ORBIT User Manual

Version 1.10

J.D. Galambos, J.A. Holmes, D.K. Olsen

SNS Building MS-8218, Oak Ridge National Laboratory
104 Union Valley Road, Oak Ridge, TN 37831-8218, U.S.A.

A. Luccio, J. Beebee-Wang

Brookhaven National Laboratory
Upton, NY.

July 16, 1999

SNS/ORNL/AP TECHNICAL NOTE

Number 011, Rev. 1

Acknowledgements

This code was written with the help from many people. Many of the specific actions of the macro-particle ‘injection’ nodes follow methods from ACCSIM, written by Fed Jones of TRIUMF . Many thanks to Scott Haney of Los Alamos National Laboratory for useful discussions on the macro-particle ‘herd’ architecture and for writing the SUPERCODE driver shell. Thanks also the Geoff Furnish for not dropping PLPLOT support. Finally thanks are due to the Spallation Neutron Source project for supporting work on this code.

CONTENTS

1. General Code Information

ORBIT (Objective Ring Beam Injection and Tracking) is a particle tracking code for Rings. This document is intended to be an introduction on how to use the ORBIT code. This includes a 1) description of the general code architecture, 2) a description of how to implementation specific features and 3) some examples of how to use features. As this is a user manual, documentation of the underlying physics is not provided here. Rather a description of how to use the code to simulate various problems is given.

1.1 Obtaining and Installation

ORBIT is available at http://www.ornl.gov/~jdg/APGroup/Codes/Codes.html . Links to the packages it uses are also on this page.

ORBIT is a C++ code and requires a C++ compiler to build. It is distributed as source. It has been built successfully with various versions of the GNU g++ compiler, as well as proprietary compilers. It also uses several additional freely available packages, as discussed below. Note: it is important to build the SuperCode driver shell (see below) with the same compiler as used to build ORBIT, to avoid name mangling problems when linking.

1.1.1 Starters

ORBIT uses a number of freely available packages. The SuperCode driver shell (see below) is required. A number of additional packages are optional, and provide additional capabilities. These packages should be built first. They are included in ORBIT during the build by previously defining the following environment variables, that should be defined in a logon script (e.g. .profile or .login file) :

CPU - Required. Set to the type hardware you are running on (you can just make up a name). This is useful when running on a system that file shares across different hardware architectures. In this case, be sure that CPU is appropriately defined to be distinct for each different hardware type at logon.

SUPERCODE_ROOT – Required. Set to the installation directory of the SuperCode shell (see below)

FFTW_ROOT – Required to use the FFT space charge implementations. Set to the installation directory of the FFTW library. See http://theory.lcs.mit.edu/~fftw/ for the distribution and documentation of this package.

PLPLOT - Required if you want to use interactive plotting. See http://www.mech.ubc.ca/Students/Computer/Software/Plplotdoc/node1.html for the distribution and documentation of this package.

If you think you want these optional features, build the packages first and define the associated environment vararible appropriately (be sure to export it if you use the ksh). Also, ORBIT uses the GNU utility `gmake` to build the code. This is often the default make utility on many systems (i.e. LINUX). Be sure it is installed on your system, and all references to 'make'in this manual, actually refer to the GNU make utility.

1.1.2 Build the Libraries

1.1.2.1 Build PLPLOT Library

This library is required for the interactive plotting capability. If it is not installed on your machine, ORBIT and SuperCode should build fine, but the plotting capability will not be available. After obtaining the PLPLOT lib from the above address, followthe instructions in the README and INSTALL file. Be sure to build it with double precision (i.e. with the –with-double flag) and without shared libraries (i.e. –without-shared flag). Then install it (preferably with root privilege in a normal place like /usr/local/plplot directory). Try building and running some of the ditributed examples to see if it's working (esp. the postscript and X-window drivers).

1.1.2.2 Build the FFTW Library

This package is used and needed for the FFT space charge routines in ORBIT. If you think you may want to use the space charge capabilities, build this package first. Get it from the above WWW address and follow the instructions to build it. It does not have an 'install' capability to copy the built libraries etc. to a common area, so you must either define FFTW_ROOT to the directory where you built this package, or else manually copy the library and header files to an appropriate common directory (e.g. /usr/local/fftw/…).

1.1.2.3 Build the SUPERCODE Driver Shell

If the SuperCode driver shell is already built on your system, just define your SUPERCODE_ROOT environment variable to be the installation directory, and skip the rest of this section.

The SuperCode driver shell must be built on the platform you want to run on. Start by defining CPU, and PLPLOT as described above. This shell has been built on Alpha, IBM, HP, SUN workstations, as well as LINUX running on PCs and Alphas[1]. Additionally, for this package you may want to define the optional environment variables, before building it:

READLINE_ROOT – directory for the libreadline.a and libtermcap.a libraries. These are standard on many systems, and are also available from GNU. Building with READLINE_ROOT defined permits interactive command line history and editing capability when running ORBIT.

[1] . The Makefile provided is for UNIX systems, but the shell has been built on Windows (with MSVC4.0) and Macintosh

GALIB – The installation directory for the GALib genetic algorithm optimization class library, see http://lancet.mit.edu/galib-2.4/ for the source . This allows use of the genetic algorithm optimizer in the Solver module. It is unlikely you'll need it for ORBIT however.

With these environment variables defined, get the SuperCode distribution (i.e. the sc.tar tar file) and untar it by typing

```
tar –xvf sc.tar
```

and then change directories to the newly created SuperCode directory. Then type

```
make configure
```

Then be sure there is a file Config/$(CPU).mak, where CPU is whatever you decided to call your CPU environement variable. There are a number of example ".mak" files you can use. Edit this file appropriately to define the compilers on your machine, and flags you wish to use for building. Also check to ensure there is a file called SCL/$(CPU).config . This file contains all device dependent source code. Again find an example that looks close to the configuration you are using and try using it. Both of these files are used later in the ORBIT build.

