reason

Table 1.8: Transport Model Selectors.

variable	*description*
`iSimple`	Simple model $[\chi, D \sim \frac{a^2}{\tau}F(\bar{\rho})]$
`iChangHinton`	Chang-Hinton neoclassical model
`iRLW`	Rebut-Lallia-Watkins critical temperature gradient model
`iKDBH`	Kotschenreuther-Dorland-Beer-Hammett transport model
`iUser`	User supplied transport model

they are not usually used on their own, but are added to the calculations of other transport models. The `iRLW` option includes the calculation of the neoclassical model, and the `iKDBH` option adds the KDBH prediction to that of the "Simple" calculation. These critical gradient models also pose another problem: the usual time-advance iteration tends to be unstable due to the dependence of the transport coefficients on the gradients. This instability is eliminated by averaging the transport coefficients over the iterates. The variable `use_avg_trcoefs` controls whether this averaging is done. (Various options are available, including straight averaging over all iterates, and relaxation (also known as an "exponential moving average") are available. See the file `average.bas` for details.)

The predictions of a model can also be calculated without actually using the model in the time advance. This is done using the functions listed in Table 1.9. This table also lists the names of the variables into which the predictions are stored. Note that

Table 1.9: Transport Model Functions.

function name	*output variable(s)*
`evalSimple`	`chiSimple, diffSimple`
`evalNeo`	`chiNeo, diffNeo, vWarePinch`
`evalRLW`	`chiRLW, diffRLW, pinchRLW`
`evalKDBH`	`chiKDBH`
`evalUserTransport`	`chi, diff`

`evalRLW` and `evalKDBH` only calculate the RLW and KDBH transport coefficients, and do not include the neoclassical or Simple contributions.

The `iUser` option allows the transport model to be determined by calling a user-defined function. Thus one can try new models, or combine existing models using the "`eval`" functions described above. In order to use the `iUser` model, a function that sets `chi`, `diff`, etc., must be written. Then the name of this function is assigned to the variable userTransportFunction and `setTrModel(iUser,0)` is called.

The transport equation for the magnetic flux, or for y, is derived from the flux-surface-averaged Ohm's law. The neoclassical theory is used for the transport model (neoclassical conductivity for the diagonal contribution and the bootstrap current for the off-diagonal parts). These calculations are done in concert with the equilibrium calculation and are not affected by `setTrModel`.

CORSICA also has a time-averaged sawtooth model that is controlled by the variable `useSawtoothModel`. This model operates by modifying the other transport coefficients, including the conductivity, inside the $q = 1$ surface.

The CORSICA code has several models for particle, heat, and current sources, and we are anxious to add additional ones. Currently we have

- Primitive gas puffing and particle recycling.
- Pellet fueling.
- Neutral beam fueling (*in development*).
- Ohmic heating.
- Synchrotron and Bremsstrahlung radiation loss.
- Drag heating/loss.
- Neutral beam heating.
- Fusion heating.
- Generic particle and heat sources (see `pAux` and `nFuel`).
- Neutral beam current drive.
- Generic current drive source (`jICTor`).

These will be documented in detail later. The user can use the on-line documentation to begin learning about these topics by doing `list ctr.groups`, and then listing the groups that are of interest.

1.5.5 History

CORSICA provides a easy-to-use interface to the Basis history package, `hst` (for details on `hst` see [12, Ch. 7]). The history of a CORSICA variable is stored in an array whose name is derived by appending "`_hst`" to the end of the variable's name. This array will have one more dimension than the original variable, with the last dimension being indexed by time. For example, the history of `densTor` is `densTor_hst`, and its shape is `[mtor, nparts, `*`nsteps_p`*`]`, where *nsteps_p* is the number of times that `densTor`'s value has been collected by the history system. The history of `betae`, the plasma β calculated using the toroidal field, is `betae_hst`, a 1D array of length *nsteps_s*.

Note that *nsteps_s* and *nsteps_p* are not necessarily the same. CORSICA groups variables into categories (using history *tags*) in order that they may be collected at different frequencies. Four history tags are pre-defined: `statout`, `profout`, `coilout`, and `gridout`. The `statout`, or "stats," tag is used to collect time histories of 0D quantities (various global parameters, volume averages, etc., like `betae`). The "profs" tag is used to collect histories of various profiles, like `densTor`. The "coils" tag is used to collect histories of variables related to the circuit equations for the external conductors, and is only used in free-boundary mode. Finally, the "grid" tag is used to collect fully 2D data, such as $\psi(R, Z)$ (`eq.psi`). The initial frequencies of collection are set by the variables `collect_stats`, `collect_profs`, `collect_coils`, and `collect_grids`. These arrays have the following interpretations:

```
collect_stats(1)  -  first timestep to collect statout (default 0)
collect_stats(2)  -  last timestep to collect statout (default 1e7)
collect_stats(3)  -  the number of timesteps between collections
```

The default frequencies are

```
collect_stats(3)  =   1
collect_profs(3)  =  10
collect_coils(3)  =   5
collect_grids(3)  = 1e4
```

Note, however, that these defaults may be overridden by values in the save file used to start the run. Also note that changing these values after the first call to `trans` has no effect. To increase the collection frequency at this point, use the Basis `andcollect` command [12, Ch. 7].

The lists of variables that have their history tracked are set up in the file `scripts/-stats.in`. To customize these lists, simply copy `stats.in` to your working directory and edit it. There are four initialization routines, one for each tag. Add new variables using the appropriate command (`stat_add`, `prof_add`, etc.). (Note that these commands are limited to a maximum of nine items per line. See the file `scripts/statout.bas` for the implementation of these commands.)

Each history tag includes a *time* array that tracks the simulation time (`time`) at which the tags were collected. These arrays are:

```
time_h  -  time array for statout
time_p  -  time array for profout
time_c  -  time array for coilout
time_g  -  time array for gridout
```

These are commonly used in plotting commands; for example to plot the central ion density and the volume averaged ion density on the same graph, one could do:

```
corsica> plot densTor_hst(1,iIon,) time_p color=red
corsica> plot densVolAvg_hst(iIon,) time_h color=green
```

Currently the history data is collected in memory. This can cause memory shortages. For this reason, `collect_profs(3)` is set to 10 by default. For very long simulations the user may want to increase it even further, and then make use of the `andcollect` command to increase the collection frequency in critical parts of the simulation. When debugging or doing short runs, the user may wish to decrease `collect_profs(3)` to 1. (Recent versions of Basis have the ability to collect the history directly on disk. We may change to this mode of operation in the future.)

The `save_transport` command does a `saveqtr` and also saves the history arrays to disk (with each tag getting its own file).

We'll wrap up this section with an example of using the history facility to save periodic dumps. We will do this by defining a new history tag and a *tagaction* associated with this tag. The tagaction will be to call a function that will do the dumps. First the history tag needs to be initialized. This can be done with the `newtag` command, but this is not necessary as the `collect` command will define it implicitly. Thus we issue:

```
corsica> collect dumptag 0 10000000 20
```

Recall that this means that the `dumptag` tag will be collected every 20 steps, starting at step 0 and ending at step 10000000. Next use `tagaction` to specify a function to call when `dumptag` is collected:

```
corsica> tagaction dumptag "dumpfcn"
```

Finally we must write `dumpfcn`. Here is an example of how to do this:

```
function dumpfcn
  integer nc = int(1000*time + sqrt(errt))
  character*24 savefile =
          trim(probname)//"-dump"//format(nc,0)//".sav"
  saveqtr(savefile)
  << "Saved dump at " << nc << " ms."
endf
```

The first statement of this function converts `time` into milliseconds and assigns it to an integer. The second statement uses this number to build up a filename (e.g. `foo-dump200.sav`). Next the file is saved to the specified filename, and finally a message is printed to the terminal.

Note that the history collection can also be specified in terms of `time`, rather than step number. To do this, simply change the arguments to `collect` to be real numbers; for example:

```
corsica> collect dumptag 0.0 1000.0 .010
```

tells Basis to collect the history every 10 milliseconds. Note, however, that history is collected when `time` $\geq n * 0.010$, so roundoff can lead to to surprises. These can be mostly avoided by adding a very small increment to the `time` variable:

```
corsica> time = time + 1.0e-7
```

1.6 An Example

In this section we present a simple DIII-D negative current ramp simulation and some of the output. First an input file is prepared that performs the desired simulation. This input file is shown in Fig. (1.2).

The first command, `shutup`, tells CORSICA to turn off all of its default output statements. Next, `trans_init` is called. Ordinarily this is not called explicitly as it is done automatically when `trans` is called for the first time. However, we need to call the history command `andcollect` before we start taking timesteps, and this cannot be done unless the tag we are calling it with has already been defined, which is done by `trans_init`.

```
call shutup  # absolute minimal output

### Initialize transport. This is done so that andcollect
### can be called before we take any timesteps.

call trans_init

### Set up the ohmic control source to do a 400 kA
### negative current ramp starting at t = 1.0 and
### ending at t = 1.5.

real plc_ramp = 40000  # units = Amps * 10.

real plc_start = plascur0
real plc_end   = plascur0 - plc_ramp

real tr_start  = 1.0
real tr_end    = 1.5

function plc_fnc(t)
  if (t < tr_start) return plc_start
  if (t > tr_end) return plc_end
  return plc_start +
    (t-tr_start)/(tr_end-tr_start) * (plc_end - plc_start)
endf

function set_plascur
  plascur0 = plc_fnc(ctr.time)
endf

add_hook("userSource_hooks","set_plascur")

### Program history to collect data more frequently
### after current ramp starts.  Our timestep will be
### greater than 0.01, so this should collect every
### step.

andcollect profout 1.0 2.0 0.01
andcollect profout 2.0 5.0 0.1

### Step to t = 1.0

real cpu_start_time = second(0)

trans(0.05,0.25,1.0)

<< "CPU time to reach start of Ip ramp = " \
            << second(0) - cpu_start_time

trans(0.02,0.02,0.5)
trans(0.02,0.25,0.5)

<< "CPU time to reach 2.0 seconds = " \
            << second(0) - cpu_start_time
```

Figure 1.2: Input file for current ramp example.

Next we set up some variables that control the current ramp and a function, `plc_fnc(t)`, that returns the desired current as a function of time. This particular simulation is being run with the fixed boundary equilibrium solver and with `fake_ohmic == true`, an option that causes the code to feedback on the total toroidal flux until the plasma current converges to `plascur0`, the desired value (which has units of `Amps * 10` - certain parts of the equilibrium code are in cgs). The function `set_plascur` is thus used to set `plascur0` to the desired plasma current. We then inform Corsica that it needs to call `set_plascur` whenever the sources are evaluated. This is done by adding the string `"set_plascur"` to the list of *hook* functions `userSource_hooks` with the `add_hook` function (a *hook* is a function that operates entirely by side-effects - in this case setting a global variable by calling a global function; such functions take no arguments and return no values).

Now the `andcollect` commands are called. The plasma will be changing rapidly during the current ramp and for a time following the end of the ramp, and so we instruct Basis to collect this history every 10 msùntil $t > 2.0$, and then every 100 ms. until the end of the run.

Finally we have the commands that do the actual time-stepping, stepping first to the start of the current ramp and printing the CPU time, and then stepping through the ramp and on out to $t = 3.0$ seconds.

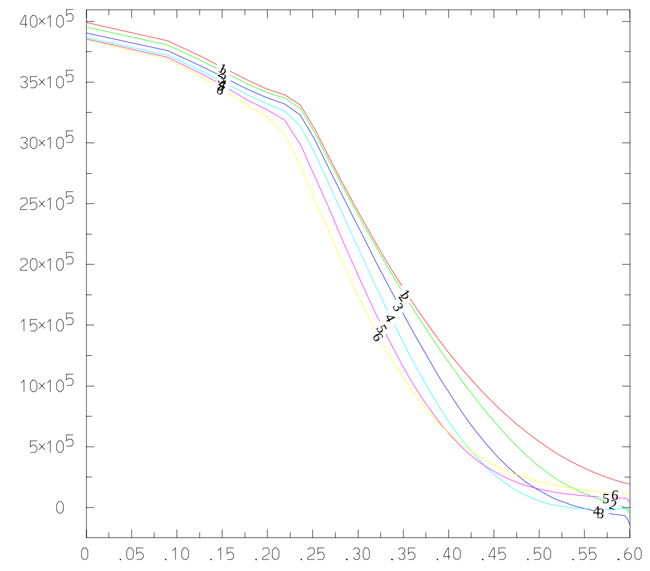
Now we run the problem as shown below:

```
% corsica -probname ex1 example.sav
Beginning CGM File ex1.001.cgm
Beginning CGM Log ex1.001.cgmlog
corsica> read ex1.bas
CPU time to reach start of Ip ramp = 1.56528E+01
CPU time to reach 2.0 seconds = 2.83051E+02
corsica> win
corsica> cgm close
Closed CGM File ex1.001.cgm,     1 frames.
Closed CGM Log File ex1.001.cgmlog
corsica> nf; plot jtTor_hst(,::10) rminTor color rainbow
corsica> ezcsetbw
corsica> cgm send
Beginning CGM File ex1.004.cgm
Beginning CGM Log ex1.004.cgmlog
corsica> cgm close
Closed CGM File ex1.004.cgm,     1 frames.
Closed CGM Log File ex1.004.cgmlog
corsica> nf; plot transpose(qTor_hst(::10,)) time_p color rainbow
corsica> nf; plot li_hst(1,) time_h color red
corsica> plot betap_hst(1,) time_h color green
```

The actual simulation is done when `ex1.bas` is read. Because `shutup` was called, only the lines giving the CPU time are printed. The `win` command opens a graphics window, and `cgm close` closes the default CGM file. The `plot jtTor_hst...` command plots the history of the current density profile ($\langle J \cdot B \rangle \,/\, \langle B \cdot \nabla\phi \rangle$). This plot is shown in Fig. 1.3. Next, `ezcsetbw` is called, setting the background color to white, rather than the default black, so that it can be converted to postscript (which has no "background color").[6] Then `cgm send` opens a new CGM file and saves the plot, and `cgm close` closes the file.[7] (The plots were saved into individual files because certain software packages, such as IslandDraw, are not capable of importing CGM files containing multiple frames. An alternative approach is to save everything in the default CGM file and then later extract the desired frames with NCAR's `ctrans` or an equivalent tool.) The next line contains the `nf` command and a `plot` command. The `nf` command tells the `ezn` graphics package to advance to the next frame. If `nf` is not called, all plotting commands act on the same figure, as will be shown below. The `plot` command makes use of Basis vector notation and the `transpose` function to plot the time history of `qTor` at every tenth grid point. This plot is shown in Fig. 1.4. Finally, the the internal inductance and the poloidal beta are plotted on the same plot (note the lack of `nf` before the final plot), and are shown in Fig. 1.5.

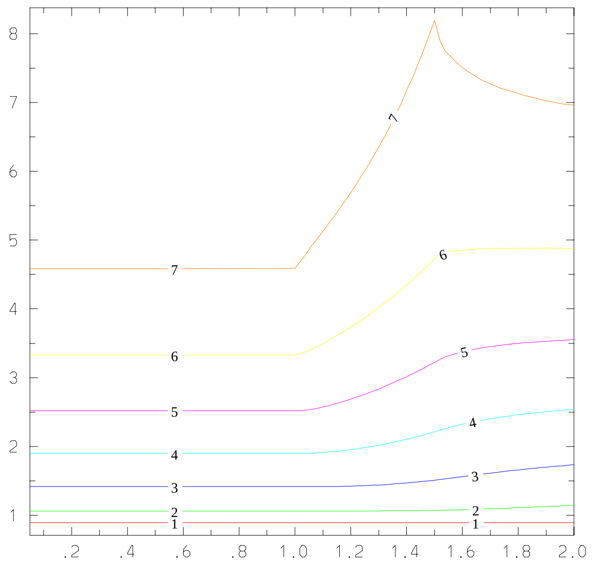
[6]This is unnecessary if Basis 11.3 or newer is used.

[7]Several plots were made before this and not all were saved, which is why the CGM file is #4.



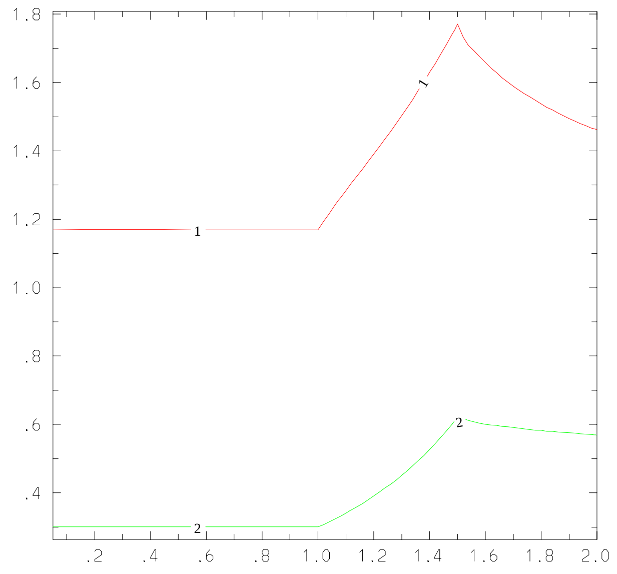
1-6: plot jtTor_hst(,::10) rminTor color=rainbow

Figure 1.3: Current density evolution.



```
1-7: plot transpose(qTor_hst(::10,)) time_p color=rainbow
```

Figure 1.4: The history of qTor at various gridpoints during negative current ramp.



```
1: plot li_hst(1,) time_h color=red
2: plot betap_hst(1,) time_h color=green
```

Figure 1.5: Evolution of L_i and β_p during current ramp.