Then in the SuperCode directory, type "`make`". After SuperCode is built, be sure SUPERCODE_ROOT is defined to be the directory where you want the shell installed, and type "`make install`". You must have root privilege if you want to install it in a normal place like /usr/local/SC .

Documentation for this driver shell is provided in the SuperCode/Docs/SCManual.ps file. Warning: the section about support for FORTRAN modules is obsolete.

1.1.3 Building ORBIT

Next untar the ORBIT distribution (ORBIT.tar) file by typing

```
tar –xvf ORBIT.tar
```

and move to the newly created ORBIT dirtectory. Then type '`make configure`' and next '`make`' to build. The ORBIT executable will be put in $(CPU)/ORBIT, where $(CPU) is the predefined environment variable. Be sure your PATH environment variable includes this directory.

1.2 Running ORBIT

1.2.1 Starting ORBIT

Make sure that the executable you created in the step above, is in your PATH environment variable, relative to wherever to want to work with it. Then just type "ORBIT" to start it up. You

can not only run the accelerator routines from here, but you can also do the general programmable script stuff described in the SUPERCODE documentation.

1.2.2 Input files

ORBIT will automatically read in the file ORBIT.sc on startup, if it exists in the directory you are running in. An example ORBIT.sc file is distributed with the code, which includes some general settings and routines. Be sure to comment out the plotting stuff if you did not build your Shell with the interactive plotting. This is not meant to be a problem specific input file.

Usually, a run will be configured by appropriate settings in an input script file. You can have this input file read at startup by including its name as the first argument to the executable ORBIT, e.g.: typing

```
ORBIT case1.sc
```

will start ORBIT, ORBIT will then read the ORBIT.sc file if it exists, and will then look for the specified file `case1.sc` to read. Input files can be read from the interactive prompt by typing:

```
include "case1.sc" //  read the file case1.sc
```

(the '//' and following text is treated as a comment, see Ref. [1]). Note that one input file can include others. For example, the file case1.sc could include a line like:

```
include "standardLatticeSetup.sc"  // do a common lattice configuration.
```

1.2.3 Output

Many of the examples in this manual refer to sending output to a stream. A stream is similar to an output "unit" in FORTRAN. Streams can be created on the fly by the user. For example in an input file a user-defined stream called "`fio`" to a file called "`fileout`" can be created on the fly by

```
OFstream fio("fileout", ios::out);
```

This stream will delete any previously defined file called "`fileout`". The command

```
OFstream fio("fileout", ios::app);
```

is similar, but will append the prescribed output to whatever exists (if anything) in a file called "`fileout`". Many output routines require a stream type argument. Output can be sent to the console with the predefined `cout` or `cerr` streams, e.g.:

```
cout << "Echo this message\n";
```

See the SuperCode manual [1] for more information on inputting/outputting information from the Shell.

1.2.4 Quiting

Just type “quit” and hit return. This can also be included in script files as well.

You can suspend whatever is happening by hitting ‘control-c’. On some systems, doing this multiple times also kills the run, so be careful.

1.3 The SUPERCODE Driver Shell

The ORBIT code is written in C++, and operates using the SUPERCODE [1] driver shell. Before discussing any specifics of the ORBIT code itself, a few words are useful to explain the relationship between user provided “physics” modules and the driver shell. Conventional scientific programs typically have a prescribed flow of logic originating in a main program. Often there are several program flow choices, governed by appropriate choice of input variable settings. The program execution remains in compiled code until program completion. If the user desires some new flow logic, an edit-compile (debug) programming cycle is required.

On the other-hand, the SUPERCODE is a programmable driver shell, which can execute interpreted script files as well as compiled “physics” modules. Generally runs are done by reading in a “script” input file. These script input files are more general than typical “namelist” files which simply assign values to variables. Since SUPERCODE is a programmable interface, calling sequences to compiled code can be customized within a script “input” file (without recompiling). In fact there is no fully compiled set of logic to do a run. Rather the “general purpose” workhorse physics modules are compiled as a “tool-set”, and the calling sequences, looping, initial variable settings etc. are done through the shell. Additionally, routines can be created on-the-fly in script files and included in the customized run sequence. This driver shell is described in detail in Reference [1].

Both interpreted and compiled code have their place. Compiled code is much faster in general. Compiling everything however leads to code clutter (i.e. one-off numerical experiments, or scans that are never used again, along with variables to control them). The general philosophy for the separation of interpreted and compiled code is: code that is execution intensive, and/or general purpose is compiled. Examples of compiled code would be particle transport through a matrix multiplication operation. Code that is problem specific and generally not called often during a run is interpreted. Interpreted code examples are variable initialization, setting up “one-off” parametric scans, etc.

1.3.1 SUPERCODE Driver Shell Modules

SUPERCODE comes with a number of general purpose modules (see [1]). These include: (1) optimization (calculus based and genetic-algorithm[2]), (2) Probabilistic risk analysis (or uncertainty

[2] Requires installation of the GALib library on your system. See http://lancet.mit.edu/galib-2.4/

analysis), (3) Mathematical tools (splines, B-splines, random numbers, Bessel functions, …), (4) interactive plotting[3], and (5) parallel processing[4]. (Note - the shell plotting capability provides rudimentary, X-Y and 3-D capabilities, and is intended to provide a quick diagnostic capability, not an exhaustive plotting capability). Note also that a number of these features require special libraries to be installed on your system as discussed in section 1.2. All of these libraries are free, and generally run on most UNIX platforms. These general purpose modules are described in Ref. 1. The Shell is also buildable on Mac and PC OS's. for those so inclined.

1.3.2 Module / Shell Relationship

The procedure for actually constructing "physics modules" to be run with the SUPERCODE shell is described in Ref.1[5]. When adding a new module in practice, it's usually sufficient to mimic the method of an existing physics module. The general relationship between the user supplied physics modules and the driver shell is shown schematically in Fig. 1.1.

The Module Descriptor File is a useful place to get information about a module. It lists all the variables and routines that the Driver shell knows about in that particular Module. This is the place where these quantities should be documented. Variable quantities in these files can be manipulated from the Shell. The routines in the Module descriptor files can also be called from the Shell. In fact, putting together a set of variable initializations, and routine calls in a script file is how a "run" is typically done.

Note that the driver shell can not directly access class members. To access class members from the Shell (manipulate, view, etc.), a Module routine must be written (and compiled) to perform the appropriate manipulation. Examples where this may be done would be a routine to create a macro-particle, or a routine to dump macro-particle information to a stream. These compiled routines could then be called from the Shell, and the actual class member manipulations would be performed in the compiled routine.

[3] Requires installation of the PLPLOT and TCL/TK libraries. See http://www.mech.ubc.ca/Students/Computer/Software/Plplotdoc/node4.html

[4] Requires installation of the PVM software library on your system (see http://www.netlib.org/pvm3/book/pvm-book.html)

[5] FORTRAN modules are no longer supported.

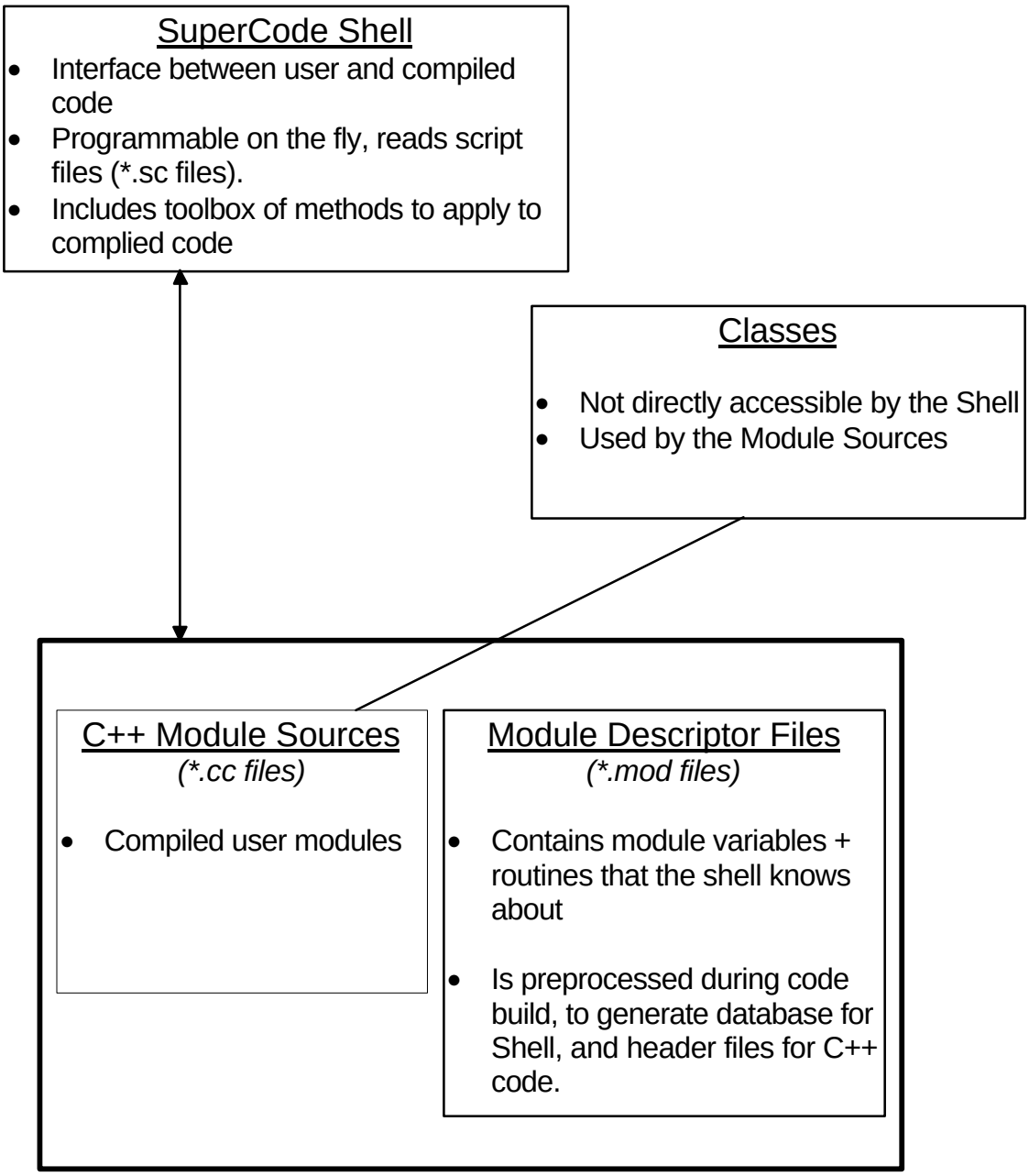


Figure 1.1 General relationship between the physics modules and the SuperCode driver Shell.