Chapter 2

Transport Analysis

2.1 Introduction

The transport analysis module of the CORSICA code consists of a Basis script file containing a collection of parser functions that are useful in computing core transport coefficients and fluxes from arbitrary density, temperature and source profiles and metric coefficients derived from a self-consistent TEQ equilibrium. The available coefficients are:

- particle diffusion coefficient for electrons, D_e
- particle diffusion coefficients for individual ion species, D_k
- average particle diffusion coefficient for summed ion species, D_i
- pinch velocity for electrons, V_{Pe}
- pinch velocities for individual ion species, V_{Pk}
- average pinch velocity for summed ion species, V_{Pi}
- thermal conductivity for electrons, χ_e
- thermal conductivity for summed ion species, χ_i

In addition, the particle fluxes, $\Gamma_{e,i}$, and heat fluxes, $q_{e,i}$, can be computed. If the plasma is not in transport steady state, the time dependence of the density and temperature profiles and boundary toroidal flux must also be specified.

In what follows the analytic expressions used to compute the transport coefficients and fluxes will be presented, and detailed instructions given for using the module to analyze transport properties of arbitrary plasmas.

2.2 Definitions and Analytic Expressions for Numerically Computed Quantities

CORSICA solves transport equations for the number densities of several species of ion, and for the electron and total-ion entropy densities. The number-density equations have the form

$$\frac{\partial N_k}{\partial t} + \frac{\partial \hat{\Gamma}_k}{\partial \bar{\rho}} + v_{\bar{\rho}}(N_k - \bar{\rho}\frac{\partial N_k}{\partial \bar{\rho}}) = \frac{V'}{\Phi_m^2} S_{nk}, \tag{2.1}$$

with the ion species indexed by the subscript k. In this and subsequent equations, the independent variables are time, t, and the dimensionless flux-surface label,

$$\bar{\rho} \equiv \frac{\Phi}{\Phi_m}, \quad 0 \leq \bar{\rho} \leq 1,$$

where Φ is the toroidal flux and $\Phi_m(t)$ its value at the plasma surface. Differentiation with respect to $\bar{\rho}$ will be denoted by a prime. Other quantities appearing in the number density equation are the density variable N_k defined in Table 1.4; the particle flux variable $\hat{\Gamma}_k$, which can be expressed in terms of transport coefficients by the closure relation

$$\hat{\Gamma}_k = -C_1 D_k \left(\frac{\partial N_k}{\partial \bar{\rho}} - \frac{V''}{V'} N_k \right) + C_4 V_{Pk} N_k, \tag{2.2}$$

where $V' = 4\pi^2 \langle \mathcal{J} \rangle$, $V'' = \partial V'/\partial \bar{\rho}$ and $\mathcal{J} = (\nabla\phi \cdot \nabla\bar{\rho} \times \nabla\theta)^{-1}$ is the Jacobian, $C_1 = \langle \mathcal{J} |\nabla\bar{\rho}|^2 \rangle / \langle \mathcal{J} \rangle$ and $C_4 = \langle \mathcal{J} |\nabla\bar{\rho}| \rangle / \langle J \rangle$ are metric elements, D_k is the particle diffusion coefficient, and V_{Pk} is the pinch velocity; the surface "velocity" $v_{\bar{\rho}} = d \ln \Phi_m / dt$; and the physical particle source, S_{nk}. The physical particle flux is

$$\Gamma_k = \frac{\Phi_m^2}{V'} \hat{\Gamma}_k \tag{2.3}$$

We determine it from profile and geometric data by integrating Eq.(2.1) with respect to $\bar{\rho}$:

$$\hat{\Gamma}_k = \int_0^{\bar{\rho}} d\rho \left[\frac{V'}{\Phi_m^2} S_{nk} - \frac{\partial N_k}{\partial t} - v_{\bar{\rho}} \left(N_k - \rho \frac{\partial N_k}{\partial \rho} \right) \right]$$

Note that only the first term in the integrand is necessary if the plasma is in transport steady state.

From Eq.(2.2) it's clear that $\hat{\Gamma}_k$ determines only a linear combination of D_k and V_{Pk} on each flux surface; to compute one of these transport coefficients from profile data at a single time point requires knowledge of the other. Thus, if V_{Pk} is assumed known, say from a transport model, we can obtain D_k from (2.2) as

$$D_k = -\frac{\hat{\Gamma}_k - C_4 V_{Pk} N_k}{C_1(N_k' - V'' N_k/V')} = -\frac{\Gamma_k - C_4 V_{Pk} n_k}{C_1 n_k'}. \tag{2.4}$$

If, on the other hand, we assume D_k to be known, V_{Pk} follows from

$$V_{Pk} = \frac{\hat{\Gamma}_k + C_1 D_k (N_k' - V'' N_k/V')}{C_4 N_k} = \frac{\Gamma_k + C_1 D_k n_k'}{C_4 n_k}. \tag{2.5}$$

If Eq.(2.1) is summed over ion species (in the remainder of this chapter, all sums on k are over individual ion species) and the definitions

$$N_i \equiv \sum_k N_k, \quad S_{ni} \equiv \sum_k S_{nk}, \quad \hat{\Gamma}_i \equiv \sum_k \hat{\Gamma}_k, \tag{2.6}$$

$$\Gamma_i \equiv \frac{\Phi_m^2}{V'} \hat{\Gamma}_i \tag{2.7}$$

introduced, the summed form of Eq.(2.2) ($k \to i$) can be used to obtain the average ion diffusion coefficient or pinch velocity:

$$D_i = -\frac{\hat{\Gamma}_i - C_4 V_{Pi} N_i}{C_1(N_i' - V'' N_i/V')} = -\frac{\Gamma_i - C_4 V_{Pi} n_i}{C_1 n_i'}, \tag{2.8}$$

or

$$V_{Pi} = \frac{\hat{\Gamma}_i + C_1 D_i (N_i' - V'' N_i/V')}{C_4 N_i} = \frac{\Gamma_i + C_1 D_i n_i'}{C_4 n_i}. \tag{2.9}$$

On the right side of Eq.(2.8), the average ion pinch velocity is defined

$$V_{Pi} \equiv \frac{\sum_k N_k V_{Pk}}{N_i} = \sum_k \frac{n_k}{n_i} V_{Pk}, \tag{2.10}$$

while on the right side of Eq.(2.9), the average ion diffusion coefficient is defined

$$D_i \equiv \frac{\sum_k (N_k' - V'' N_k/V') D_k}{N_i' - V'' N_i/V'} = \sum_k \frac{n_k'}{n_i'} D_k. \tag{2.11}$$

CORSICA obtains the electron density from charge neutrality, so the electron equivalent of Eq.(2.1) is not solved numerically. To obtain the electron fluxes and transport coefficients, therefore, we sum over ion species with charge-number weighting:

$$N_e = \sum_k Z_k N_k, \quad \hat{\Gamma}_e = \sum_k Z_k \hat{\Gamma}_k, \tag{2.12}$$

$$\Gamma_e \equiv \frac{\Phi_m^2}{V'} \hat{\Gamma}_e \tag{2.13}$$

Assuming a closure relation like Eq.(2.2) for electrons, we then have

$$\begin{aligned} D_e &= -\frac{\hat{\Gamma}_e - C_4 V_{Pe} N_e}{C_1(N_e' - V'' N_e / V')} = -\frac{\Gamma_e - C_4 V_{Pe} n_e}{C_1 n_e'}, & (2.14) \\ V_{Pe} &= \frac{\hat{\Gamma}_e + C_1 D_e (N_e' - V'' N_e / V')}{C_4 N_e} = \frac{\Gamma_e + C_1 D_e n_e'}{C_4 n_e}. & (2.15) \end{aligned}$$

To analyze heat transport, we start with the generic entropy equation solved in CORSICA :

$$\frac{3}{2}\Phi_m^2 \left(\frac{\Phi_m}{V'}\right)^{2/3} \left[\frac{\partial \mu_j}{\partial t} + v_{\bar{\rho}}(\mu_j - \bar{\rho}\frac{\partial \mu_j}{\partial \bar{\rho}})\right] + \frac{\partial}{\partial \bar{\rho}}[(q_j + \frac{5}{2}\Gamma_j T_j)V'] = V'(S_{Ej} + Q_j) \tag{2.16}$$

In Eq.(2.16), the subscript j can be either "e" or "i". The entropy variable μ_j is defined in Table 1.4; T_j is the temperature (all ion species are assumed to have the same temperature, which is, in general, different from that of the electrons); S_{Ej} is the physical heat source; Q_j is the electron-ion energy exchange rate ($Q_e + Q_i = 0$); and q_j is the heat flux, expressed in terms of the thermal conductivity χ_j by the closure relation

$$\begin{aligned} q_j &= -C_1 \chi_j n_j \frac{\partial T_j}{\partial \bar{\rho}} & (2.17) \\ &= -C_1 \chi_j \Phi_m \left(\frac{\Phi_m}{V'}\right)^{5/3} \left[\frac{\partial \mu_j}{\partial \bar{\rho}} - \frac{\mu_j}{N_j}\left(\frac{\partial N_j}{\partial \bar{\rho}} + \frac{2}{3} N_j \frac{V''}{V'}\right)\right] & (2.18) \end{aligned}$$

To determine χ_j we integrate Eq.(2.16) with respect to $\bar{\rho}$:

$$\begin{aligned} q_j = &-\frac{5}{2}\Gamma_j T_j + \frac{\Phi_m^2}{V'} \int_0^{\bar{\rho}} d\rho \left\{ \frac{V'}{\Phi_m^2}(S_{Ej} + Q_j) \right. \\ &\left. -\frac{3}{2}\left(\frac{\Phi_m}{V'}\right)^{2/3} \left[\frac{\partial \mu_j}{\partial t} + v_\rho(\mu_j - \rho \frac{\partial \mu_j}{\partial \rho})\right] \right\} \end{aligned} \tag{2.19}$$

Here again, only the first term in the integrand is necessary in steady state. With Γ_j given by Eq.(2.13) (electrons) or (2.7) (ions), we can compute χ_j from Eqs.(2.18), (2.19)

$$\chi_j = -\frac{q_j}{C_1 \Phi_m} \left(\frac{V'}{\Phi_m} \right)^{5/3} \left[\frac{\partial \mu_j}{\partial \bar{\rho}} - \frac{\mu_j}{N_j} \left(\frac{\partial N_j}{\partial \bar{\rho}} + \frac{2}{3} N_j \frac{V''}{V'} \right) \right]^{-1} = -\frac{q_j}{C_1 n_j T_j'}. \quad (2.20)$$

Notice that, since no "heat pinch" term appears in the closure relation (2.17), no transport-model assumptions are necessary to compute χ_j.

2.3 Numerical Determination of Transport Fluxes and Coefficients

Calculation of transport fluxes and coefficients is performed by Basis parser functions defined in the script file `analysis.bas`. All of the functions return one-dimensional arrays of length `nrho` containing the values of the relevant quantity on the `rhobar` grid. The names of the functions, together with the corresponding symbol, units and equation numbers from Section 2.2, are given in Table 2.1. Only those computing ion-species-specific quantities take an argument, *viz.*, the species-selector variable, chosen from the list in Table 1.3.

Table 2.1: Transport quantities available in analysis module

transport quantity	*units*	*analytic expression equation number*	*function name*	*function argument*
Γ_k	$10^{20}/m^3/s$	2.3	`gamma_ion_species`	`species`
Γ_i	$10^{20}/m^3/s$	2.7	`gamma_ion`	
D_k	m^2/s	2.4	`d_ion_species`	`species`
D_i	m^2/s	2.8	`d_ion`	
V_{Pk}	m/s	2.5	`v_pinch_ion_species`	`species`
V_{Pi}	m/s	2.9	`v_pinch_ion`	
q_i	MW/m^3	2.19	`q_ion`	
χ_i	m^2/s	2.20	`chi_ion`	
Γ_e	$10^{20}/m^3/s$	2.13	`gamma_elec`	
D_e	m^2/s	2.14	`d_elec`	
V_{Pe}	m/s	2.15	`v_pinch_elec`	
q_e	MW/m^3	2.19	`q_elec`	
χ_e	m^2/s	2.20	`chi_elec`	

In addition to the functions listed in Table 2.1, the transport-analysis module contains the global integer parser variable `steady_state`, whose default value is 1, indicating that the plasma is assumed to be in transport steady state. In that case, all time-dependent terms are ignored when computing transport coefficients and fluxes. If `steady_state` $\neq 1$ the arrays `nTorDot` and `muTorDot` must be defined, with the same shape as `nTor` and `muTor`, and set equal to $\partial N_k/\partial t$ and $\partial \mu_j/\partial t$, respectively, before calling any of the functions in Table 2.1. Terms proportional to $v_\rho = d \ln \Phi_m/dt$ are automatically included if `steady_state` $\neq 1$.

Finally, the transport analysis module contains the global real parser variables `diff_mult` and `pinch_mult` and the functions `initTrCoefs(iModel,iDefault)`, `set_pinch_ion_ave` and `set_diff_ion_ave`, all of which relate to the model transport coefficients appearing on the right side of Eqs. (2.4), (2.5), (2.8), (2.9), (2.14) and (2.15). The arguments of `initTrCoefs` specify the transport model to be used to calculate these coefficients, and are the same as those of the function `setTrModel` described in Section 1.5 1.5.4. The resulting coefficients are stored in the arrays `diff` and `pinch`. (Note that `diff` and `pinch` are affected by the user's choice of value of the variable `useSawtoothModel`.) The function `set_pinch_ion_ave` computes V_{Pi} from Eq.(2.10) and stores it in `pinch(,iIon)`, while `set_diff_ion_ave` computes D_i from Eq.(2.11) and stores it in `diff(,iIon)`. The variables `diff_mult` and `pinch_mult` are multipliers applied to the coefficients, giving the user a knob with which to vary their magnitudes. The default value of each is unity.

2.4 Examples

Here we give examples of use of the transport analysis module to derive transport coefficients from a set of density, temperature and source profiles associated with a self-consistent TEQ equilibrium assumed already to exist.

Example 1. Assuming the pinch velocity to be given by the RLW model, plot the deuterium diffusion coefficient with the RLW model superimposed for comparison, and a zero baseline:

```
corsica> read analysis.bas     # if not already done
corsica> initTrCoefs(iRLW,1)
corsica> nf; plot [d_ion_species(iDeut),diff(,iDeut)] rhobar
         color=rainbow
corsica> real zero(nrho); plot zero
```

The first line simply reads the script file containing the necessary functions and variables. This, of course, need be done only once. The next line causes transport coefficients to be computed using the RLW model and stored in the arrays `diff`, `pinch` and `chi`. The third line begins a new frame and plots the deuterium diffusion coefficient derived from the profile information (assuming the RLW pinch velocity), as well as the RLW model diffusion coefficient, as functions of $\bar{\rho}$ on the same grid. The last line adds a zero baseline to the plot.

Example 2. Assuming the diffusion coefficient to be given by the Chang-Hinton neoclassical model, plot the pinch velocity with the RLW model superimposed for comparison:

```
corsica> initTrCoefs(iChangHinton,1)
corsica> nf; plot v_pinch_species(iDeut) rhobar color=red
corsica> initTrCoefs(iRLW,1)
corsica> plot pinch(,iDeut) color=green
corsica> plot zero
```

Here the first line causes computation of model transport coefficients according to the Chang-Hinton neoclassical model. The second line plots the pinch velocity that follows from the profiles and model diffusion coefficient *vs.* $\bar{\rho}$. The third line recomputes the model transport coefficients using RLW, and the fourth line plots the new model pinch velocity in a contrasting color. The last line adds a baseline.

Example 3. Plot the electron and ion thermal conductivities:

```
corsica> nf;plot [chi_elec,chi_ion] rhobar color=rainbow
corsica> plot zero
```

In this example the first line plots the electron and ion thermal conductivities χ_e and χ_i derived from profiles. No reference to a transport model is necessary.

Example 4. Investigate the effect of the strength of the model diffusion coefficient on the derived pinch velocity:

```
corsica> diff_mult=1.
corsica> plot v_pinch_ion(iDeut) rhobar color=red
corsica> diff_mult=.5
corsica> plot v_pinch_ion(iDeut) color=green
```

Here the first line sets the model diffusion-coefficient multiplier to 1.0, and the next plots the pinch velocity derived from profiles, using whatever model diffusion coefficient was set by the most recent call to `initTrCoefs`. The next line resets the multiplier to 0.50, which reduces the contribution of the diffusive term to the pinch velocity. The last line plots the resulting pinch velocity in a contrasting color.

Example 5. Compare the electron and ion thermal conductivities in a time-dependent plasma:

```
corsica> steady_state=0
corsica> plot[q_elec,q_ion] rhobar color=rainbow
```

In this last example, the first line sets the steady-state variable to indicate that the plasma is not assumed to be in steady state. The second line plots the electron and ion heat fluxes with time-dependent terms included. In order for this to work the user must have appropriately defined the array `muTorDot`. Otherwise Basis will issue the error message: `Unknown variable:  muTorDot`.

Chapter 3

1-1/2 D Core Transport Equations

In this section we derive the transport equations solved by CORSICA. We start with the low-frequency limit of Maxwell's equations:

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{3.1}$$

$$\nabla \times \mathbf{B} = \mu_0 \mathbf{J} \tag{3.2}$$

$$\sum_j q_j n_j = 0 \qquad (q_j = Z_j e) \tag{3.3}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{3.4}$$

and the zeroth order moments of the Boltzman equation, including fluctuations:

$$\frac{\partial n_j}{\partial t} + \nabla \cdot (n_j \mathbf{u}_j + \mathbf{\Gamma}_{Aj}) = S_{n,j} \tag{3.5}$$

$$\frac{\partial}{\partial t}(m_j n_j \mathbf{u}_j) + \nabla \cdot (m_j n_j \mathbf{u}_j \mathbf{u}_j + p\,\mathrm{I} + \Pi) n_j q_j (\mathbf{E} + \mathbf{u}_j \times \mathbf{B}) + \mathbf{F}_j + \mathbf{S}_{\mathrm{momentum},j} \tag{3.6}$$

$$\frac{3}{2}\frac{\partial p_j}{\partial t} + \nabla \cdot (q_j + q_{A,j} + \frac{5}{2} p_j \mathbf{u}_j) = Q_j + \mathbf{u}_j \cdot (\mathbf{F}_j + q_j n_j \mathbf{E}) + S_{E,j} + S_{EA,j} \tag{3.7}$$

where the index j refers to all ion species plus electrons, and the quantities with subscript A are anomalous transport terms resulting from turbulent fluctuations. In addition, note that the viscous terms have been lumped into the source term.

3.1 Quasi-equilibrium

We are interested in modeling core transport in toroidal axisymmetric plasmas. By "core" we mean the part of the plasma characterized by closed magnetic surfaces. Given this situation it is convenient to define a coordinate system consisting of ψ, a magnetic surface label, θ, a poloidal angle variable, and φ, a toroidal angle variable.

Both θ and φ vary between 0 and 2π. All physical quantities must be periodic in these angles to ensure single-valuedness, and axisymmetry requires that all physical scalars be independent of the φ. The (ψ, θ, φ) coordinates are described in more detail in Sec. A.A.1.

The $\psi =$ constant surfaces are called magnetic flux surfaces because the magnetic field lines lie in these surfaces. As a result, we can write:

$$B^{\psi} = \mathbf{B} \cdot \nabla\psi = 0 \tag{3.8}$$

Given that the magnetic field lines lie in the flux surfaces, along with the axisymmetry assumption, one can express the magnetic field in the general form:

$$\mathbf{B} = \nabla\varphi \times \nabla\psi + F\nabla\varphi \tag{3.9}$$

where $F(\psi)$ is an arbitrary function (it is shown in Sec. A.A.2 that axisymmetry implies that F is independent of θ). It is easy to show that this expression for $\mathbf{B}$ guarantees that $\mathbf{B}$ is divergence-free.

We proceed by calculating the total momentum balance, summing Eq. **??** over all species. This gives:

$$\mathbf{E} \sum_j q_j n_j + \mathbf{J} \times \mathbf{B} + \sum_j \mathbf{F}_j = \nabla p \tag{3.10}$$

Using quasi-neutrality, and the fact that net force due to interspecies friction must be zero, we have

$$\mathbf{J} \times \mathbf{B} = \nabla p \tag{3.11}$$

This is the ideal MHD equilibrium relation. Thus, as the plasma evolves on the transport timescale it moves through a series of quasi-static MHD equilibrium.

There are several implications of Eq. 3.11. First, we see that the constant flux surfaces must also be constant pressure surfaces since

$$\mathbf{B} \cdot \nabla p = 0 \tag{3.12}$$

This implies that $p = p(\psi)$. Next we see that the current must also flow in these surfaces since

$$\mathbf{J} \cdot \nabla p = \mathbf{J} \cdot \nabla\psi \, p'(\psi) = 0 \tag{3.13}$$

which implies $J^{\psi} = 0$.

We can calculate an expression for the current by substituting our expression for the magnetic field, Eq. 3.9, into Ampere's law. The result is:

$$\mu_0 \mathbf{J} = \nabla\varphi \, \Delta^* \psi - \nabla\varphi \times \nabla\psi \, \frac{\partial F}{\partial \psi} \tag{3.14}$$

where

$$\Delta^*\psi \equiv R^2\nabla\cdot\frac{1}{R^2}\nabla\psi \tag{3.15}$$

Finally, substituting this result into Eq. 3.11 leads to the Grad-Shafranov equation for the magnetic flux function $\psi(\mathbf{R})$:[1]

$$\Delta^*\psi = -\mu_0 R^2\frac{\partial p}{\partial\psi} - F\frac{\partial F}{\partial\psi} \tag{3.16}$$

This is a quasilinear partial differential equation for ψ. Many computer codes have been written to solve this problem, with a variety of functional forms for $p(\psi)$ and $F(\psi)$, and with various types of boundary conditions. The twist here is that p and F depend parametrically on time, being evolved consistent with the flux-surface averaged transport equations that will be derived below.

3.2 Scalar Transport Equations

Next we derive a set of transport equations for the ion density, the electron and ion energy, and the magnetic flux.

3.2.1 Ion Density

The full evolution of the ion density is given by Eq. 3.5.

3.2.2 Electron Density

The electron density is found from quasi-neutrality; i.e. we can solve Eq. 3.3 for n_e to give:

$$n_e = \sum_k Z_k n_k \tag{3.17}$$

Furthermore, since $J^\psi = 0$, the radial electron particle flux must equal the weighted radial ion particle flux:

$$\Gamma_e\cdot\nabla\psi = \sum_k Z_k(n_k\mathbf{u}_k + \Gamma_{\mathrm{A}k})\cdot\nabla\psi \tag{3.18}$$

[1]See Sec. A.A.2 for details of the derivations of Eqs. 3.14 and 3.16.

3.2.3 Energy

The disparity between the electron and ion masses makes the equilibration time between electrons and ions much longer than the equilibration time amongst ion species:

$$\tau_{ee} : \tau_{ii} : \tau_{ei} \sim 1 : \frac{1}{Z^3}\sqrt{\frac{m_i}{m_e}}\left(\frac{T_i}{T_e}\right)^{3/2} : \frac{1}{2Z}\frac{m_i}{m_e} \tag{3.19}$$

Given this disparity, we are justified in assuming that

- all ion species are characterized by the same temperature T_i.
- the electrons have a different temperature T_e.

The electron-ion equilibration time τ_{ei} is still short for most fusion plasmas. However, preferential heating of one species can result in a large difference between T_e and T_i.

We now write Eq. 3.7 for the electrons and ions, with the ion equation obtained by summing the equations for the individual ion species:

$$\frac{3}{2}\frac{\partial p_e}{\partial t} + \nabla \cdot (q_e + \frac{5}{2}p_e\mathbf{u}_e) = Q_e + \mathbf{u}_e \cdot (\mathbf{F}_e - en_e\mathbf{E}) + S_{E,e} \tag{3.20}$$

and

$$\frac{3}{2}\frac{\partial p_i}{\partial t} + \nabla \cdot (q_i + \frac{5}{2}p_i\mathbf{u}_i) = S_{E,i} + \sum_k Q_k + \mathbf{u}_k \cdot (\mathbf{F}_k + q_k n_k\mathbf{E}) \tag{3.21}$$

where

$$x_i = \sum_k x_k, \quad x = p, q, S_E \tag{3.22}$$

$$p_i\mathbf{u}_i = \sum_k p_k\mathbf{u}_k \tag{3.23}$$

and where the anomalous contributions to q_i and S_E have been lumped in with the non-anomalous parts. Note that Eq. 3.23 is an implicit definition of $\mathbf{u}_i$. This is consistent with $\mathbf{u}_i$ being the total ion particle flux since all ion species share the same temperature.

Conservation of energy and momentum by collisional processes leads to the following relationship between the collisional heating source and the collisional friction term:

$$\sum_j Q_j + \mathbf{u}_j \cdot \mathbf{F}_j = 0 \tag{3.24}$$

Chapter 4

Hyper-resistivity

4.1 Background

In this section, we discuss the additions to the Ohm's Law equation introduced by hyper-resistive effects. We follow the formulation of A.H. Boozer [13] and its development for toroidal geometry by D.J. Ward and S.C. Jardin [14].

The Ohm's law equation including hyper-resistivity is written as:

$$\mathbf{B} \cdot \mathbf{E} = \eta\, \mathbf{J} \cdot \mathbf{B} - \nabla \cdot \Lambda \nabla (\frac{\mathbf{J} \cdot \mathbf{B}}{B^2}) \tag{4.1}$$

where we define the hyper-resistive coefficient as

$$\Lambda = \eta B^2 \frac{D}{\nu} \tag{4.2}$$

Here, ν is the electron-ion collision frequency, D is a diffusion coefficient, and η is the resistivity.

Several examples of the hyper-resistive coefficient have been implemented in the Corsica code for calculations applied to the Sustained Spheromak Experiment (SSPX) at the Lawrence Livermore National Laboratory and the Madison Symmetric Torus (MST), a toroidal reversed-field pinch device at the University of Wisconsin - Madison. Included among the hyper-resistive coefficient options is the formulation of H.L. Berk, et al. [15]. The actual forms of the coefficient will be described in more detail below.

4.2 Corsica Formulation of the General Transport Equation

The transport equations in Corsica in general are cast in the form of a parabolic partial differential equation (PDE) for numerical solution. The general form of the equation is:

$$\begin{aligned} 0 &= H_1 \frac{\partial U}{\partial t} - \frac{\partial}{\partial x}[H_2 \frac{\partial U}{\partial x} + H_3 U + H_4] - H_5 \frac{\partial U}{\partial x} - H_6\, U \\ & -H_7 - H_9 \frac{\partial U(x_e)}{\partial x} - H_{10}\, U(x_e) \end{aligned} \tag{4.3}$$

on the interval from 0 to x_e subject to the initial condition

$$U(x, 0) = U_0(x) \tag{4.4}$$

and the boundary conditions

$$H_2(x) \frac{\partial U(0,t)}{\partial x} = 0 \tag{4.5}$$

$$\alpha \frac{\partial U(x_e,t)}{\partial x} + \beta\, U(x_e, t) = \gamma \tag{4.6}$$

If $H_2(x)$ scales as x near the origin, then $\frac{\partial U(0,t)}{\partial x}$ has some finite value and $\frac{\partial U(0,t)}{\partial r} = 0$, where $x = r^2$. If the H_i coefficients depend on U or its derivatives with respect to x, then Eq. 4.3 must be iterated to obtain implicit solutions. H_8 is supplied as an aid for implementing a drag term. Eq. 4.3 is solved by a Galerkin method with a choice of cubic or linear finite elements.

4.2.1 Corsica's Ohm's Law Equation with Poloidal Flux as the Independent Variable

In the case when the independent variable in the transport equations is the poloidal flux, the independent x variable in the Corsica is $\bar{\rho} \equiv \Psi/\Psi_m$ in MKS units of Webers and the dependent variable U is $y \equiv \Psi_m\, q/(2\pi)$. (Note that the equilbrium calculations in Corsica use a variable, ψ, which is related to the toroidal flux by $\Psi = 2\pi\psi$.) The H coefficients in the case of no hyper-resistivity are $H_1 = 1$, H_5 ... $H_{10} = 0$, and

$$H_2 = \frac{\eta\, C_{23}}{\mu_0\, C_3^2} \tag{4.7}$$

$$H_3 = -\frac{\eta\, \frac{\partial C_{23}}{\partial \bar{\rho}}}{\mu_0\, C_3^2} + \frac{V_{\text{loop}}}{q\, \Psi_m} + \frac{\bar{\rho}}{\Psi_m} \frac{\partial \Psi_m}{\partial t} \tag{4.8}$$

$$H_4 = -\sigma_T\, \eta \frac{\langle \mathbf{J}_{ni} \cdot \mathbf{B} \rangle}{\langle \frac{B_{Tor}}{R} \rangle} \tag{4.9}$$

where -σ_T is the sign of the toroidal plasma current, and the loop-voltage is given by

$$V_{\text{loop}} = 2\pi\, \eta \left[\frac{\langle \mathbf{J}_{total} \cdot \mathbf{B} \rangle}{\langle \frac{B_{Tor}}{R} \rangle} - \frac{\langle \mathbf{J}_{ni} \cdot \mathbf{B} \rangle}{\langle \frac{B_{Tor}}{R} \rangle} \right] \tag{4.10}$$

and the coefficients C_2, C_3, and C_{23} are defined in the table in section 4.5 below. The code variables in terms of the physical variables are summarized in section 4.5 and the $\langle ... \rangle$ notation denotes the flux surface average.

4.2.2 Corsica's Ohm's Law Equation with Toroidal Flux as the Independent Variable

The case where the toroidal flux, Φ, is the independent variable is functionally similar to the previous case where the poloidal flux is the dependent variable. The only difference is that the independent x variable is now $\bar{\rho} \equiv \Phi/\Phi_m$, and the dependent variable U is $y \equiv \Phi_m/(2\pi q)$. Here we also have

$$H_3 = \frac{\eta\,\frac{\partial C_{23}}{\partial \bar{\rho}}}{\mu_0\, C_3^2} + \frac{\bar{\rho}}{\Phi_m}\frac{\partial \Phi_m}{\partial t} \tag{4.11}$$

This differs from the expression in equation 4.8 in that the loop voltage term is not present and the Ψ_m term is written in terms of Φ_m. For the H_4 coefficient, the sign is different for the toroidal flux case and is now:

$$H_4 = \sigma_T\,\eta\frac{\langle \mathbf{J}_{ni} \cdot \mathbf{B}\rangle}{\left\langle \frac{B_{Tor}}{R}\right\rangle} \tag{4.12}$$

4.3 Ohm's Law with Hyper-Resistivity Contributions

When hyper-resitive effects are included, the second-order PDE form of Ohm's Law becomes a fourth-order equation (Eq. 11 in Ward and Jardin [14]). However, solving this fourth order equation is equivalent to solving an appropriately defined system of two second order PDE's. The system of equations can be represented as a matrix equation

$$\begin{pmatrix} M_1 & M_3 \\ M_4 & M_2 \end{pmatrix} \cdot \begin{pmatrix} U_1 \\ U_2 \end{pmatrix} = 0 \tag{4.13}$$

4.3.1 Poloidal Flux as the Independent Variable

The U_1 variable is the same as U in section 4.2.1. The U_2 variable is the hyper-resistivity variable, h_r

$$h_r \equiv \frac{2\pi}{q\Psi_m}\frac{\partial}{\partial \bar{\rho}}\left[\frac{\langle \mathbf{J}_{total} \cdot \mathbf{B}\rangle}{\langle B^2\rangle}\right] \tag{4.14}$$

The Ohm's Law equation without hyper-resitivity is the equation produced by this matrix form when all the H_{ij} coefficients in matrix M_j for j=2,3,4 are zero, and we are left with the equation

$$M_1 \cdot U_1 = 0 \tag{4.15}$$

which is the same equation as Eq. 4.3.

The coefficients H_{i1} of the matrix M_1 are the coefficients H_i from section 4.2.1 above where $i = 1...10$. The appropriate definitions of the H_{ij} coefficients in the other matrix blocks (j=2,3,4) produce the appropriate equation set. We see that for M_3

$$H_{23} = \Lambda \tag{4.16}$$

$$H_{33} = \frac{\partial \Lambda}{\partial \bar{\rho}} \tag{4.17}$$

where Λ is the hyper-resistive coefficient of Eq. 4.2 and the remaining $H_{i3} = 0$.

For matrix block M_4, we have

$$H_{24} = \frac{C_{23}}{f_{1HR}\, C_3^2} \tag{4.18}$$

$$H_{34} = -\frac{\frac{\partial C_{23}}{\partial \bar{\rho}}}{f_{1HR}\, C_3^2} \tag{4.19}$$

and the remaining coefficients $H_{i4} = 0$. The variable f_{1HR} is defined as

$$f_{1HR} = \frac{\mu_0}{(2\pi)^2} \left\langle B^2 \right\rangle \frac{\partial V}{\partial \bar{\rho}} \tag{4.20}$$

where V is the plasma volume. For matrix block M_2, we have $H_{62} = 1$ and the remaining coefficients $H_{i2} = 0$.

In addition, when hyper-resistivity is included, the loop-voltage expression in equation 4.10 contains a term due to hyper-resistivity and is:

$$V_{\text{loop}} = 2\pi\, \eta \left[\frac{\langle \mathbf{J}_{total} \cdot \mathbf{B} \rangle}{\left\langle \frac{B_{Tor}}{R} \right\rangle} - \frac{\langle \mathbf{J}_{ni} \cdot \mathbf{B} \rangle}{\left\langle \frac{B_{Tor}}{R} \right\rangle} \right] - 2\pi \left[\Lambda \frac{\partial h_r}{\partial \bar{\rho}} + \frac{\partial \Lambda}{\partial \bar{\rho}}\, h_r \right] \tag{4.21}$$

where recall that h_r is defined in Eq. 4.14.

Including hyper-resistive effects, the matrix Eq. 4.13 becomes

$$M_1 \cdot U_1 + M_3 \cdot U_2 = 0 \tag{4.22}$$

$$M_4 \cdot U_1 + M_2 \cdot U_2 = 0 \tag{4.23}$$

The first of these equations is the new Ohm's Law. The equations in terms of the physical variables follow. The Ohm's Law becomes

$$\begin{aligned} 0 &= \frac{\partial y}{\partial t} - \frac{\partial}{\partial \bar{\rho}} \Bigg[C_{23} \frac{\eta}{\mu_0\, C_3^2} \frac{\partial y}{\partial \bar{\rho}} + y \Bigg\{ -\frac{\eta}{\mu_0\, C_3^2} \frac{\partial C_{23}}{\partial \bar{\rho}} \\ &+ \frac{2\,\pi}{\Psi_m\, q} \left(\eta \left\{ \frac{\langle \mathbf{J}_{total} \cdot \mathbf{B} \rangle}{\left\langle \frac{B_{Tor}}{R} \right\rangle} - \frac{\langle \mathbf{J}_{ni} \cdot \mathbf{B} \rangle}{\left\langle \frac{B_{Tor}}{R} \right\rangle} \right\} \underline{- \left\{ \Lambda \frac{\partial h_r}{\partial \bar{\rho}} + h_r \frac{\partial \Lambda}{\partial \bar{\rho}} \right\}} \right) + \frac{\bar{\rho}}{\Psi_m} \frac{\partial \Psi_m}{\partial t} \Bigg\} \\ &+ \eta \frac{\langle \mathbf{J}_{ni} \cdot \mathbf{B} \rangle}{\left\langle \frac{B_{Tor}}{R} \right\rangle} \underline{- \left\{ \Lambda \frac{\partial h_r}{\partial \bar{\rho}} + h_r \frac{\partial \Lambda}{\partial \bar{\rho}} \right\}} \Bigg] \end{aligned} \tag{4.24}$$

where the hyper-resiStive contributions are the underlined terms in addition to the second equation shown below.

$$h_r = -\frac{(2\pi)^2}{\mu_0}\frac{\partial}{\partial\bar\rho}\left[\frac{C_{23}\frac{\partial y}{\partial\bar\rho}}{C_3^2\,\langle B^2\rangle\frac{\partial V}{\partial\bar\rho}} - \frac{y\,\frac{\partial C_{23}}{\partial\bar\rho}}{\mu_0\,C_3^2\,\langle B^2\rangle\frac{\partial V}{\partial\bar\rho}}\right] \tag{4.25}$$

4.3.2 Toroidal Flux as the Independent Variable

Again, the variable $\bar\rho \equiv \Phi/\Phi_m$, the dependent variable U_1 is the same as the variable U in section 4.2.2, and the h_r variable is defined as

$$h_r \equiv \frac{2\pi}{\Phi_m}\frac{\partial}{\partial\bar\rho}\left[\frac{\langle \mathbf{J}_{total}\cdot\mathbf{B}\rangle}{\langle B^2\rangle}\right] \tag{4.26}$$

As before, the coefficients H_{1j} for the matrix element M_1 are defined as the H_j coefficients from section 4.2.2. The coefficients H_{2j} and H_{3j} for the other matrix blocks M_2 and M_3 are the same as for the poloidal flux independent variable case in section 4.3.1.