1.4 Units

Unless stated otherwise, MKS units are employed. There are a number of notable exceptions however. Macro-particle coordinates are in mm for distance and mrad for angles. Also the particle energy is in GeV, and RF and space charge voltages are in kV. Generally the units of variables are documented throughout the code. Every module member has a description with should include its unit (if any). The description can be found by looking in the module descriptor file (*.mod), or for example by typing in the shell:

```
ORBIT[1]:  about("lRing");  // find out what lRing is.
```

The result:

```
//  Total ring length [m]
static Real Ring::lRing;
```

2. Class Hierarchy for the Ring and Macro-particles

In thinking about the tracking of macro-particles around a ring, two main ideas come to mind: the macro-particles themselves and the Ring elements which operate on them in various ways. These are each represented as distinct classes, and much of the code is based on these. As such they are described in some detail here. We stress that the class structure adopted here has separated the macro-particle information container (MacrPart class) and actions done on macro-particles (Node base class) into two separate class structures. The connection between these classes is that the Node classes require a reference to a MacroPart herd. This separation facilitates the easy tracking of multiple herds through a single ring, which, for example, is useful for tracking prescribed test particles in the presence of a main herd.

We reiterate that the actual user implementation of these classes is done via "modules". These modules contain the user interaction mechanisms for instantiating objects, performing member function calls, etc. The modules and actual use of the classes are described later in Section 3.

2.1 MacroPart and SynchPart Class

There is a separate `SyncPart` class to hold the synchronous particle information. A synchronous particle must be instantiated before any macro-particles can be created. This synchronous particle[6] class is described in Table A.1.

The `MacroPart` class is described in Table A.2. It is a simple container class to hold information specific to macro-particles. A `MacroPart` object actually contains information about a "herd" (or

[6] Presently, only one synchronous particle is allowed, but it is straightforward to extend this to multiple synchronous particles, if ever needed.

array) of an arbitrary number of individual macro-particles[7]. When a macro-particle object is instantiated, the "herd" size must be specified. If the initially specified herd size is exceeded in the course of a calculation, the storage is automatically resized, but the subsequent execution may not proceed as efficiently. Most of the information held for particles is self-evident. Of particular interest is that each macro-particle object contains a reference to a "synchronous particle" object.

2.2 Node Class

As the macro-particles circulate around the ring, different operations will be performed on/with them. For instance they may undergo a transfer matrix operation, or a space-charge kick, or an information dump to a stream (or file). Each of these actions is performed at a "node" of the Ring. The `Node` class represents the common set of features such operators have. This is an abstract class, with the general-purpose information, common to all nodes in the ring. As this is an abstract class, no `Node` objects will be created, but rather this class will be inherited by sub-classes that will have actual objects (for example a Transfer-Matrix class). The `Node` class is described in Table A.3. Most of the `Node` members are self-evident, but a few require additional discussion.

The `_oindex` member is used for ordering the nodes into the calling sequence around the ring for which the operations will be done on macro-particles. The calling sequence is done in ascending order of the `_oindex` value of each node. Whenever a node is instantiated, an `_oindex` value is required. Before any actual ring calculations are done, the nodes are automatically sorted into the order inwhich the calculations should be done (in the `Ring::initNodes` routine) The order is in ascending value of `_oindex`. The `_length` member is the length of the node (which can be 0). Each node presumable should do some operation with the macro-particles. Two `Node` member function hooks are provided for this: (1) a `_nodeCalculator` , and (2) an `_updatePartAtNode` routine. Both these routines are automatically called for each node included in the ring in the order of their `_oindex` values. The first routine, `_nodeCalculator(macroPart &h),` is a place where preliminary calculations can be done on herd "`h`", which may depend on more than one macro-particle. For example, calculating the potential resulting from the ensemble of macro-particles could be calculated here for a space-charge kick node. The second routine `updatePartAtNode(macroPart &h)` is provided to operate on individual macro-particles from herd "`h`". For example, a "Transfer matrix" node could advance each individual macro-particle through a transfer matrix here. Note that either of these routines can be omitted if not needed.

2.2.1 Derived Node Sub-Classes

As mentioned above, the Node class is an abstract class. It is inherited by the actual Node sub-classes. These derived classes perform the actual calculations needed to transport the macro-particles around the ring, and perform other manipulations on them. The mechanism for implementing these classes is generally to create a module for each class. While the class contains the routines that do the calculations, the "module" members contain the interface mechanisms between the class objects and the driver Shell. Examples of Node sub-classes are: 1) a

[7] This approach is taken to allow contiguous memory allocation for a large number of macro particles. This appears to help execution performance for cache limited CPUs.

TransferMatrix class for transporting macro-particles around the Ring, (2) a LSpaceCharge class to give longitudinal space charge kicks, and (3) Acceleration class to give RF voltage kicks.

When a Node sub-class is added, a constructor should be supplied that at a minimum accepts input for the Node name and the _oindex value specifying where in the ring the Node calculator is called. Also, at least one calculator should be supplied (either a nodeCalculator or updatePartAtNode), or else the Node would not do anything! With these class members defined and declared, we know both "where" each node should is called as we work our way around the ring and "what" it does.

2.3 Diagnostic Classes

A number of diagnostic classes are provided for use in analyzing beams. The actual implementation and practical application of these is described in the Diagnostic part of Section 3. Here we only describe the general features of these classes, as they are distinct from the Node and MacroPart classes described above. There is a base diagnostic class (DiagnosticBase) which is described in Table A.4. The constructor for the derived diagnostic sub-classes is required to have a reference to a MacroPart input to it. This is the identifier for the herd to do the diagnostics on. Each derived Diagnostic class defines its own calculation (_diagCalcuator()) and can optionally provide a routine to dump "human" readable diagnostic results to a stream (_showDiagnostic(Ostream &sout)) or dump data to a stream for post-processing (_dumpDiagnostic(Ostream &sout)). Examples of derived diagnostic classes are beam moments, beam canonical coordinates, and beam emittances. Some of these classes are used in nodes in the code, for example in calculating the emittance for printing beam statistics at regular intervals.

3. General Modules

Note: This section is not an exhaustive user manual. See the appropriate Module descriptor file to get detailed descriptions of the meanings, variable units, and routine implementations etc. Also look in the module source for a definition of the input / output to routines.

3.1 The Particle Module

3.1.1 SynchParts, MacroParts and herds

This module contains the interfaces between the user/shell and the MacroPart class. Member functions in this module offer a window into the MacroPart class. Before actually creating any particluar macro-particles, the user must first create a synchronous particle (see the Particle::addSyncPart routine) and also create a macro-particle "herd" (see the Particle::addMacroHerd routine). Presently only one SyncPart object can be created. For example, the following script creates a herd to store macro-particles:

```
////////////////////////////
// Make a synchronous particle:
////////////////////////////

  Real TSync = 1.;       // Kinetic Energy (GeV)
  Real mSync = 1;        // Mass (AMU)
  Real charge = 1;       // charge number
  addSyncPart(mSync,charge,TSync);
  mainHerd = addMacroHerd(12000);
```

This example creates a 1 GeV proton synchronous particle. Then it creates a herd (identified by the predefined Integer `mainHerd`) with storage vectors initially sized at 12,000. Note that none of the actual macroparticles have been initialized yet, only storage space has been allocated. The `mainHerd` is a special herd, which is the default herd for many operations. But it also possible to create an additional separate herd, for example with

```
  Integer testHerd = addMacroHerd(100);
```

Test particles could be introduced into `testHerd` at some point, and tracked independently from the `mainHerd`. One of the `MacroPart` herd members is the switch `_feelsHerds`. Herds are created with a default value of 0, which means it only interacts with itself via space charge. If this value is switched to 1 by using:

```
   setHerdFeelLevel(testHerd,1);
```

it will feel, but not push the herd `mainHerd`. This is useful for tracking a set of test particles in the presence of the herd `mainHerd`.

3.1.2 Adding macro-particles

Now that we have a herd set up, we need to be able to actually put some particles in it. The `addMacroParticle` (or `addMacroParticle2`) routines allow direct addition of a single macro-particle with specified values. For example, one could do:

```
  Real x,xp,y,yp, dE, phi;

  x = 1.;xp = 0., y=0., yp=1.; // mm, mrad
  addMacroPart(x, xp, y, yp);
  x = 2.;xp = -1., y=1., yp=-1.; // mm, mrad
  addMacroPart(x, xp, y, yp);
```

to add 2 particles to the mainHerd, with 0 for the longitudinal *DE* and f coordinates. One may also specify a specific herd to add particles to with:

```
  addMacroPartBase(mainHerd, x, xp, y, yp);  , or
  addMacroPartBase(testHerd, x, xp, y, yp); .
```

To also specify the macro-particle longitudinal coordinates, use

```
  x = 1.;xp = 0., y=0., yp=1.; // mm, mrad
  dE = 0.001; phi=90.;  // GeV, degrees
```

```
addMacroPart2(x, xp, y, yp, dE, phi);
addMacroPartBase2(testHerd, x, xp, y, yp, dE, phi);
```

These routines are the core means of adding particles to a herd, and can be called directly from the shell as shown above. However it is too cumbersome to actually directly use these routines to create a herd of say 100,000 particles. In practice, they are more often called automatically from routines that sample from a prescribed distribution type (see the Injection Module below), or by using a feature to read from a file. For example, using:

```
readParts(mainHerd, "filename", nParts);
```

will read `nParts` macro-particles from the file "`filename`" into the `mainHerd`. The format of the file is one data set per line, and each line containing the values for *x*, *x'*, *y*, *y'*, *DE* and f (in mm, mrad, GeV and rads) with entries delimited by whitespace. The actual format of the numbers themselves is not important. If the file contains less than the specified number of macro-particles, only as many as are contained in the file are read.

3.1.3 Getting macroPart information

Since the Shell does not have direct access to class members, routines are provided here to dump macro-particle information to forms accessible by the Shell. `nMacroParticles` specifies the number of macro-particles in the `mainHerd` instantiated so far. The routines `xVal(i), yVal(i), xpVal(i), ypVal(i), deltaEVal(i),` and `phiVal(i)` return the *x, y, x', y', DE* and f values for a single macro-particle number number "i" in the `mainHerd`. For example,

```
Real x1 = xpVal(11); // x1 contains the x' value of the 11th
                     // macro-particle in the mainHerd.
```

A similar set of routines exists to get particle coordinate information for an arbitrary herd. For example:

```
Real x2, yp3;
X2 = xValBase(2, testHerd); // x2 now contains the x value
                            // of the 2nd particle in herd testHerd
yp3 = ypValBase(3, testHerd);  // yp3 now contains the y' value
                               // of the 3rd particle in herd testHerd
```

The routines `xVals(v), yVals(v), xpVals(v), ypVals(v), deltaEVals(v)` and `phiVals(v)` dump the specified macro-particle information for the `mainHerd` into a vector "`v`" (which is automatically sized). For example,

```
RealVector xx;
xVals(xx);  // xx now contains all nMacroParticles x values of the mainHerd
```