The H_{34} coefficient for matrix block M_4 is slightly different, while $H_{24} = H_{14} = 0$ again and H_{24} is the same as in section 4.3.1. H_{34} is now:

$$H_{34} = \frac{\frac{\partial C_{23}}{\partial\bar\rho}}{f_{1HR}\,C_3^2} \tag{4.27}$$

The new Ohm's Law with hyper-resistive effects where toroidal flux is the independent variable is then

$$\begin{aligned} 0 \quad &= \frac{\partial y}{\partial t} - \frac{\partial}{\partial\bar\rho}\left[C_{23}\frac{\eta}{\mu_0\,C_3^2}\frac{\partial y}{\partial\bar\rho} + y\left\{\frac{\eta}{\mu_0\,C_3^2}\frac{\partial C_{23}}{\partial\bar\rho} + \frac{\bar\rho}{\Phi_m}\frac{\partial\Phi_m}{\partial t}\right\}\right. \\ &\left. +\eta\,\frac{\langle\mathbf{J}_{ni}\cdot\mathbf{B}\rangle}{\langle\frac{B_{Tor}}{R}\rangle} \underline{- \left\{\Lambda\frac{\partial h_r}{\partial\bar\rho} + h_r\frac{\partial\Lambda}{\partial\bar\rho}\right\}}\right] \end{aligned} \tag{4.28}$$

and the second equation becomes

$$h_r = -\frac{(2\pi)^2}{\mu_0}\frac{\partial}{\partial\bar\rho}\left[\frac{C_{23}\frac{\partial y}{\partial\bar\rho}}{C_3^2\,\langle B^2\rangle\frac{\partial V}{\partial\bar\rho}} + \frac{y\,\frac{\partial C_{23}}{\partial\bar\rho}}{\mu_0\,C_3^2\,\langle B^2\rangle\frac{\partial V}{\partial\bar\rho}}\right] \tag{4.29}$$

4.4 Calculation of the Hyper-Resistive Coefficient

In Corsica, the hyper-resistive coefficient, Λ, is modelled in several ways. For the SSPX calculations, a simple model is used where

$$\Lambda = 4\pi\times10^{-7}\,\bar\rho \tag{4.30}$$

4.4.1 Berk Model

In Corsica, the model for the hyper-resistive coefficient in Berk et al. [15] uses the definition for Λ from equation 4.2. We take $\eta/\nu = 1/(\epsilon_0\, \omega_{pe^2})$, where ω_{pe} is the electron plasma frequency and $\epsilon_0 = 8.8542 \times 10^{-12}$ is the permittivity of free space. The diffusion coefficient D can be evaluated in several regimes, the long mean-free-path limit (D_A), the short mean-free-path limit (D_B), and the thin-island limit (D_C). The long mean-free-path limit in this case means that the island transit rate is much larger than the electron collision frequency. For our calculations, we use the effective D constructed as:

$$\frac{1}{D} = \frac{1}{D_A} + \frac{1}{D_B} + \frac{1}{D_C} \tag{4.31}$$

For all Corsica calculations, the physical parameters are calculated in MKS units except the temperature, which is in keV.

Long Mean-Free-Path Limit

At each resonant surface, a contribution to D is calculated and the total hyper-resistive coefficient is the sum of the effects from each resonant surface. The contribution to D_A from the resonant surface at the minor radius r_{isl} is

$$D_A(r) = \Delta_{isl}^2\, \omega_{tr} \exp\left[-\frac{(r - r_{isl})^2}{2(\Delta_{isl}^2 + w_m^2)}\right] \tag{4.32}$$

where r is the minor radius, Δ_{isl} is the island width, and ω_{tr} is the transit rate for electrons moving around an island. w_m is some small number introduced into the calculation to ensure that the denominator of the exponent does not go to zero.

The transit rate, ω_{tr} is given by

$$\omega_{tr} = v_{te} \left(\frac{2sm}{qRr}\frac{B_\Phi}{B}\frac{\delta B}{B}\right)^{\frac{1}{2}} \tag{4.33}$$

where R is the major radius, m is the resonance number, B is the total magnetic field, B_Φ is the toroidal magnetic field, v_{te} is the electron thermal velocity in m/s given by

$$v_{te} = 4.19 \times 10^5 \sqrt{1000 T_e} \tag{4.34}$$

where T_e is in keV, and the magnetic shear parameter is given by

$$s = \frac{2\bar{\rho}}{q}\frac{\partial q}{\partial \bar{\rho}} \tag{4.35}$$

The model for D is applied using the following equation relating the fractional change in $\delta B/B$ to the island width, Δ_{isl},

$$\Delta_{isl} = \left(\frac{2qRr}{sm}\frac{B}{B_\Phi}\frac{\delta B}{B}\right)^{\frac{1}{2}} \tag{4.36}$$

We have the option of specifying a value for $\delta B/B$ (typically, .01) and using Eq. 4.36 to calculate the island width, Δ_{isl} or of solving the Rutherford island width equations for Δ_{isl} and then calculating $\delta B/B$ using Eq. 4.36.

Short Mean-Free-Path Limit

In the short mean-free-path limit, the diffusion coefficient is calculated from the expression

$$D_B(r) = \frac{\Delta_{isl}^4}{\nu_{\parallel}} \left(\frac{v_{te} s m B_\Phi}{q r R B} \right)^2 \exp\left[-\frac{(r - r_{isl})^2}{2(\Delta_{isl}^2 + w_m^2)} \right] \tag{4.37}$$

We calculate $\nu_{\parallel}$ from the expression for the electron 90° collision rate

$$\nu_{\perp} = 2.9 \times 10^{-6} \, \ln \Lambda_{ee} \left(\frac{n_e 10^{14}}{(T_e 10^3)^{\frac{3}{2}}} \right) \tag{4.38}$$

and

$$\nu_{\parallel} = \nu_{\perp}/1.96 \tag{4.39}$$

where the electron density, n_e, is in units of $10^{20}/m^3$ and the electron temperature, T_e is in keV. The Coulomb logarithm, is

$$\ln \Lambda_{ee} = 24. - \ln \frac{\sqrt{n_e 10^{14}}}{1000 T_e} \tag{4.40}$$

Thin-Island Limit

The thin island limit considers the case where the particles can diffuse across the island structure in a time that is shorter than the transit time around the island. In this case, the diffusion coefficient is calculated as

$$D_C(r) = \frac{\Delta_{isl}^4}{D_{\perp}^{\frac{1}{3}}} \left[\frac{B_\Phi}{B} \frac{s m v_{te}}{q r R} \right]^{\frac{4}{3}} \exp\left[-\frac{(r - r_{isl})^2}{2\left((\Delta r)^2 + w_m^2\right)} \right] \tag{4.41}$$

where the perpendicular diffusion coefficient, $D_{\perp}$ is taken to be the typical value of $1\ m^2/s$ for these calculations, and

$$(\Delta r)^2 = \left[\frac{B}{B_\Phi} \frac{r R}{m s} \frac{D_{\perp}}{v_{te}} \right]^{\frac{2}{3}} \tag{4.42}$$

4.5 Code Variables and Their Physical Values

Code Variable	Physical Value	Units	Comments[1]
`yTor`	$y = \Phi_m/(2\pi\ q)$	Wb	when fluxCord=0
`yTor`	$y = \Psi_m\ q/(2\pi)$	Wb	when fluxCord=1 or 2
`hrTor`	$h_r = \frac{2\pi}{\Phi_m}\frac{\partial}{\partial\bar{\rho}}\left[\frac{\langle \mathbf{J}_{total}\cdot\mathbf{B}\rangle}{\langle B^2\rangle}\right]$	A/Wb^2	when fluxcord = 0
`hrTor`	$h_r = \frac{2\pi}{q\Psi_m}\frac{\partial}{\partial\bar{\rho}}\left[\frac{\langle \mathbf{J}_{total}\cdot\mathbf{B}\rangle}{\langle B^2\rangle}\right]$	A/Wb^2	when fluxcord = 1 or 2
`rhobar`	$\bar{\rho} = \Phi/\Phi_m$		when fluxCord=0
`rhobar`	$\bar{\rho} = \Psi/\Psi_m$		when fluxCord=1 or 2
`torf`	Φ_m	Wb	when fluxCord=0
`torf`	Ψ_m	Wb	when fluxCord=1 or 2
`torfDot`	$\frac{1}{\Phi_m}\frac{d\Phi_m}{dt}$	1/s	when fluxCord=0
`torfDot`	$\frac{1}{\Psi_m}\frac{d\Psi_m}{dt}$	1/s	when fluxCord=1 or 2
`vprTor`	$V' = \frac{\partial V}{\partial\bar{\rho}} = \frac{\Phi_m}{q}\oint\frac{d\ell}{B}$	m^3	when fluxCord=0
`c1`	$C_1 = \oint\frac{d\ell}{B}\frac{1}{q^2}\lvert\nabla\Phi\rvert^2\frac{1}{V'\Phi_m}$	1/m^2	when fluxCord=0
`c1`	$C_1 = \oint\frac{d\ell}{B}\frac{1}{q^2}\lvert\nabla\Phi\rvert^2\frac{1}{V'\Psi_m q}$	1/m^2	when fluxCord=1 or 2
`c2`	$C_2 = \oint B\cdot d\ell\frac{q}{\Phi_m}$	1/m	when fluxCord=0
c2	$C_2 = \oint B\cdot d\ell\frac{1}{\Psi_m}$	1/m	when fluxCord=1 or 2
`c3`	$C_3 = \Phi_m/(2\pi F)$	m	when fluxCord=0
`c3`	$C_3 = \Psi_m q/(2\pi F)$	m	when fluxCord=1 or 2
`c23`	$C_{23} = C_2\ C_3$		
`c4`	$C_4 = \oint\frac{d\ell}{B}\frac{1}{q^2}\lvert\nabla\Phi\rvert^2\frac{1}{V'}$	1/m	
`qTor`	q		
`sigTor`	$1/\eta$	mho/m	
`jparTor`	$\frac{\langle\mathbf{J}\cdot\mathbf{B}\rangle}{\langle B^2\rangle}$	A/m^2/T	
`jniTor`	$\frac{\langle\mathbf{J}_{ni}\cdot\mathbf{B}\rangle}{\left\langle\frac{B_{Tor}}{R}\right\rangle}$	A/m	all non-inductive sources
`jniTor`	$\frac{\langle\mathbf{J}_{ni}\cdot\mathbf{B}\rangle}{\left\langle\frac{B_{Tor}}{R}\right\rangle}\ q$	A/m	all non-inductive sources; fluxcord=2
`jtTor`	$\frac{\langle\mathbf{J}_{total}\cdot\mathbf{B}\rangle}{\left\langle\frac{B_{Tor}}{R}\right\rangle}$	A/m	all sources
`jtTor`	$\frac{\langle\mathbf{J}_{total}\cdot\mathbf{B}\rangle}{\left\langle\frac{B_{Tor}}{R}\right\rangle}\ q$	A/m	all sources; fluxcord=2

[1]fluxCord=0 : $\bar{\rho} = \Phi/\Phi_m$; fluxCord=1 or 2 : $\bar{\rho} = \Psi/\Psi_m$

Code Variable	Physical Value	Units	Comments
`loopVoltage`	$V_{loop} = 2\pi\eta \left[\frac{\langle \mathbf{J}_{total} \cdot \mathbf{B} \rangle}{\left\langle \frac{B_{Tor}}{R} \right\rangle} - \frac{\langle \mathbf{J}_{ni} \cdot \mathbf{B} \rangle}{\left\langle \frac{B_{Tor}}{R} \right\rangle} \right]$	V	when fluxCoord=0 or 1
`loopVoltage`	$V_{loop} = 2\pi\eta q \left[\frac{\langle \mathbf{J}_{total} \cdot \mathbf{B} \rangle}{\left\langle \frac{B_{Tor}}{R} \right\rangle} - \frac{\langle \mathbf{J}_{ni} \cdot \mathbf{B} \rangle}{\left\langle \frac{B_{Tor}}{R} \right\rangle} \right]$	V	when fluxCoord=2

Appendix A

Coordinate systems, derivations, etc.

A.1 Coordinate systems

We work primarily in the flux coordinate system (ψ, θ, φ). Here ψ is a flux surface label, θ is a poloidal angle variable, and φ is a toroidal angle variable. These variables form a right handed coordinate system, as shown in Fig. **??**.

In general, this system is not orthogonal. In particular, $\nabla\theta \cdot \nabla\psi$ is not generally zero, even in an axisymmetric system. Thus we must deal with covariant and contravariant representations of vectors. For convenience in writing various identities we define:

$$u^1(\mathbf{R}) \equiv \psi(\mathbf{R}) \tag{A.1}$$
$$u^2(\mathbf{R}) \equiv \theta(\mathbf{R}) \tag{A.2}$$
$$u^3(\mathbf{R}) \equiv \varphi(\mathbf{R}) \tag{A.3}$$

The tangent or covariant basis vectors are

$$\mathbf{e}_1 \equiv \frac{\partial \mathbf{R}}{\partial u^1} = \frac{\partial \mathbf{R}}{\partial \psi} \tag{A.4}$$
$$\mathbf{e}_2 \equiv \frac{\partial \mathbf{R}}{\partial u^2} = \frac{\partial \mathbf{R}}{\partial \theta} \tag{A.5}$$
$$\mathbf{e}_3 \equiv \frac{\partial \mathbf{R}}{\partial u^2} = \frac{\partial \mathbf{R}}{\partial \varphi} \tag{A.6}$$

and the reciprocal or contravariant basis vectors are

$$\mathbf{e}^1 \equiv \nabla u^1 = \nabla\psi \tag{A.7}$$
$$\mathbf{e}^2 \equiv \nabla u^2 = \nabla\theta \tag{A.8}$$
$$\mathbf{e}^3 \equiv \nabla u^3 = \nabla\varphi \tag{A.9}$$

The Jacobian for the transformation is

$$\mathcal{J} \equiv \frac{1}{\nabla\psi \cdot \nabla\theta \times \nabla\varphi} \tag{A.10}$$

$$= \frac{\partial \mathbf{R}}{\partial u^1} \cdot \frac{\partial \mathbf{R}}{\partial u^2} \times \frac{\partial \mathbf{R}}{\partial u^3} \tag{A.11}$$

$$= \mathbf{e}_1 \cdot \mathbf{e}_2 \times \mathbf{e}_3 \tag{A.12}$$

giving a volume element

$$d^3\mathbf{x} = \mathcal{J}\, d\psi\, d\theta\, d\varphi \tag{A.13}$$

The contravariant and covariant basis vectors are related by

$$\mathbf{e}^1 = \frac{\mathbf{e}_2 \times \mathbf{e}_3}{\mathbf{e}_1 \cdot \mathbf{e}_2 \times \mathbf{e}_3} \tag{A.14}$$

$$= \frac{1}{\mathcal{J}}\, \mathbf{e}_2 \times \mathbf{e}_3 \tag{A.15}$$

and similarly for $\{i, j, k\} = \{2, 3, 1\}$ and $\{3, 1, 2\}$.