It is possible to dump the particle coordinates to a file (in a format compatible with the `readParts` command discussed above) with the `dumpParts` routine. This routine dumps the values for *x, x', y, y', DE* and f (in mm, mrad, GeV and rad) to a specified stream. Each of the particles' coordinates are on a separate line. For example:

```
Ofstream fio("particle.dmp", ios::out); // create a stream to a new
                                        // file called "particle.dmp"
dumpParts(mainHerd, fio); // dump the coordinates of the mainHerd
                          // macro-particles to stream fio
```

It is also possible to calculate and send the extrema information (maximum /minimum particle coordinates) of a herd to a specified stream. For example,

```
OFstream fio("case1.out", ios::app);  // create a stream to an
                                      //existing file called "case1.out"
showExtrema(mainHerd, fio); // dump the extrema of the mainHerd
                            // particles to stream fio
showExtrema(mainHerd, cout); // dump the extrema of the mainHerd
                             // particles to the screen
```

Finally, a formatted listing of all the herd macro-particle parameters can be sent to a stream with the `showMacroHerd` routine. For example,

```
showHerd(cout, testHerd);
```

will send a listing of the particle parameters in `testHerd` to the console. Be careful, doing this with a large herd could dump a lot of screen-fulls.

3.1.4 Lost macro-particles

The `macroPart` class contains a `LostMacroPart` member. The `LostMacroPart` class contains information about each macro-particle that is lost during tracking (for example at an aperture). The member function `_addLostMacro` does the transfer of information from an active macro-particle to a lost macro-particle.

3.2 The Ring Module

The Ring module is a general module, which controls the overall execution flow for the macro-particle tracking. Although it has no classes itself, it contains some tracking initialization and "hooks" that actually perform the calls to the Node class calculators. Some of the important routines from this Module are described below.

The `initRing` routine performs some general initialization calls. Importantly, it sorts all Nodes which have been added to the ring by ascending order of the "`oIndex`" value[8]. Note that in an input file it is not necessary to create ring nodes in the order you want them to appear in the ring. Their order is determined by their "`oIndex`" value. This routine does not need to be called by the user, and is automatically called before any tracking is done, if needed.

[8] It's a good idea to include some "padding" when assigning an oindex value to a Node. For example you might use intervals of 10 or so, to allow "space" between elements in which to insert more nodes later.

3.2.1 Doing turns

The `doTurn(Integer &nt)` pushes the herd `mainHerd` a specified number of turns around the ring. It takes an `Integer` argument specifing the number of turns to do. This routine loops over all active nodes. For each node it first calls the appropriate `_nodeCalculator` routine and then calls the `_updatePartAtNode` routine. For example, the following command will cause the `mainHerd` to be pushed 100 turns:

```
doTurn(100);
```

A related routine is `doTurnBase(Integer &herd, const Integer &nt)`. This routine causes a specified herd to be turned a specified number of turns; for example:

```
doTurnBase(mainHerd, 100);
```

causes the herd `mainHerd` to be tracked for 100 turns (same as the previous example). Another example:

```
doTurnBase(testHerd, 2);
```

causes the herd `testHerd` to be tracked for 2 turns, without modifying the herd `mainHerd`. A number of other routines are available for tracking a herd to an arbitrary ring node. These are the `turnToNode` and `finishTurn` routines. The routine `turnToNode` will track a specified herd to a specified ring node. For example, the call:

```
turnToNode(mainHerd, 23);
```

will cause the `mainHerd` to be tracked up to, and including the action of the 23rd node. The first argument is the integer herd identifier, and the second argument is the integer node number. To determine the node number you want to track to it may be useful to call the Output module routine "`showRing(cout)`" to list the ring nodes to the console The node number is in the first column of this output. Once at the desired ring node, other actions can be done, for example plotting the herd, dumping its contents to a file, performing some diagnostic on it, etc. This action can be continued to arbitrary nodes of increasing magnitude (farther along the ring). The turn can be finished by using the `finishNode(mainHerd);` command.

Occasionally it may be necessary to push multiple herds around the ring together. The routine `turnHerds(Integer & nt)` is provided for this purpose. This routine turns all the defined herds a prescribed number of turns around the ring. The outer loop is on the nodes, and the inner loop on the herds.

3.2.2 Calculating the longitudinal separatrix

The `calcBucket` routine is a special routine to track "a bucket particle", which can be used to find the separatrix bucket boundary in longitudinal space. It takes two arguments, (1) an initial energy (GeV) and, (2) an initial angle (deg). The calculated energy and angle coordinates for the

bucket are stored in the vectors `bucketDE` (GeV) and `bucketPhi` (deg) respectively. These vectors are automatically dimensioned as needed.

3.2.3 Longitudinal only tracking

Most of this manual applies to the case for both transverse and longitudinal tracking. But it is possible to do longitudinal only tracking by specifying "`longTrackOnly = 1;`" . In this case the lattice input discussed in 4.1 is not needed (and should not be used). Also be sure to include an Accelerating node (see section 4.3) and optionally a longitudinal space charge node (see section 4.4). See the decription in section 4.3 for specifying the ring parameters for the case of longitudinal only tracking.

3.3 Diagnostic Module

This module implements diagnostic class objects in routines that are accessible from the shell. The classes are declared in file DiagClass.h. Some of these are implemented so that they can be called directly from the shell, and provide information about some aspect of a herd. Other diagnostics are implemented as ring Nodes, which can be called regularly as the beam traverses the ring.