Now suppose that our coordinate system is time-dependent. In this case time enters all of the above equations as a parameter. Then the Jacobian is, in general, also time dependent, and we have

$$\frac{\partial \mathcal{J}}{\partial t} = \frac{\partial}{\partial t}\left(\frac{\partial \mathbf{R}}{\partial u^1} \cdot \frac{\partial \mathbf{R}}{\partial u^2} \times \frac{\partial \mathbf{R}}{\partial u^3}\right) \tag{A.16}$$

$$= \frac{\partial}{\partial u^1}\left(\frac{\partial \mathbf{R}}{\partial t}\right) \cdot \frac{\partial \mathbf{R}}{\partial u^2} \times \frac{\partial \mathbf{R}}{\partial u^3} + \tag{A.17}$$

$$\frac{\partial}{\partial u^2}\left(\frac{\partial \mathbf{R}}{\partial t}\right) \cdot \frac{\partial \mathbf{R}}{\partial u^3} \times \frac{\partial \mathbf{R}}{\partial u^1} + \tag{A.18}$$

$$\frac{\partial}{\partial u^3}\left(\frac{\partial \mathbf{R}}{\partial t}\right) \cdot \frac{\partial \mathbf{R}}{\partial u^1} \times \frac{\partial \mathbf{R}}{\partial u^2} \tag{A.19}$$

We define the velocity of the coordinate system, $\mathbf{u}_c$, as

$$\mathbf{u}_c = \frac{\partial \mathbf{R}}{\partial t} \tag{A.20}$$

Using Eqs. A.20 and A.14 in Eq. A.16 we have:

$$\frac{\partial \mathcal{J}}{\partial t} = \frac{\partial \mathbf{u}_c}{\partial u^i} \cdot \mathcal{J}\mathbf{e}^i \tag{A.21}$$

but $\mathbf{e}^i \frac{\partial}{\partial u^i}$ is just the representation of the ∇ operator in the covariant basis. Thus we have the desired result:

$$\frac{\partial \mathcal{J}}{\partial t} = \mathcal{J}\nabla \cdot \mathbf{u}_c \tag{A.22}$$

Finally, due to axisymmetry, we have the following identities:

$$\nabla\varphi = \frac{1}{R}\,\hat{\mathbf{e}}_\varphi \tag{A.23}$$

where $\hat{\mathbf{e}}_\varphi$ is a unit vector in the toroidal direction, and

$$\nabla\varphi \cdot \nabla\psi = 0 \tag{A.24}$$
$$\nabla\varphi \cdot \nabla\theta = 0 \tag{A.25}$$

A.2 The Grad-Shafranov Equation

In this section we derive the expression for the current, $\mathbf{J}$, in our flux coordinate system, and then derive the Grad-Shafranov equation for the magnetic flux function ψ. First we write:

$$\mathbf{B} = \mathbf{B}_\mathrm{T} + \mathbf{B}_\mathrm{p} \tag{A.26}$$

where

$$\mathbf{B}_\mathrm{T} \equiv F\nabla\psi \tag{A.27}$$

and

$$\mathbf{B}_\mathrm{p} \equiv \nabla\phi \times \nabla\psi \tag{A.28}$$

Then we can write:

$$\nabla \times \mathbf{B}_\mathrm{T} = \frac{1}{\mathcal{J}}(\mathbf{e}_1\frac{\partial}{\partial\theta}\mathbf{B}_\mathrm{T}\cdot\mathbf{e}_3 - \mathbf{e}_2\frac{\partial}{\partial\psi}\mathbf{B}_\mathrm{T}\cdot\mathbf{e}_3) \tag{A.29}$$
$$= \frac{1}{\mathcal{J}}\mathbf{e}_1\frac{\partial F}{\partial\theta} - \frac{1}{\mathcal{J}}\mathbf{e}_2\frac{\partial F}{\partial\psi} \tag{A.30}$$
$$= \nabla\theta\times\nabla\varphi\frac{\partial F}{\partial\theta} - \nabla\varphi\times\nabla\psi\frac{\partial F}{\partial\psi} \tag{A.31}$$

and

$$\nabla \times \mathbf{B}_\mathrm{p} = \frac{1}{\mathcal{J}}\mathbf{e}_3(\frac{\partial}{\partial\psi}\mathbf{B}_\mathrm{p}\cdot\mathbf{e}_2 - \frac{\partial}{\partial\theta}\mathbf{B}_\mathrm{p}\cdot\mathbf{e}_1) \tag{A.32}$$

We need the covariant components of $\mathbf{B}_\mathrm{p}$. These are:

$$\mathbf{B}_\mathrm{p}\cdot\mathbf{e}_1 = (\nabla\varphi\times\nabla\psi)\cdot(\mathcal{J}\nabla\theta\times\nabla\varphi) \tag{A.33}$$
$$= \mathcal{J}(\nabla\varphi\cdot\nabla\theta)(\nabla\psi\cdot\nabla\varphi) - \mathcal{J}|\nabla\varphi|^2\nabla\psi\cdot\nabla\theta \tag{A.34}$$
$$= -\mathcal{J}\frac{\nabla\psi}{R^2}\cdot\nabla\theta \tag{A.35}$$

and

$$\mathbf{B}_{\rm p}\cdot\mathbf{e}_2 = (\nabla\varphi\times\nabla\psi)\cdot(\mathcal{J}\nabla\varphi\times\nabla\psi) \tag{A.36}$$
$$= \mathcal{J}|\nabla\varphi|^2(\nabla\psi\cdot\nabla\psi) - \mathcal{J}(\nabla\varphi\cdot\nabla\psi)^2 \tag{A.37}$$
$$= \mathcal{J}\frac{\nabla\psi}{R^2}\cdot\nabla\psi \tag{A.38}$$

and thus

$$\nabla\times\mathbf{B}_{\rm p} = \mathbf{e}_3\frac{1}{\mathcal{J}}(\frac{\partial}{\partial\psi}\mathcal{J}\frac{\nabla\psi}{R^2}\cdot\nabla\psi + \frac{\partial}{\partial\theta}\mathcal{J}\frac{\nabla\psi}{R^2}\cdot\nabla\theta) \tag{A.39}$$
$$= \mathbf{e}_3\nabla\cdot\frac{1}{R^2}\nabla\psi \tag{A.40}$$
$$= \nabla\varphi R^2\nabla\cdot\frac{1}{R^2}\nabla\psi \tag{A.41}$$

and finally:

$$\mu_0\,\mathbf{J} = \nabla\varphi R^2\nabla\cdot\frac{1}{R^2}\nabla\psi + \nabla\theta\times\nabla\varphi\frac{\partial F}{\partial\theta} - \nabla\varphi\times\nabla\psi\frac{\partial F}{\partial\psi} \tag{A.42}$$

First, we see that

$$\mu_0\,\mathbf{J}\cdot\nabla\psi = \frac{1}{\mathcal{J}}\frac{\partial F}{\partial\theta} \tag{A.43}$$

but we know that $\mathbf{J}$ lies in the flux surface, and thus we have

$$\frac{\partial F}{\partial\theta} = 0 \tag{A.44}$$

and

$$\mu_0\,\mathbf{J} = \nabla\varphi R^2\nabla\cdot\frac{1}{R^2}\nabla\psi - \nabla\varphi\times\nabla\psi\frac{\partial F}{\partial\psi}$$
$$= \nabla\varphi\Delta^*\psi - \nabla\varphi\times\nabla\psi\frac{\partial F}{\partial\psi} \tag{A.45}$$

where

$$\Delta^*\psi \equiv R^2\nabla\cdot\frac{1}{R^2}\nabla\psi \tag{A.46}$$

Next we substitute Eq. A.45 into the MHD force balance equation, Eq. 3.11. This gives:

$$\mu_0\,\mathbf{J}\times\mathbf{B} \;=\; (\nabla\varphi\,\Delta^*\psi - \nabla\varphi\times\nabla\psi\,\frac{\partial F}{\partial\psi})\times(\nabla\varphi\times\nabla\psi + F\nabla\varphi) \tag{A.47}$$
$$= \nabla\varphi\times(\nabla\varphi\times\nabla\psi)\,\Delta^*\psi + -F\frac{\partial F}{\partial\psi}(\nabla\varphi\times\nabla\psi)\times\nabla\varphi \tag{A.48}$$
$$= -\frac{1}{R^2}\,\nabla\psi\,(\Delta^*\psi + F\frac{\partial F}{\partial\psi}) \tag{A.49}$$

and so

$$-\frac{1}{R^2}\,\nabla\psi\,(\Delta^*\psi + F\frac{\partial F}{\partial\psi}) = \mu_0\frac{\partial p}{\partial\psi}\nabla\psi \tag{A.50}$$

or

$$\Delta^*\psi = -\mu_0 R^2\frac{\partial p}{\partial\psi} - F\frac{\partial F}{\partial\psi}, \tag{A.51}$$

which is the desired result.

Appendix B

Developer information

This appendix documents several features of the Corsica development environment that are more-or-less unique to Corsica. Note that the `README` file at the top of the Corsica source tree contains a summary of how to build the code, including environment setting for specific platforms. Please try to keep this up-to-date.

B.1 The CVS repository

The Corsica source and documentation are maintained under the cvs system on LLNL's MFE cluster of workstations. For documentation on cvs see the cvs man page and Per Cederqvist's "Version Management with CVS" (a.k.a `cvs.texinfo`). The cvs executable is in `/usr/local/bin` on most machines. Note that cvs has good online "usage" information: type "`cvs --help`" for a list of options, "`cvs --help-commands`" for a list of cvs commands, and "`cvs --help command`" for help on a particular command.

The Corsica cvs repository is `/cvs/Corsica` on the MFE Sun workstations. In order to use cvs you must be a member of UNIX group "`corsica`."[1]You must either set the environment variable `CVSROOT` to `/cvs/Corsica`, or specify the path with the "`-d repository`" option to the `cvs` command.

This repository contains the following modules:

- `corsica`: the source tree
- `docs`: this document
- `imake`: the Basis/Imake system.
- `fftpack`: a library needed by Basis' `fft` package.

Thus, to check out the source, simply do:

[1]Type "`groups`" at the shell prompt to list the groups to which you belong

```
% cvs -d /cvs/Corsica co corsica
```

CVS will create a new subdirectory, `corsica`, containing the source tree.

Ordinarily CVS examines all files and directories (except for the "`CVS`" database directories). This results in cluttered output as CVS informs the user that it doesn't know about files like "`Makefile`." One can set the environment variable `CVSIGNORE` to cause CVS to ignore certain names or patterns. Here is a useful example:

```
setenv CVSIGNORE \
     "Makefile *.mac .corsica-command-line *.cmt $CPU"
```

This ignores makefiles, the output from Basis' `mac` program, the dot-files that `corsica` creates when it starts up, Basis comment database files, and the directories where the binaries and other build output are stored (the `$CPU` directories—see the Imake section below). (Note that the double quotes are important as they allow the environment variable `$CPU` to be expanded.) Here are two other useful customizations:

```
alias ch 'cvs -n -q update'
alias llog 'cvs log -d"`date +%T`PST" \!*'
```

The `ch` alias causes CVS to print the abbreviated output from an update, without actually doing the update. This is a handy way to check the status of files in the source tree. To get a list of files that have been modified or added, do

```
% ch | grep -v \^U
```

The`llog` alias gets the last log entry for a particular file.

B.2 The Basis system

As mentioned in the Introduction, detailed information on Basis is available from the Basis documentation [7–12]. This section highlights a few CORSICA-environment-specific details.

CORSICA must be built with the EZN graphics package and the PFB portable binary file package. The latter is relatively new and was not completely robust until Basis 11, so one should avoid building CORSICA with older versions of Basis.[2] The Basis version be checked by running the Basis executable (`basis`) and looking at the first few lines of output. You can check the version of an existing CORSICA executable by running it and typing "`version`" at the `corsica>` prompt.

Two Basis-related environment variables must be set to build the code. The `BASIS_ROOT` variable gives the path to the Basis installation and `NCARG_ROOT` gives

[2]Unfortunately, save files created with Basis 9 (or earlier) versions of CORSICA may not be readable with a newer code. The remedy is to build a code with Basis 9.11. When built with this version, CORSICA can read the old-format files and write the PFB-format files, thus serving as a conversion tool. Hopefully there aren't too many old-format files floating around.

the path to the NCAR graphics installation. These may be inter-dependent; for example, Basis 11.4 should only be used with NCAR Graphics 4.0 or newer. Contact the Basis maintainer for details. You can review the Basis-relevant settings by typing the command `basisenv` at the shell prompt:

```
gandalf:jac(config)> basisenv
Environment variables that need to be set by users:
   BASIS_ROOT /usr/local/vbasis
   NCARG_ROOT /mfe/theory/Basis/ncarg4
      or
   NCARG_PARAMETER_FILE (NCAR 3.1)
If using NCAR 4, set environment variable NCARG4.
   NCARG4 (any nonzero value)
If ATC-GKS used in program set gksdir.
   gksdir (ATC-GKS)
To use PACT utilities if available set SCHEME
   SCHEME /mfe/theory/Pact/pact/scheme
Environment variables that need to be set by Basis source installers:
   PACT /mfe/theory/Pact/pact
Environment variables that need to be set by Imake code developers:
   CPU SOL
   IMAKECPP (choice of cpp)
```

The `NCARG4` setting is not used by Imake. If ATC GKS is installed, then `gksdir` should be set. (Whether or not it is used depends on the Basis/Imake configuration files.) `SCHEME` must be set to use the PDB utilities `pdbdiff` and `pdbview`. The former can be particularly useful for comparing PDB save files. `PACT` is not used explicitly unless you are installing Basis, however `$PACT/bin` should be in your path if you want to use `pdbdiff`. `CPU` and `IMAKECPP` are related to Imake and will be discussed below.

Note that Corsica extends the Basis system to allow packages to be written in C++. These extensions are implemented via customizations to the Basis `mac` utility and via a C++ class library in `corsica/libs/CCL`. For more information, see Appendix C.

B.3 The Imake system

Corsica uses the UNIX `imake` utility to build its makefiles. This utility reads (via the C preprocessor) platform- and project-specific variables and macros from a set of central configuration files. The user can then write portable "`Imakefile`s" using high-level macros. General Basis-related definitions are located in

```
$BASIS_ROOT/imake/config
```

and

```
$BASIS_ROOT/imake/config/Basis.
```

CORSICA-specific files are contained in the `corsica/config` directory. The Basis/Imake system and its CORSICA extensions are described in detail in Appendix E.

Note: You *must* use the GNU version of `make` with this system. This is not a general requirement of `imake`, but it is much more powerful and portable than relying on the vendor's `make` utility.

The user must set the `CPU` environment variable to the value appropriate for the platform on which the code is being built. This variable is used to name the "output" subdirectories where all build products are created. For example, on Suns running Solaris, the proper value for `$CPU` is `SOL`. After the build, the CORSICA executable will be `corsica/src/SOL/corsica`. Table B.1 lists various values of `$CPU` appropriate for various platforms. The `corsica/README` file should also list all possible options, along with other details for each platform.

Table B.1: Values of `$CPU` for various platforms

`SUN4`	Sun Sparcstations running SunOS 4.x
`SOL`	Sun Sparcstations running Solaris 2.x or newer
`HP700`	Hewlett-Packard 9000/7xx class workstations
`RS6000`	IBM RS6000s
`SGI`	SGI R8000 and R10000-based workstations
`ALPHA`	DEC Alpha's running OSF/1
`C90`	Cray C90
`CRAY2`	Cray 2

Note that some C preprocessors do not work correctly with `imake`. For this reason it is sometimes necessary to override the default with the `IMAKECPP` environment variable. Usually this is simply set to `$BASIS_ROOT/bin/imakecpp`, which is a shell-script that runs the correct C preprocessor. Platforms that need special treatment should be mentioned in the `README` file.

Imake-produced makefiles include rules to rebuild target makefiles if the corresponding `Imakefile` is modified. However, a "bootstrapper" script is needed to create the first makefile. For Basis, this is the script `mkbmf` ("make Basis makefile"). CORSICA supplies a custom script, `corsica/MMF`, that *must* be used in order to enable the CORSICA customizations.

B.4 Building the code

Once the code has been checked out and all of the Basis and Imake related environment variables have been set, building the code should be straight-forward:

```
    % cd corsica              # go to the top of the source tree
```

```
% MMF                      # bootstrap the top-level Makefile
% make World >&make.out  # build the code
```

The resulting executable will be `src/$CPU/corsica`. If the build fails, examine `make.out`, fix the problem, and repeat.

Appendix C

Extending Basis to work with C++

The CORSICA project extended the Basis `mac` utility to generate stubs and interface information to allow packages to be written in C++. A primary design goal was to make the coupling between Fortran, Basis, and C++ as transparent as possible. Thus, `mac` generates stubs to allow calling C++ routines from Fortran and from Basis, and for calling Fortran routines from C++. `mac` also generates an C++ interface to the package's data. The various C++ wrappers and interface routines make use of the CORSICA class library (CCL), which is discussed in Appendix D.

Note that the CORSICA extensions do not add any additional functionality to Basis itself. In particular, there is nothing object-oriented *OO* about the interface that a C++ package presents to Basis. Instead, any C++ routines that are added to a Basis package must be callable by Fortran. However, this does not prevent one from using an OO design in the implementation of the package.

The next section describes the use of this system. The following section discusses the implementation of the interface using a simple example.

C.1 Using the Corsica Basis/C++ interface

The CORSICA modifications to the `mac` program are enabled by invoking `mac` with the "`-c`" option. When invoked in this manner, `mac` understands two new function types, `cfunction` and `csubroutine`, and it generates two additional files, ***pkg***`.h` and ***pkg***`Base.cc` (where "***pkg***" is the name of the package).

As the names imply, `cfunction` and `csubroutine` are used for functions and "subroutines" (functions returning `void`) written in C++. The implementation of such functions must `#include` the ***pkg***`.h` file, and must pass all arguments by reference. Note that the current implementation does not support passing `character*()` arguments to functions written in C++.

From the C++ point-of-view, all package variables and functions appear to be static members of a class named ***pkg***, which we shall refer to as the *package-class*.[1] For example, here is a variable descriptor file that declares a function that takes a 2-D array and 2 integer argument giving the array size, and returns the sum of the squared values of the elements.

```
tst
******* C_Function_Example:
testfcn(a:real, m:integer, n:integer) csubroutine
```

and here is an example C++ source to implement this function:

```
#include "tst.h"
Real tst::testfcn(Real &a_data, Integer &m, Integer &n)
{
  Matrix<Real> amat(m,n,&a_data);
  Real ssum = 0;
  for (int i = 1; i <= m; i++)
    for (int j = 1; j <= n; j++)
      {
        ssum += Sqr(amat(i,j));
      }
  return ssum;
}
```

Note the handling of the array `amat`. All arguments are passed in by reference, as is required by Fortran. Remember that `a_data` is a reference to the raw Fortran array, and `amat` is a `Matrix<Real>` constructed to alias this data. Also note that the function that is implemented is `tst::testfcn()` and not a global function `::testfcn()`. A global function is created by `mac`, but its name must be mangled in the appropriate manner so that it can be called from Fortran (and Basis). The global function simply calls the static member function. This does incur extra function call overhead, but it has the advantage that the function has the same name everywhere; i.e. in any member of the `tst` package-class (or a class that inherits from `tst`, which is our anachronistic method for importing the namespace) one can simply write:

```
Matrix<Real> amat(3,5);
// ...initialize amat...
Real s = testfcn(amat,3,5);
```

And in Fortran:

[1]This is an anachronistic method of implementing separate namespaces for separate packages; if this were written using ANSI C++, actual namespaces could be used, but this feature is still not widely available.

```
real amat(3,5)
real s
! ... initialize amat ...
s = testfcn(amat,3,5)
```

Thus we have the same interface from both Fortran and C++.

Package variables are handled in much the same way as package functions: they are implemented as static data members in the package-class. Arrays declared in the variable descriptor file show up as objects of the corresponding container class (`Vector<T>`, `Matrix<T>`, `Array3<T>`, etc.[2]). Note that the data in these arrays is owned by Fortran. This has two implications. First, invoking the `resize` member function is a bad idea. Furthermore, unlike Fortran arrays, the C++ arrays know know their size. There needs to be a way for these sizes to be initialized and, in the case of dynamically allocated arrays, to be reset if the Fortran arrays are resized. Unfortunately, this is not done automatically.[3] In order to allow synchronization, the package-class has a `sync()` method that calls the `init()` method of each dynamic array with the new data address and the new dimensions. This should always be called after a call to `Basis::gchange()` or other Basis memory reallocation function.