3.3.1 Stand-alone diagnostics

3.3.1.1 Emittance

The emittance of a herd can be calculated with the routine `calcEmitBase(Integer &herd)`. This routine accepts a reference to the herd identifier. It calculates the horizontal and vertical emittance of each herd member (stored in the vectors `xEmit` and `yEmit` respectively). Additionally, some statistical quantities are stored:

`xEmitRMS` = horizontal RMS emittance of the herd (π-mm-mrad)
`yEmitRMS` = vertictal RMS emittance of the herd (π-mm-mrad)
`xEmitMax` = maximum horizontal emittance of the herd (π-mm-mrad)
`yEmitMax` = maximum vertictal emittance of the herd (π-mm-mrad)

The emittance distributions can be binned by defining the number of bins (`nEmitBins`) and the bin width (`deltaEmitBin`) in π-mm-mrad, and calling the routine `binEmit()`. This routine stores the percentage of the beam with horizontal emittance $\geq$ `i * deltaEmitBin` in `xEmitFrac(i)`. Similarly the vertical information is stored in `xEmitFrac` and the percetange of beam with horizontal or vertical emittance $\geq$ `i * deltaEmitBin` is stored in `xyEmitFrac`. For example the emittance information of the main herd could be calculated with:

```
nEmitBins = 50; deltaEmitBin = 2.; // Bin from 0 to 100 pi-mm-mrad
calcEmitBase(mainHerd);        // Calculate the main herd emittance
cerr << "X RMS Emit = " << xEmitRMS << "\n";
cerr << xEmitFrac;  // Dump the vector of the horizontal emittance
                    //  distribution to the console.
```

A simpler way to get these calculations done for you, and have the results printed in a readable format to a stream, use either:

```
showEmitBase(mainHerd, cout);  // this routine dumps emittance
                               // information for any specified herd.
showEmit(cout);                // this only works for the mainHerd.
```

3.3.1.2 Canonical Coordinates

The canonical coordinates of a particle are the coordinates in the transformed coordinate system, in which the betatron oscillations are circular. These are used in many of the diagnostic calculations, but may also be of interest to the user. Executing

```
OFstream fio("test.dat", ios::app)  // get a file ready to append to.
dumpEAndCC(mainHerd, fio);  // dump the tunes and canonical coordinates
```

dumps the following quantities on a separate line for each member of a herd to the prescribed stream: horizontal emittance (π-mm-mrad), : 2) vertical emittance (π-mm-mrad), 3) canonical x coordinate (mm), 4) canonical y coordinate (mm), 5) canonical x momentum (mrad), 6) canonical y momentum (mrad), and 7) $\delta p/p$.

3.3.1.3 Action Variables:

The action variables (i.e. Courant-Snyder actions) of a herd can be calculated and dumped to a stream. For example:

```
calcActions(mainHerd); // calculate the action variables for each
                       // macro-particle in the mainHerd.
```

The action variables are stored in the vectors `xAction` and `yAction` (unitless). The quantities can be calculated and dumped to a file with:

```
OFstream fio("Actions.dat", ios::out)  // create a file.
dumpActions(mainHerd, fio); //  calculate and print the action variables
                            // for each particle in the mainHerd.
```

The dump file has the following quantities on a separate line for each particle in the herd: 1) horizontal action, 2) the vertical action. See the description below of the `dumpTAndA` routine in the Transfer Matrix module also, which dumps both the actions and tunes of each particle.

3.3.1.4 Moments

Moments of the beam can also be calculated. First the average center of the beam is calculated ($\langle x \rangle$, $\langle y \rangle$). The horizontal i^{th} moments are calculated as $\langle (x - \langle x \rangle)^{i-1} \rangle$, and similarly for the vertical moments. The moments are stored in the Moment class Matrix member `_momentXY(i,j)` which holds

$\langle (x - \langle x \rangle)^{i-1} \rangle * \langle (y - \langle y \rangle)^{j-1} \rangle$

We also store the moments normalized by $(b_x)^{(i-1)/2} * (b_y)^{(i-1)/2}$ in the Moment class Matrix member `_momentXYNorm(i,j)`. Note that i(j) = 3 actually represents the 2nd order moments and i=1 represents the 0th order moment. These moments can be calculated and dumped to a file with:

```
showMoments(mainHerd, 3, fio); // calculate up to 2nd order moments
                               // of the main herd and dump
                               // them to stream fio.
```

The format of the moment dump is to first print the order number, and then print the matrices `_momentXY` and `_momentXYNorm` in Mathematica format. These moments may also be applied as a Ring node (see below), in which case the output is printed every turn to a single line (see below).

3.3.2 Diagnostics Nodes

Any of the above diagnostics can be used interactively at any time. They can be used repetitively by including them in a loop. However, sometimes it is useful to be able to easily get diagnostic information about a beam many times as it traverses the ring a single turn ("high frequency" diagnostics) . Use of the above functions to do this would be tedious. Therefore, a special derived `Node` class (`DiagnosticNode`) is used to facilitate easy implementation of many diagnostics about a ring. The `DiagnosticNode` class is itself also a base class, and classes derived from it (1) perform a diagnostic calculation, and (2) print the diagnostic results to a file. Special routines are provided to automatically add these nodes after each transfer matrix in the lattice being used[9].

The Diagnostic nodes can be added anytime during a run. By default they are inactive, and are skipped over until they are activated. After they are used, they may subsequently be deactivated, and "normal" tracking calculations can be continued. These nodes can be activated and deactivated as much as needed, but use caution with them, as large amounts of data may be output when they are active.