A full example, creating a Basis code with an extended version of the above `tst` package, is checked in to CVS as the module `docs/cxxtst`. To check it out, do

```
% cvs -d /cvs/Corsica co -d cxxtst docs/cxxtst
```

To build the example code, one must have access to a built CORSICA source tree (since the example code needs `libCCL.a` and associated headers). Denoting the *relative path* (this is important, due to the way Imake works) to the top CORSICA source tree by `$CTOP`, this example can be built by typing:

```
% MMF $CTOP
% make Makefiles
% make
```

(This may only work on Solaris. It should work on other platforms with a little hacking to get the templates instantiated properly. Check the `corsica/config/*.config` files to see how CORSICA handles this.)

[2]Higher dimensioned arrays are not currently supported, although it would be straight-forward to extend `Array3<T>`.

[3]Automatic synchronization would be a fairly trivial change to the Basis runtime system. One needs to store another field in the runtime database informing Basis that this array is shadowed in C++, and then Basis could call the appropriate resize function whenever its reallocation functions are called.

C.2 The C++ interface: Inner working

Interfacing C++ with Fortran is generally non-portable and somewhat messy. As a result, the mac-generated interface is also somewhat messy. This section will cover the highlights of this interface's internals.

As was mentioned in the previous section, mac creates two files relating to the C++ interface. *pkg*.h declares the interface, and must be included by all C++ source that wants access to this interface. *pkg*Base.cc defines the stub and initialization functions. These include certain static initializers that are executed before the C++ main program is called.

Consider the docs/cxxtst example mentioned above. The variable descriptor file declares a group of functions and a group of data:

```
tst
******* C_Function_Example:
testfcn(a:real, m:integer, n:integer) real cfunction
testsq(a:real, m:integer, n:integer) csubroutine
testinit() csubroutine
testfoo() subroutine
******* C_Data_Example:
m integer
n integer
data(m,n) _real
sdata(10,10) complex
```

The interface file, tst.h, first includes a number of standard header files and then defines macros that are used to hide the Fortran-mangled function and common-block names (the latter appear to C++ as global data structures). For example, here is a portion of tst.h:

```
#if defined(sun)
/* Group C_Function_Example */
#define tst_testfcn testfcn_        /* c-function (sun) */
#define tst_testsq testsq_          /* c-subroutine (sun) */
#define tst_testinit testinit_      /* c-subroutine (sun) */
#define tst_testfoo testfoo_        /* subroutine (sun) */
/* Group C_Data_Example */
#define TST10_C_Data_Example tst10_ /* common (sun) */
#define TST16_C_Data_Example tst16_ /* common (sun) */
#define TST14_C_Data_Example tst14_ /* common (sun) */
#endif  /*** sun ***/
```

Finally, the tst package-class is declared. Here is an elided version:[4]

[4]The actual file also contains code to support the Basis ctl functionality. Examine the files for details.

```
class tst {
public:
// Group C_Function_Example
  static Real testfcn ( Real &, Integer &, Integer & );
  static void testsq ( Real &, Integer &, Integer & );
  static void testinit ( void );
  static void testfoo ( void );
// Group C_Data_Example
  static Integer  &m;
  static Integer  &n;
  static Matrix<Real> data;
  static Matrix<Complex> sdata;

  static void sync();
};
```

This class definition represents the full C++ interface to the package. Note that `tst::testfoo()` is a Fortran function, which is implemented in the file `foo.m`. Also note that all data members will be references to actual data declared in the Fortran common blocks that `mac` defines. Scalar variables are explicit C++ references. Arrays refer to the Fortran data through an internal pointer (see below for the initiazation proceedure).

The implementation file, `tstBase.cc` contains function and data definitions for each group in the variable descriptor file. In this example, the first section defines the various wrapper functions for the `C_Function_Example` group. For the functions written in C++, a global stub is created so that the function can be called from Fortran:

```
// testfcn is a c++ function
extern "C" {
  Real tst_testfcn( Real &a, Integer &m, Integer &n )
    {
      return tst::testfcn( a, m, n );
    }
};
```

For the functions written in Fortran, the external function is declared and a static member function is generated that calls the external function:

```
// testfoo is a fortran subroutine
extern "C" void tst_testfoo( void );
void tst::testfoo()
  {
      ::tst_testfoo();
  }
```

Note the use of machine-independent function names; e.g. `tst_testfoo`.

The next section handles the data for the `C_Data_Example` group. Here is an elided version:[5]

```
extern "C" struct {
   Complex sdata[(10)*(10)];
 } TST14_C_Data_Example;

extern "C" struct {
   Real *data;
 } TST16_C_Data_Example;

extern "C" struct {
   Integer m;
   Integer n;
 } TST10_C_Data_Example;

Integer  &tst::m = TST10_C_Data_Example.m;
Integer  &tst::n = TST10_C_Data_Example.n;
Matrix<Real> tst::data(false);
Matrix<Complex> tst::sdata(1,10,1,10,
                           TST14_C_Data_Example.sdata);
```

The three data structures are C's view of the three common blocks that `mac` has created, as can be seen by comparing the above definitions with the data declarations in `tst.mac`:

```
complex sdata(10,10)
integer m,n
Dynamic(data, real, [m,n])
common /tst14/ sdata
common /tst16/ Point(data)
common /tst10/ m, n
```

The last four lines of C++ *define* the static data members that are declared in `tst.h`. The scalar references`tst::m` and `tst::n` are initialized to directly alias the common block data. The dynamic array `tst::data` is initialized with the constructor

[5]The Cray loader does not handle the code, as shown, correctly. As a result, a copy of this file, `tstCBase.c`, is compiled with the C compiler. Using strategically placed tests on `__cplusplus`, the C version actually *defines* these data structures, rather than declaring them to be `extern`. Both `tstBase.o` and `tstCBase.o` are loaded, and the loader then (somehow) patches things up. This hack should not be required, and the need for it should be re-checked. Perhaps newer compilers have removed the problem, although one would expect multiple-definition errors if the loader worked as it does on other platforms.

```
Matrix(Boolean willBeMyData),
```

which creates an uninitialized `Matrix<T>`, as it must since the Fortran array has not yet been allocated. Finally, `tst::sdata` is initialized with the constructor:

```
Matrix(Integer rb, Integer re, Integer cb, Integer ce,
        T *dat, Boolean makeCopy = false)
```

which defines `tst::sdata` to be a complex array with the first and second indices running from 1 to 10, and with the data located at the address

```
TST14_C_Data_Example.sdata.
```

The last argument defaults to `false`, so no copy is made and `tst::sdata`'s data is aliased to the data in the common block.

Finally, the `sync` function is defined:

```
void tst::sync()
{
  data.init(1,m,1,n, TST16_C_Data_Example.data);
}
```

Calling `tst::sync()` causes the `init()` method (which has the same signature as the second constructor above) to be called for all dynamic arrays in the `tst` package, in this case the `tst::data` object. It refers to the current values of the dimensions (through the references `tst::m` and `tst::n`) and to the current value of the data pointer stored in the Fortran common block.

C.3 The Basis mac utility

Our original `mac` modifications were done to an old version that was written in C. In Basis version 10 and newer `mac` is a PERL script. The Basis group (Zane Motteler, in particular) kindly ported our functionality to this new implementation. Problems do occasionally occur, especially when porting to new platforms. The mac script is quite complicated, so if problems arise, figure figure out what is going wrong in the ***pkg*.h** and ***pkg*Base.cc** files and to then either mimic what is done for other platforms, or contact the Basis group for help.

Appendix D

The Corsica Class Library

The Corsica class library, `libCCL.a`, contains various container classes, mathematical classes, and utility classes. These are listed in the following sections. For a full description of the public interfaces to these classes, see the header files in `corsica/libs/CCL`.

D.1 Library overview

Basic data types

- `template<class T> class CCLComplex`
  `CCLComplex` allows us to do `Complex<float>` and `Complex<double>`.
- `class Character`
  Fortran character string class, sort of.

Containers

- `template<class T> class Array`
  `Array` is a simple linear container, not used for number crunching.
- `template<class T> class Matrix`
- `template<class T> class Vector : public Matrix<T>`
- `template<class T> class Array3`
  `Matrix`, `Vector`, and `Array3` are used for number crunching. They are pre-expression templates and don't provide overloaded operators as this was found to be too slow. They have the ability to manage data that they don't own, allowing them to shadow Fortran arrays whose memory is managed by Basis. Also, they look and act like Fortran arrays, making it easy for non-C++ programmers to figure out the code.

Mathematical classes

- `class BSpline`
- `class LinearFit`
  These spline classes are used for general fitting purposes and are also used in the PDE solvers.
- `class DiffusionEqn`
  An abstract class defining the interface for our diffusion equation solver package. See comments below.
- `class ZeroD : public DiffusionEqn`
  A zero-dimensional option (I don't know if this is tested).
- `class BSplineFE : public DiffusionEqn`
  A `DiffusionEqn` class that uses B-Spline finite elements as a solution technique.
- `class LinearFE : public DiffusionEqn`
  A `DiffusionEqn` class that uses linear finite elements as a solution technique.

From the comments in `DiffusionEqn.cc`:

```
      This class is used to solve a parabolic PDE of the form

            dU   d       dU                    dU
         H1 -- - -- [ H2 -- + H3 U + H4 ] - H5 -- - H6 U - H7 -
            dt   dx      dx                    dx

                dU
           H9 -- (xe) - H10 U(xe)
                dx

      Subject to the initial condition

         U(x,0) = U0(x)

      and the boundary conditions

         dU                   dU
         --(0,t) = 0,   alpha --(xe, t) + beta U(xe,t) = gamma.
         dx                   dx

      H1-H10 can be non-linear functions of U and its derivatives
      with respect to x.
```

```
    H8 is supplied as an aid for implementing a drag term in
    the heat equation.
```

Note that it is assumed that `H2` $\to 0$ at the origin; i.e. that the equations are singular there and no boundary-condition is required. Also note that there is no provision for making the off-diagonal transport terms fully implicit; i.e. there are no flux terms proportional to `dV/dX`, where `V` is some other field. Rather these terms are put into `H3` or `H4`, whichever is appropriate. The exception to this rule is that we added `H8` to allow us to make the drag term fully implicit in the heat equations.

The CORSICA `mth` package provides a Basis parser interface to these classes. This is really nice when one wants to play with new sorts of transport models or when one wants to investigate the numerics of the scheme—this can be done without recompiling the code.

Library interfaces

These classes simply wrap various library routines. These shouldn't be classes, but there were no namespaces in 1993.

- `class Basis`
  Provides a limited number of wrappers for `libbasis.a` functions.
- `class CPlusMath`
  Provides a number of linear algebra, etc., routines.

Appendix E

Imake

Imake is a UNIX tool to help build portable programs [16]. It excels at managing complex builds of software spanning multiple directories, having dependencies on multiple external libraries, having platform-specific dependencies, or involving complicated build procedures.

The name "imake" is short for "include make." The basic idea is quite simple: A makefile is constructed from a template using the C preprocessor (`cpp`). The template reads files (using `cpp` "`#include`" statements) that have definitions specific to building applications on a particular platform. The template also includes files that define various `cpp` macros (defined by `cpp` "`#define`" statements) that provide simple interfaces to complex make tasks. The last thing that the template includes is the user's "`Imakefile`," which can be written in a portable, high-level manner. Finally, the output is cleaned up to satisfy `make`'s picky spacing rules, and the makefile is written to disk.

The `imake` module in `/cvs/Corsica` was set up to manage the source for the Basis system, as well as to build Basis applications. It includes

- a custom imake executable (in `imake/src`),
- generic config files (in `imake/config`),
- config files specific to Basis applications (in `imake/config/Basis`).

The `imake` executable itself is a slightly modified version of the X11R3 version. The modifications fix problems with parsing colons in certain constructs that are specific to GNU make.[1] The generic Imake macros were borrowed with minor changes from the InterViews distribution. (These were chosen over the X11 versions because the build byproducts are kept in separate subdirectories.) The macros in `imake/config/Basis` were added to simplify the building, testing, and installation the Basis distribution,

[1]These customizations may not be required with newer versions of `imake`.

and to build Basis applications.[2]

The next section is a primer on using this system. The following section explain how this system works, starting with the template file, the various config files, etc. The final section will discuss CORSICA customizations.

E.1 An Imake Primer

Some preliminaries are necessary to use the Basis/Imake system. Various environment variables need to be set for this system to work. These include `$CPU`, `$BASIS_ROOT`, `$NCARG_ROOT`, etc. See Appendix B for more information on setting these variables.

The Basis/Imake system puts all object files, libraries, and executables in subdirectories that have the name specified by `$CPU` (or a name derived from `$CPU`). This not only keeps the source directory tidy, it also allows the same source tree to be used to build the code on multiple platforms, or to be built with different levels of optimization, without getting the binary files mixed up.

To bootstrap the system, there must be an "`imake`" executable in the user's path. Most systems have one in the X11 binary directory

```
/usr/bin/X11 or /usr/local/bin/X11,
```

and a version is also installed with the Imake system (usually in `$BASIS_ROOT/bin`). This can be checked by typing "`which imake`" at the shell prompt. If it says "`imake: Command not found`" then the path needs fixed.

One *must* use the GNU version of the make utility with the Basis/Imake. (This is good advise in general since it is both more portable and more powerful than traditional UNIX make utilities.) This is usually installed as `gmake` in `/usr/local/bin`. To avoid mistakes, it is good practice to alias `make` to `gmake`. The following examples assume this has been done and simply refer to GNU make as "`make`."

Finally, the Basis/Imake system assumes that all MPPL source files explicitly use "`include`" statements to include the required "`.mac`" files.

Imake Prerequisites in a Nutshell

- Set the `$CPU` environment variable.
- Set other environment variables.
- Check that `imake` and GNU `make` (`gmake`) are in your path.
- Make sure that all MPPL sources explicitly include the appropriate `.mac` files.

[2]The infrastructure to support building Basis is still there. This is no longer needed and should be cleaned up at some point. The `imake` module does use pieces of this to do its own install, however, so care should be taken in doing the house-cleaning.

A "Simple" Example

Use `cvs` to check out the `cbk` package from the `docs` module as follows:

```
% cvs -d /cvs/Corsica co -d cbk docs/cbk
```

(The "`-d cbk`" causes CVS to put the sources under the directory "`cbk`" rather than "`docs/cbk`.") Now create a file named "`Imakefile`" that contains the following lines:

```
SimplePackageTarget(cbk)
SimpleBasisTarget(cbk,ezn)
```

(The indentation is cosmetic; the lines should not be indented).

The first line, "`SimplePackageTarget(cbk)`," indicates that `cbk` is a "Simple" package. A Simple package is one that has a single MPPL source file, a single variable descriptor file (VDF), and that follows the default naming conventions:

```
*.m     -- MPPL source files
*.v     -- Variable descriptor files
*.pack  -- config file, containing only the package
           information for the present package
*.yy.m  -- MPPL output from mac
*.mac   -- include file written by mac
*.pkg.m -- MPPL output from config
```

For "Simple" packages, the names are further restricted to match the package. Thus, the `cbk` package includes source `cbk.m`, VDF `cbk.v`, and *pack* file `cbk.pack`. These naming restrictions allow all of the file names to be deduced from the package name, greatly simplifying the `Imakefile`.

`SimplePackageTarget(cbk)` creates the part of the makefile necessary to build `pkgcbk.o`, the *package object*.[3] `SimpleBasisTarget(cbk,ezn)` creates the part of the Makefile that runs Basis' `config` command and loads the code. The first argument specifies the name of the resulting executable, and the second argument specifies the name of the graphics package.

Ordinarily one would need to modify cbk.m to add the statement

```
include cbk.mac
```

This is not necessary here since `cbk.m` does not use any groups; this is definitely the exception rather than the rule.

To build the code, enter the following commands:[4]

[3]The pkgfoo.o files consolidate the results of compiling all source files and the `.yy.m` file. The choice to use `.o` files (created with `ld -r` on most systems) rather than libraries was a compile-time optimization choice. Given the speed of todays computers, this is probably no longer necessary, and furthermore it may be responsible for a number of problems that we have had with C++ templates.

[4]Note that the path in the first line may vary depending on the installation—on platforms where Basis is maintained by non-MFE personnel, the Imake system is usually installed separately from the Basis system.

```
% $BASIS_ROOT/bin/mkbmf
% make Makefiles
% make
```

The `mkbmf` script is a bootstrap program that builds the top level makefile from the `Imakefile`. Once the makefile is present, it can be reconstructed by typing "`make Makefile`." (Actually, this is usually unnecessary as the `Makefile::` target has a dependency on the `Imakefile`, and GNU make should automatically execute this target anytime it notices that `Makefile` is out of date. However it will not automatically take the next step of running "`make Makefiles`.")

The second command, "`make Makefiles`," causes make to descend the directory tree, building all of the makefiles at lower levels. In this case, the only lower level is the `$CPU` directory, which is created if it doesn't already exist.

Finally "`make`" or "`make all`" will build the `all::` targets in the `$CPU/Makefile`. In this case, it builds `$CPU/pkgcbk.o` and then loads the `cbk` code.

A More "Complex" Example

Suppose a project contains more than one source file. Here is an example `Imakefile` for such a project:

```
VDFS = foo.v
SRC.M  = foo.m bar.m
SRC.CC = ack.cc
ComplexPackageTarget(foo)
ComplexBasisTarget(xfoo,ezn)
```

The first line lists the variable descriptor file for this package. The second and third lines list the MPPL and C++ sources (there is also a `SRC.C` and `SRC.F` for C and Fortran sources). The last two lines are just the "Complex" equivalents of the `SimplePackageTarget` and `SimpleBasisTarget` explained above.

There are also make variables available for specifying external packages and libraries; for example, the HAWC code uses the history package from Basis and uses external libraries for its FFTs, its ODE solver, and for HDF output:

```
EXTRA_INCLUDES.M = -I$(HAWC_LIBDIR)
EXTERN.LOCALLIBS = $(FFTLIB) $(ODELIB) $(HDF.LIB)
EXTERN.PKGOBJS   = $(HDFPKG) $(HST2_FILES)
EXTERN.PKGS      = $(HDF.PACK) StdPackFile(hst) StdPackFile(pfb)
```

The first line is used to add additional `-I` options to the MPPL command line. The second specifies the external libraries for the load line. The third specifies extra package objects (`pkghdf.o`, `pkghst.o`, etc.). The last line specfies the `.pack` files that correspond to these extra packages. (The make variables on the right are defined elsewhere in the `Imakefile`.)

Many More Possibilities

The above examples are fairly simple, and thus they may make the Basis/Imake system look better than it is. Complex programs with many packages, may require complicated `Imakefile`s. But maintaining these is generally much simpler than maintaining the makefiles by hand, especially if you are building on a number of platforms. In summary, this approach:

- Keeps the Basis stuff localized in the Imake configuration files and hidden from the user, implying....
  - Basis version independence: The same `Imakefile` should work from one basis version to the next.
  - Portability: the same `Imakefile` can be used on all platforms that support Basis. If there are machine differences that the user must be aware of (e.g. the name of the IMSL library), then `cpp` statements can be used to specify the version appropriate for the machine:

    ```
    #ifdef SunArchitecture
          IMSL = -L/usr/local/lib -limsl
    #endif
    ```

- Flexibility: Remember that all Imake does is run the `Imakefile` through the `cpp` after including a template and a number of default definitions. One can use all of `cpp`'s capabilities in the `Imakefile`, and can also use any of the features of GNU make, which pass through `cpp` unchanged. Furthermore, the default template can be replaced with a custom version, allowing, for example, the declaration of new variables and macros that are global to a particular project. (CORSICA does this, as will be discussed below.)

Also note that there are many more macros and variables defined by the Basis/Imake system than have been presented in this section. The best way to learn about these is to look at the Imakefiles that use this system. Non-trivial examples include the CORSICA system, the HAWC code, and the HWDIA code. Finally, read the next section and look at the Basis- and CORSICA-specific config files.

E.2 Imake Undressed

This section describes the inner workings of the Imake system. A more detailed account of the general subject (sans Basis and CORSICA details) can be found in DuBois' book [16].

As described above, Imake relies on the C preprocessor to do most of its work. As macro-preprocessors go, `cpp` is not particularly powerful. This results in some rather messy code in order to get the macros to do what one wants. Frankly, a combination of `m4` and GNU make would be much more powerful. At any rate, this section will explain most of the details and tricks used to make this system work.

The Imake template

An ellided version of the default Basis/Imake template is shown in Fig. E.1. The actual file is[5]

```
$BASIS_ROOT/imake/config/Basis/UNIX.tmpl
```

The structure is fairly simple. First a number of initialization files are included. These define primitive macros, determine the system type, and allow site, platform, and Basis-specific overriding of macros that are defined later. Next the template includes generic variable definitions, Basis-specific variable definitions, generic *rules* (complex `cpp` macros) and Basis-specific rules. This is followed by a set of standard targets. Finally, the template includes `INCLUDE_IMAKEFILE`, which is normally defined to be `"Imakefile"`.

Common Idioms

Several common constructs should be explained before proceeding. The first is the mechanism by which the initialization files "override" definitions that come later. This is made possible by a using a combination of make variables and `cpp` macros. For example, the generic config files might contain something like the following:

```
#ifndef DefaultFooValue
#define DefaultFooValue 43
#endif

DEFAULT_FOO_VALUE = DefaultFooValue
```

Thus this file will only set `DefaultFooValue` if it has not previously been set. If `DEFAULT_FOO_VALUE` should be 22 on a particular platform, then one of the initialization files can include the line:

```
#define DefaultFooValue 22
```

and this will take precedence.

One might ask why `DEFAULT_FOO_VALUE` is used at all? `DefaultFooValue` could be used directly. There two reasons for this. One is that the resulting makefile is easier to read—if `DefaultFooValue` were used, only 43 (or 22) would appear in the final makefile. Second, this construct adds another level of flexibility since make

[5]Note that CORSICA replaces this template with its own version, as will be discussed in the next section.

```
/* Initialization */

#include <CppMacros.defs>
#include <Basis/Version.defs>
#include <arch.c>
#include MacroIncludeFile
#include <Basis/Project.config>
#include <site.defs>

/* Generic variables. */

#include <Generic.defs>
#include <Generic.tmpl>

/* Basis-specific variables. */

#include <Basis/Project.defs>
#include <Basis/Project.tmpl>

/* Rules */

#include <Rules.defs>
#include <Basis/Project.rules>

/* Predefine common targets for all Makefiles. */

all::
MakefileTarget()
Makefiles::
depend::
install::
CleanTarget()

/* Include the local Imakefile. */

#include INCLUDE_IMAKEFILE
```

Figure E.1: Template makefile (elided) for the Basis system.

variables can be overridden either in the `Imakefile` or on make's command line. For example, suppose that a particular application *really* needed `DEFAULT_FOO_VALUE` to be 123, even though all other applications being built with this Imake system on this particular platform use 43. Since `Imakefile` is read last, the user can override the system's value by placing the line

```
DEFAULT_FOO_VALUE = 123
```

in her `Imakefile`. Equivalently, she can give the value on the command line:

```
% make DEFAULT_FOO_VALUE=123 all
```

One disadvantage of this parallel technique is that the config files become very cluttered with `cpp` logic, making it difficult to connect the final makefile to the config files. For this reason, the definitions of make variables are put into separate files from the definitions of the `cpp` macros. The latter typically have the suffix `.defs` and the former have the suffix `.tmpl` (for "definitions" and "templates"). As an example, the make variable `IMAKE` is defined as follows:

```
/* in Generic.defs: */
#ifndef ImakeCmd
#define ImakeCmd imake
#endif

/* in Generic.tmpl: */
IMAKE = ImakeCmd
```

But that's not the whole story—the Basis/Imake system supplies its own version of `imake`, so we find:

```
/* in Basis/Project.config */
#ifndef ImakeCmd
#  ifdef UseInstalled
#    define ImakeCmd $(BASIS_BINDIR)/imake
#  else
#    define ImakeCmd $(TOP)/imake/src/$(ARCH)/imake
#  endif
#endif
```

This allows either the installed version of the Basis-specific Imake executable to be used (if `UseInstalled` is defined), or the version in the Imake source tree. Furthermore, this definition is also protected. Particular platforms can override this definition by placing a definition in the "`MacroIncludeFile`" (more on this file in the next section).

Another idiom in the Imake system is the extensive use of double-colon ("`::`") rules. This is a feature of make that allows a particular rule to have many parts, all of which are executed when that target is made. The default targets (`all`, `install`, `clean`, etc.) are all done in this fashion, allowing the user's `Imakefile` to add to the list of things done for these targets, either directly or indirectly via a `cpp` macro.

One problem with `cpp` is that it often strips whitespace, especially within macros. In order to ensure that the output ends up the way the user wants it, Imake has a special end-of-line sentinel for use in macros: the characters "`@@`." Here is a simple example demonstrating its use:

```
#ifndef Clean
#define Clean(files)           @@\
clean::                        @@\
        @$(RM) files
#endif
```

Without the sentinels, `cpp` would likely expand `Clean(foo)` to

```
clean:: @$(RM) foo
```

The sentinel serves as a place-holder. When Imake goes through the `cpp` output to clean up the spacing, it replaces the `@@` sequence with newlines and fixes up the tabs if these happen to be identified as actions. As a result, the line `Clean(foo)` will be expanded to

```
clean::
        @$(RM) foo
```

which is the desired result.

Initialization files

The first six files included in the example template are initialization files. These serve three purposes: to define some basic macros, to determine which platform this is and define standard macros identifying the platform, and to include architecture-, site-, and project-specific files that override the generic- and project-specific defaults set later.

Basic macros: CppMacros.defs

The file `CppMacros.defs` defines several basic macros, some of which depend on the type of C preprocessor being used. These macros are:

```
XCOMM
  -- Comment macro. The rest of the line will appear in the
  -- Makefile as a comment.

NILL
```

```
  -- Empty macro for passing an empty argument to another
  -- macro

Concat(a,b)
  -- Concatenates its arguments in a portable fashion.

Concat3(a,b,c)
  -- Concatenates its arguments in a portable fashion.

Stringize(a)
  -- Returns a string containing the macro argument.

YES (1)
NO (0)
```

The macro `XCOMM` is necessary because lines with '`#`' are interpreted by C preprocessors as directives (some `cpp`'s only view these as directives if they have the form `#[a-z]`, and thus lines starting with `##` are safe, but one should not count on this behavior). `XCOMM` is simply expanded to be the comment character. The `Stringize` macro and the concatenation macros have `#ifdef`s to determine whether to do these the UNIX way or the ANSI `cpp` way. Note that some `cpp`'s that claim to be ANSI conformant still screw these up (most notably, Cray's). For these platforms, one must override the default `cpp` with the `$IMAKECPP` variable.

Determining the platform: arch.c

The file `arch.c`[6] checks for `cpp` *trigger symbols*—macros that are predefined by `cpp` on a particular platform—to identify a particular platform, and to then define several standard symbols to identify the platform.[7] For instance, a section of the file is shown below:

```
#ifdef CRAY
#undef CRAY
#undef cray
#define CrayArchitecture
#define ArchitectureName CRAY
#define architectureName cray
#define MacroIncludeFile <cray.cf>
#define MacroFile cray.cf
#endif /* CRAY */
```

[6]Note that this file is named `Imake.cf` in newer Imake distributions.

[7]Some `cpp`'s don't define such a symbol, in which case Imake must put an explicit '`-D`*`trigger`*' on the `cpp` command line. See `$(BOOTSTRAPCFLAGS)`

This section first identifies the machine, then undefines the trigger and any other pre-defined macros that might screw up the use of their names elsewhere in the config files. Next it defines several macros, using a standard naming pattern that is repeated for all supported platforms. The first is the "architecture-indicator" symbol that can be tested in other config files and in the `Imakefile` to selectively include lines that are specific to that architecture. These always have the form *`System`*`Architecture`, where *`System`* is a name like `Sun`, `HP`, etc. Sometimes multiple symbols are defined; for example, a given OS may run on multiple hardware platforms. The `ArchitectureName` macros can be used for various purposes, including naming "object" subdirectories, although we rely on `$CPU` environment variable for this task.[8] The final lines identify the "`MacroIncludeFile`" that is included immediately after `arch.c`.

Vendor, project, and site configuration

After determining the system type, the template reads files that customize the system for a particular platform, a particular project, and a particular site. Note that in this incarnation, the separation between "platform," "project," and "site" dependencies is not particularly clean. The goal of this separation was that the "platform" file would be the same on all similar hardware/OS combniation, and that site-specific customizations would be done in "`site.defs`." The X11R6 version of Imake has apparently done a much better job of this, but the Basis/Imake versions have not been upgraded to incorporate these ideas.

The MacroIncludeFile: Vendor*.cf*

This file sets the version numbers for the OS and possibly for various tools, like compilers. These version number can then be used in logical expressions to set a variety of other default values. For instance, in `sun.cf` we have

```
#ifdef SVR4
# define OSName            SunOS 5.2
# define OSMajorVersion    5
# define OSMinorVersion    2
# define SolarisArchitecture
#else
# define OSName            SunOS 4.1
# define OSMajorVersion    4
# define OSMinorVersion    1
```

[8]There are two reasons for this: one is that some of the `ArchitectureName`s aren't very specific—we needed to differentiate between `$CPU == C90` and `$CPU == CRAY2`, for instance. Second, the original incarnation of these files supported "cross-compilation"; i.e. one could build the makefiles for the Cray on the Sun. Also, note that `ArchitectureName` is not defined in X11 config files - this is an InterViews relic.

```
#endif

#if OSMajorVersion >= 5
#  define BootstrapCFlags  -DSVR4 -DSYSV
#endif
```

The `SVR4` symbol is a bootstrap flag that the Solaris version of `mkbmf` (or `MMF`) defines. The definition of `BootstrapCFlags` shown above ensures that `make Makefiles` will also define this symbol. This file is also used to identify a variety of other options specific to the platform (again, some of these are really *site*-specific selections, but currently they reside in this file).

```
#if OSMajorVersion >= 5
#  define MpplOSDefine            SOL
#  define HasGraflib              NO
#else
#  define MpplOSDefine            SUNOS
#  define HasGraflib              YES  /* For now */
#endif

#define HasReadlineLibrary        YES
#define HasPDB                    YES
```

This example demonstrates another standard convention: `YES/NO` macros. Generally these macros have names that suggest a question. In particular, the form `Has`***`Property`*** are always of this type. Default values are conditionally assigned to these values in later files, but usually they are set in the vendor-config file. Since these macros are `always` defined, one must be careful to test them with `#if` and not `#ifdef`. This is a common Imake programming pitfall.

Basis/Project.config

The main use for this file is to override generic definitions by definitions that are specific to the Basis system. A good example was given above, where `Basis/Project.-config`'s definition of `ImakeCmd` overrode the default value set in `Generic.defs`. This file also sets certain options depending on which platform is being used. Most of this stuff has been moved directly to the `Vender.cf` file—this is a good example of the difficulties in separating the dependencies.

site.defs

This is where site dependencies *should* go. In practice, the Basis/Imake distribution pretty much identifies particular platforms with particular sites. If the system were used on the same architecture at a number of differently configured sites, then it might make sense to make a "`site.defs`" file for each of these specific sites (e.g. `nersc-c90.defs`) and then to link the appropriate file to `site.defs`.

Note that newer versions of Imake have somewhat different logic for including the `site.defs` file:

```
#define BeforeVendorCF
#include <site.defs>
#undef BeforeVendorCF

#include <vendor.cf>

#define AfterVendorCF
#include <site.defs>
#undef AfterVendorCF
```

This arrangement gives the `site.defs` the flexibility both to override defaults set in `vendor.cf` (in which case it becomes important to protect these definitions, something which is not usually done in the Basis/Imake files), and then to also react to definitions made in the `vendor.cf` file.

Generic and Project config files

Generic.defs and Generic.tmpl

After the various initializations and the battle-to-be-the-first-definition is over, the template reads a number of generic and project config files, starting with `Generic.defs` and `Generic.tmpl`. These files define hundreds of `cpp` symbols and make variable names that set default values for options, paths, command names, and command options and flags. Here are some examples:

- Default options (`YES` and `NO`):
  - `HasCPlusPlus`, `MakeHasPatterns`, `StripInstalledPrograms`
- Default paths:
  - `XIncDir` and `XINCDIR`
  - `UsrLibDir` and `USRLIBDIR`
  - `TmpDir` and `TMPDIR`
- Standard commands used to build, maintain, and install codes under UNIX:
  - `CcCmd` and `CC`
  - `InstallCmd` and `INSTALL`
  - `CpCmd` and `CP`

- Default flags, etc.

These files also include a number of other files that set symbols and make variables specific to certain computer languages.

Programming language support

One of the innovations of the Basis/Imake system was organized support for a variety of programming languages. The original X11 system assumed that everything was written in C. Use of other languages had to be coded directly into the `Imakefile`. The Basis/Imake system needed to support Fortran, MPPL, and C++, in addition to C. This support is implemented in the files:

```
      C.defs            C.tmpl
      C++.defs          C++.tmpl
      Fortran.defs      Fortran.tmpl
      MPPL.defs         MPPL.tmpl
```

For example, here is a slightly elided version of `Fortran.tmpl`:

```
             FC = FortranCmd
   DEBUGFLAGS.F = DefaultFortranDebugFlags
      OPTIONS.F = DefaultFortranOptions
  ALL_DEFINES.F = $(STD_DEFINES.F) $(PROJECT_DEFINES.F) $(DEFINES.F)
        FLAGS.F = $(REQUIRED_OPT_FLAGS) $(SBFLAG) $(DEBUGFLAGS.F) \
                    $(PROJECT_OPTIONS.F) $(OPTIONS.F) \
                    $(ALL_DEFINES.F) $(FFLAGS)
      COMPILE.F = $(FC) -c $(FLAGS.F)
    LDOPTIONS.F = $(STATIC_LOAD_FLAG) $(DEBUGFLAGS.F) \
                    $(OPTIONS.F) $(LDFLAGS)
         LOAD.F = $(FC) $(REQUIRED_OPT_FLAGS) $(LDOPTIONS.F)
```

These files adopt the convention of using a suffix on the make variable name to associate it with a particular language (some make utilities are allergic to this, but it works fine with GNU make). We also incorporate the standard names, such as `$(FFLAGS)` and `$(LDFLAGS)`.

Basis.defs and Basis.tmpl

These are Basis-specific versions of `Generic.defs` and `Generic.tmpl`, setting default options, paths, command names, and command flags, that are specific to building and using Basis.

Generic and Project "rules"

The *rules* files, `Rules.defs` and `Project.rules`, define `cpp` utility macros that are used write `Imakefiles` and to build other rules. Rules generally expand to multiple lines and may take arguments.

Rules.defs

A simple example of a rule was given above when explaining the `@@` sentinel. Here is a somewhat more complicate example, the rule to make a non-shared library (from `Rules.defs`):

```
#ifndef NormalNonSharedLibraryTarget
#define NormalNonSharedLibraryTarget(name,objlist) @@\
AllTarget(Concat(lib,name.a))                      @@\
                                                   @@\
Concat(lib,name.a): objlist                        @@\
        @echo "building $@"                        @@\
        $(RM) $@                                   @@\
        $(AR) $@ objlist                           @@\
        $(RANLIB) $@
#endif
```

This uses the `Concat` macro to build the name of the library. Note that we make use of implicit concatenation that occurs when a `cpp` macro argument appears next to a non-alphanumeric character. Thus, if the `Imakefile` contains the line:

```
NormalNonSharedLibraryTarget(foo,foo.o bar.o)
```

the makefile will contain:

```
libfoo.a: foo.o bar.o
        @echo "building $@"
        $(RM) $@
        $(AR) $@ foo.o bar.o
        $(RANLIB) $@
```

The `Rules.defs` file defines sixty or so rules. Some of these are listed below.

```
SetDefaultOptimization(flag)   - what is says
MakeWorld(flags)               - World:: targets
MakefileTarget()               - Makefile:: targets
AllTarget(deps)                - generate "all:: deps"
CleanTarget(files)             - clean:: targets
TidyTarget(files)              - tidy:: targets
MakeDirectories(step,dirs)     - make dirs on step:: targets
```

```
InstallProgram(program,dest)    - install:: targets
GenerateDependencyPatterns()    - generate pattern dependencies
MakefileObjectCodeDir(dir)      - Makefile:: in $CPU
MakeInObjectCodeDir()           - make all targets in $CPU
MakefilesSubdirs(dirs)          - Makefiles:: in subdirs
MakeInSubdirs(dirs)             - all targets in subdirs
```

Most of these are fairly straight-forward, but a few are pretty complicated, especially those that involve actions in subdirectories.

InObjectCodeDir

An `Imakefile` can actually serve as a template for two makefiles: the one in the same directory as the `Imakefile`, and the one in the `$(ARCH.S)` subdirectory.[9] We will refer to this directory as the *object directory.* In order to divide up the `Imakefile` into sections appropriate for each directory, we introduce the symbol `InObjectCodeDir`. This symbol is defined when the makefile is being built in the object directory, and it is undefined when the makefile is being built in the source directory. The result of these definitions is that `Imakefiles` often have the following structure:[10]

```
#ifndef InObjectCodeDir
MakeInObjectCodeDir()
#else
/* Build the program */
#endif
```

`InObjectCodeDir` is also used in the definition of certain symbols and variables; for example:

```
/* in Generic.defs */
#ifndef SrcDir
#ifdef InObjectCodeDir
#define SrcDir ..
#else
#define SrcDir .
#endif
#endif

/* in Generic.tmpl */
SRC = SrcDir
```

[9]`$(ARCH.S)` usually has the same value as `$CPU`, but it can also have values such as `$CPU.profile` or `$CPU.debug`. The `.S` stands for "special."

[10]Note that some macros, such as `ComplexPackageTarget()` hide this detail by defining both sections of the makefile.

This allows the `Imakefile` author to refer to the source directory as `$(SRC)` from either the source or object directories.

The `MakeInObjectCodeDir` rule generates all of the standard targets with actions that run `make target` in the object directory. Here is an ellided version:

```
#ifndef MakeInObjectCodeDir
#define MakeInObjectCodeDir()                                     @@\
MakefileObjectCodeDir($(ARCH.S))                                  @@\
MakeSubdirs($(ARCH.S))                                            @@\
InstallSubdirs($(ARCH.S))                                         @@\
CleanSubdirs($(ARCH.S))                                           @@\
TidySubdirs($(ARCH.S))
#endif
```

The ***Target***`Subdirs` macros are general macros that are being used here to do operations in the object directory. These will be discussed in the next section. The only target specific to the object directory is the one that makes the makefile. Here is its definition:

```
#ifndef MakefileObjectCodeDir
#define MakefileObjectCodeDir(dir)                                @@\
Makefiles::                                                       @@\
        @echo "Making Makefiles" \                                @@\
        "for $(ARCH) in $(CURRENT_DIR)/"Stringize(dir)            @@\
        -@if [ ! -d dir ]; then \                                 @@\
                mkdir dir; \                                      @@\
                chmod g+w dir; \                                  @@\
        fi; \                                                     @@\
        if [ -f dir/Makefile ]; then \                            @@\
                $(RM) dir/Makefile.bak; \                         @@\
                $(MV) dir/Makefile dir/Makefile.bak; \            @@\
        fi; \                                                     @@\
        $(IMAKE) $(IMAKEFLAGS) \                                  @@\
        -DTOPDIR=../$(TOP) -DCURDIR=$(CURRENT_DIR)/dir \          @@\
        -DInObjectCodeDir -s dir/Makefile
```

This prints a message, makes the directory if necessary, makes a backup of the makefile, and finally runs `imake`. Note the arguments given to `imake`. Not only is `InObjectCodeDir` defined, but `TOPDIR` and `CURDIR` are also defined. In this manner, the make variables `$(TOP)` and `$(CURRENT_DIR)` are always defined correctly. Thus, for example, one can refer to `$(TOP)/lib/CCL` from any `Imakefile` in the CORSICA hierarchy and get the correct relative path.

Like many of the more complex rules, this one makes use of the Bourne shell statements. Note that these commands have "\" *before* the `@@` sentinel. This is because the Bourne shell commands must be one-liners to work with make.

IntoSubdirs

The `IntoSubdirs` rule is a workhorse that is used by a number of other rules to carry out actions in subdirectories. For example, here is the definition of `CleanSubdirs`:

```
#ifndef CleanSubdirs
#define CleanSubdirs(dirs)                               @@\
IntoSubdirs(clean,dirs,"cleaning")
#endif
```

And here is the definition of `IntoSubdirs`:

```
#ifndef IntoSubdirs
#define IntoSubdirs(name,dirs,verb)                      @@\
name::                                                   @@\
        -@for i in dirs; \                               @@\
        do \                                             @@\
          if [ -d $$i ]; then ( \                        @@\
            echo verb \                                  @@\
            "for $(ARCH) in $(CURRENT_DIR)/$$i"; \       @@\
            cd $$i; \                                    @@\
            $(MAKE) VERS="$(VERS)" \
            COMMAND="$(COMMAND)" $(PASSARCH) name; \     @@\
          ) else continue; fi; \                         @@\
        done
#endif
```

This generates a `name::` target that prints a message (using the `verb` argument to specify the action name in the message), and then runs `make name` (with various command line arguments) in each of the specified subdirectories (`dirs`).

Basis/Project.rules

The file `Basis/Project.rules` defines about thirty additional rules, although a number of these are only used for installing Basis. First there are the rules similar to those used in the examples:

```
SimplePackageTarget(name)          - see above
ComplexPackageTarget(name)         - see above
ComplexBasisTarget(name,gfx)       - see above
ComplexBasisTarget(name,gfx)       - see above
DataPackageTarget(name)            - like Simple, but no functions
```

These rules are very simple to use, if they do what you want. They are implemented using make variables, and so, for example, only one `ComplexPackageTarget` can be used per `Imakefile`. If one needs additional flexibility, it may be necessary to use the “`Normal`” rules:

```
PackageObjectTarget(package,objlist,deplibs,locallibs,syslibs)
PackageLibraryTarget(package,objlist)
NormalPackageTarget(package,objlist,deplibs,locallibs,syslibs)
NormalBasisTarget(name,gfx,depobjs,objs,deplibs,locallibs,sys,flags)
NormalConfigFileTarget(name,packagelist,macfiles,mflags,includepaths)
```

These take all of their information via arguments, and so are extremely flexible, but not as simple to use. For detailed description, see the comments in `Basis/-Project.rules`. Next there are a number of utility rules:

```
BuildTimeStamp(target)              - adds glbtmdat.o
UseReadlineLibrary()                - use readline library
UseCCMain()                         - use C++ main program
SourceFileDependencies()            - general pattern dependencies
MacFileDependencies()               - dependencied to run mac
ConfigFileDependencies(name)        - dependencies to run config
```

and finally there are some rules that are only used to implement other rules. Here are a subset of these:

```
__BasisBuildRule(name)
__ComplexBasisTarget(name)
__CBTezn(name)
__CBTnog(name)
IntoSubdirs2(target,cmd,dirs,verb)
```

The rules starting with a double underscore are meant for internal use only.

ComplexBasisTarget

As an example of the hackery involved in getting this all to work, we'll take a look at `ComplexBasisTarget`. Actually, it's definition is deceptively simple:

```
#define ComplexBasisTarget(name,gfx) Concat(__CBT,gfx(name))
```

Thus `ComplexBasisTarget(foo,ezn)` expands to

```
__CBTezn(foo)
```

The definition of `__CBTezn` is also fairly simple:

```
#ifndef __CBTezn
#define __CBTezn(name)                                      @@\
  name.GFX.PACKFILE = StdPackFile(ezn)                      @@\
       name.GFX.PKG = $(EZN_PKG)                            @@\
      name.XGRAFCOR =                                       @@\
      name.GFX_LIBS = $(NCAR_LIBS)                          @@\
                                                            @@\
__ComplexBasisTarget(name)
#endif
```

At this point, our rule for building `foo` has become:

```
        foo.GFX.PACKFILE = StdPackFile(ezn)
             foo.GFX.PKG = $(EZN_PKG)
            foo.XGRAFCOR =
            foo.GFX_LIBS = $(NCAR_LIBS)

      __ComplexBasisTarget(name)
```

The last macro is defined to be:

```
      #ifndef __ComplexBasisTarget
      #ifdef InObjectCodeDir
      #define __ComplexBasisTarget(name)                  @@\
                                                          @@\
      AllTarget(name)                                     @@\
                                                          @@\
      __BasisBuildRule(name)                              @@\
                                                          @@\
      clean::                                             @@\
              -$(RM) name                                 @@\
                                                          @@\
      ConfigFileDependencies(name)
      #else
      #define __ComplexBasisTarget(name) NILL
      #endif
      #endif
```

In the object directory, this adds to the `all::` and `clean::` targets and sets up the dependencies needed to run config—I won't give the full expansion here. It does nothing in the source directory—it is assumed that this will be paired with a `ComplexPackageTarget`, which includes a `MakeInObjectCodeDir`. Finally, we need the rule to load the code:

```
#    define __BasisBuildRule(name)                                      @@\
name: $(PKG.OBJS) name.ConfigFileObjSuffix                              @@\
       $(BASIS_LD) -o $@ \                                              @@\
               name.ConfigFileObjSuffix $(PKG.OBJS) $(EXTERN.PKGOBJS)\ @@\
               $(TIMESTAMP_OBJ) $(READLINE_OBJS) $(EXTERN.LOCALLIBS) \ @@\
               $(STD.PKG.O) $(BASISLIB) $(name.GFX.PKG) $(LTSSLIBS)  \ @@\
               $(BASIS_SPECIAL_LIBS) \
               $(BASIS_STD_UNIXLIBS) $(EXTRA.SYSLIBS)
```

Essentially all communication between `ComplexPackageTarget` and `Complex-BasisTarget` is done via "global" make variables, which is why these macros can only be used once per `Imakefile`.

E.3 Corsica extensions

The CORSICA system extends the Basis/Imake system to support C++ extensions and to centralize other CORSICA configuration information. CORSICA supplies its own template makefile, overrides a number of the default Basis macros, and has a set of machine-specific configuration files. These files are contained in the `corsica/config` directory, and they include

- `Corsica.tmpl`: custom template,
- `Corsica.rules`: custom rules,
- `Corsica.config`: general configuration,
- `SOL.config, HP.config,` etc.: machine-specific configuration.

The machine-specific configuration files are a recent addition (Aug. '97). This file is read immediately after the `Vendor.cf` file, and has led to a much cleaner version of `Corsica.config`. The new organization is (not surprisingly) to define defaults in `Corsica.config` and override them in the machine-specific files. The current batch of machine-specific files are:

```
ALPHA.config   - DEC Alpha (not finished)
CRAY.config    - Cray
HP.config      - HP700 (not tested)
RS6000.config  - IBM RS6000 (not tested)
SGI.config     - SGI R8K (not tested)
SUN4.config    - Dummy - picks one of the following
SOL.config     - Solaris
SUNOS.config   - SunOS 4.x (not tested)
```

For example, Fig. E.2 shows the Solaris configuration file. As a result, there is now only one test on an *System*`Architecture` symbol in `Corsica.config`, and that is because the `SetDefaultOptimization` macro does not work on the Cray. (The UNICOS 9 `cf77` now accepts a more UNIX-like optimization Syntax, so this could be changed. However there may still be a problem as I believe that the `CC` uses a different syntax, and `SetDefaultOptimization` assumes that all compilers accept the same syntax. This may also be a problem if we use `KCC` on the workstations.)

E.4 Building and installing Imake

To build and install imake, first check out the distribution from CVS. Imake is usually installed in `$BASIS_ROOT`, so this must be set.[11] Also, check `Basis/Version.defs` to

[11]If Imake must be installed elsewhere, edit the *Vendor*`.cf` file and change the definition of `ImakeIncludes` and `ImakeCmd`.

```
XCOMM######################################################################
XCOMM##
XCOMM##   Architecture-specific Imake config-file for corsica.
XCOMM##   $Id: SOL.config,v 1.2 1997/08/14 00:00:22 jac Exp $
XCOMM##   Solaris version.
XCOMM##
XCOMM##   NOTE: this file is read very early. Only define macros here.
XCOMM##   Make variables will likely be overwritten by later files.
XCOMM##
XCOMM######################################################################

/*
// Configuration options
*/

#define HasNCAR         YES
#define HasNCAR4        YES

/*
// If any of the following are YES, then the paths must be set
// below, unless Corsica.config provides a suitable default.
*/

#define HasNAG          YES
#define HasIMSL         NO
#define HasPVM          NO
#define HasRBT          NO

/*
// Required libraries (comment out to use defaults).
*/

/* #define CxxFortranLibs */
#define LAPackLibrary -L/usr/local/lib -llapack -lblas
#define CorsicaClassLibrary $(TOP)/libs/CCL/$(ARCH.S)/libCCL.a

/*
// Optional Libraries
*/

#define NagLibrary  -L/usr/local/nag/nagfl15df -lnag
#define ImslLibrary /**/
#define PVMIncludePath /usr/local/pvm3/include
#define PvmLibrary -L/usr/local/pvm3/lib/SUN4SOL2 -lfpvm3 -lgpvm3 -lpvm3
#define RBTlibrary -L/home/tangyin/xu/CORSICA/DI -lfluedgebkgd

/*
// Compilation and load flags
*/

#define DefaultOptFlag -O4

/* -g doesn't turn off -O4 with new compilers */
#define MacOptimization -O1

#define CxxFlags -DIndirect_Indexing
#define CxxLoadFlags -ptv -I$(TOP)/src/ctr -I$(TOP)/libs/CCL

#define LoadMapOptions -Qoption ld -m
#define F77Flags -Nx600 -Nn2000

/* Hack to get templates loaded with Sun C++ 4.1: */
#define BasisSpecialLibs ptrepository/Templates.DB$(SLASH)*.o
```

Figure E.2: CORSICA config file for Solaris.

make sure that the version numbers are correct for the version of Basis that you are using. Then go to the top of the source tree and do:

```
% MakeTopMakefile
% make Makefiles
% make
% make install
```

This should install everything. Test the system out by checking out the `docs/cbk` example and building it on the new system.

Bibliography

[1] T. D. Rognlien, J. L. Milovich, M. E. Rensink, and G. D. Porter. A fully implicit, time dependent 2-D fluid code for modeling tokamak edge plasmas. *J. Nucl. Mater.*, 196-198:347–51, 1992.

[2] T. D. Rognlien, P. N. Brown, R. B. Cambell, T. B. Kaiser, et al. 2-D fluid transport simulations of gaseous/radiative divertors. *Contrib. Plasma Phys.*, 34:362–367, 1994.

[3] A. E. Koniges, J. A. Crotinger, and P. H. Diamond. Structure formation and transport in dissipative drift-wave turbulence. *Phys. Fluids B*, 4(9):2785–2793, September 1992.

[4] X.-Q. Xu, R. H. Cohen, J. A. Crotinger, and A. I. Shestakov. Fluid simulations of nonlocal dissipative drift-wave turbulence. *Phys. Plasmas*, 2(3):686–701, March 1995.

[5] M. A. Beer. *Gyrofluid models of turbulent transport in tokamaks.* PhD thesis, Princeton University, January 1995.

[6] S. W. Haney, W. L. Barr, J. A. Crotinger, L. J. Perkins, C. J. Solomon, E. A. Chaniotakis, J. P. Freidberg, J. Wei, J. D. Galambos, and J. Mandrekas. A "SuperCode" for systems analysis of tokamak experiments and reactors. *Fus. Technol.*, 21(3):1749–1758, May 1992.

[7] Kelly A. Barrett, Yu-Hsing Chiu, Paul F. Dubois, Jeff F. Painter, and Zane C. Motteler. Running a Basis Program; A Tutorial for Beginners. Technical Report UCRL-MA-118543 Part I, Lawrence Livermore National Laboratory, Livermore, CA, 1995.

[8] Paul F. Dubois and Zane C. Motteler. Basis Language Reference Manual. Technical Report UCRL-MA-118543 Part II, Lawrence Livermore National Laboratory, Livermore, CA, 1994.

[9] Yu-Hsing Chin and Paul F. Dubois. EZN User Manual. Technical Report UCRL-MA-118543 Part III, Lawrence Livermore National Laboratory, Livermore, CA, 1994.

[10] Yu-Hsing Chin and Paul F. Dubois. EZD User Manual. Technical Report UCRL-MA-118543 Part IV, Lawrence Livermore National Laboratory, Livermore, CA, 1994.

[11] Paul F. Dubois and Zane C. Motteler. Writing Basis Programs; A Manual for Program Authors. Technical Report UCRL-MA-118543 Part V, Lawrence Livermore National Laboratory, Livermore, CA, 1994.

[12] Paul F. Dubois. The Basis Package Library; A Manual for Program Authors. Technical Report UCRL-MA-118543 Part VI, Lawrence Livermore National Laboratory, Livermore, CA, 1994.

[13] A. H. Boozer. Ohm's law for mean magnetic fields. *J. Plasma Phys.*, 35(1):133–139, 1986.

[14] D. J. Ward and S. C. Jardin. Modelling the effects of the sawtooth instability in tokamaks using a current viscosity term. *Nucl. Fusion*, 29(6):905–914, 1989.

[15] H. L. Berk, T. K. Fowler, L. L. LoDestro, and L. D. Pearlstein. Hyper-resistivity theory in a cylindrical plasma. Working paper, 2001.

[16] Paul DuBois. *Software portability with imake.* O'Reilly & Associates, Inc., 1996.