3.3.2.1 Moment Nodes

The beam moments described above can be printed at an ring azimuthal position by adding a "`momentNode`" in the appropriate location. For example:

```
// stick a moment node at node position 22,
// calculating beam moments up to order 3:

addMomentNode("moment Node", 22, 4, "MomentDump.dat");

activateMomentNodes();    // activate this momentNode
doTurn(10);               // turn the mainHerd 10 turns
deactivateMomentNodes();  // deactivate this momentNode

 doTurn(100);             // track the mainHerd some more
                          // without dumping moments.
```

[9] To get diagnostics at finer intervals than the lattice transfer matrix representation, you must generate another lattice set, with finer spacing of transfer matrices.

Each line in the dump file (“ MomentDump.dat” in the above example) has: (1) the number of completed turns, (2) the azimuthal location of the present turn (m), (3) the integrated azimuthal distance the `mainHerd` has traveled since the start of the run (m), and then the beam moments in order: (1,1), (2,1), (1,2), (3,1), (2,2), (1,3), (4,1), … (4,1), …
Where the first index = (horizontal moment+1) and the second index = (vertical moment+1).

It is possible to add a set of moment nodes after each transfer matrix advance all around the ring with:

```
// stick a moment node after every transfer matrix in the ring
// calculating beam moments up to order 3:

addMomentNodeSet(4, “MomentDump.dat”);

activateMomentNodes();    // activate this momentNode
doTurn(10);               // turn the mainHerd 10 turns
deactivateMomentNodes();  // deactivate this momentNode

 doTurn(100);             // track the mainHerd some more.
```

If the transfer matrix advance is small compared to the betatron oscillation length, the output from this set of diagnostics is useful for looking at the oscillations in the moments of the beam.

3.3.2.2 Statistical Lattice Parameters

Statistical lattice parameters for a herd can be calculated using this diagnostic. The statistical lattice parameters are calculated using the moments of the canonical coordinate variables discussed above, along with the emittance. These nodes are called “StatLat” nodes and use the StatLatDiagnostic class. A single statistical lattice parameter node can be added using:

```
// stick a StatLatnode at node position 22,

addStatLatNode(“StatLat Node”, 22, “StatLat.dat”);

activateStatLatNodes();   // activate this StatLatNode
doTurn(20);               // turn the mainHerd 20 turns
deactivateStatLatNodes();  // deactivate this StatLatNode
```

Each line in the output file (“StatLat.dat” above) contains: (1) the turn number, (2) the azimuthal position around the ring (m), (3) the statistical horizontal beta (m), (4) the statistical vertical beta (m), (5) the statistical horizontal alpha, (6) the statistical vertical alpha, (7) the lattice horizontal beta (m), (8) the lattice vertical beta (m), (9) the lattice horizontal alpha, and (10) the lattice vertical alpha. It is possible to add a set of statistical lattice nodes all around the ring, to get “high frequency” lattice information dumped to a file, e.g.:

```
// stick a StatLat node after every transfer matrix in the ring

addStatLatNodeSet(4, “MomentDump.dat”);

activateStatLatNodes();    // activate this StatLatNode
doTurn(3);                 // turn the mainHerd 3 turns
```

```
   deactivateStatLatNodes();  // deactivate this StatLatNode

    doTurn(100);             // track the mainHerd some more.
```

3.3.2.3 Poincare Moment Tracking Nodes

Poincare Moment Tracking Nodes (PMTNode) are provided for tracking the coordinates of individual test particles, at every oscillation period of a prescribed moment of the `mainHerd`. This can be used to generate plots of islands and separatrices if the appropriate test particles are used.

This is only meaningful only if the transverse space charge calculation is on, and the lattice being used is fine enough so that the transfer matrix advance is shorter than the moment oscillation length. The procedure works by following multiple herds. First, after every transfer matrix advance, the requested moment of the `mainHerd` is calculated. If it is found to me a maximum, the coordinates of the particles of any other herd with member `_feelsHerd` set to 1 are dumped to the prescribed file. Since there should be a fine distribution of these nodes around the ring in order to "catch" the maximum of the moment oscillation, a full set of these nodes should be employed. As an example:

```
// Add a PMT set:

   addPMTNodeSet("PMTTest.dat", 2,1); //  track at horizontal 1st order moment
                                      // oscillations
// Make a test herd:

Integer testHerd = addMacroHerd(10); // small test herd
setHerdFeelLevel(testHerd, 1); // this herd only feels the mainHerd.
addMacroPartBase(testHerd, 10., 0.1, 5., 0.1);
addMacroPartBase(testHerd, 1., 1., 0.1, 1.);

  doTurn(2); // Turn the main herd a few turns:


// Turn both herds 3 turns with PMT's on:

  activatePMTNodes();
  turnHerds(3);
  deactivatePMTNodes();
```

The output in the file "PMTTest.dat" contains on each line: (1) the turn number, (2) the azimuthal position (m). Then on the same line, for each particle in the test herd being used : (3) the canonical horizontal position (mm), (4) the canonical vertical position (mm), (5) the canonical horizontal momentum (mrad), and (6) the canonical vertical momentum. This technique can create large output files if test herds with many particles are tracked.

3.3.3 General Diagnostic Node Status

One can get a status report of all the diagnostic nodes in the ring sent to a stream with the command:

```
   OFstream fio("Case_1_2_3.out", ios::out)  // create a file.
```

```
showDiagnostics(fio);
showDiagnostics(cout);  // show a list of all diagnostic nodes in the
                        // ring to the screen
```

This listing includes the status of each node's `_activeFlag`.

3.2 Parallel Runs

It is possible to run ORBIT in a parallel processing mode. The approach taken here is to launch multiple ORBIT processes on different CPUs, each parsing the same input script file. Most of the calculation proceeds independently on each processor, with the exceptions:

- Initialization of the random number seed used to generate macro-particles from prescribed distributions
- Space charge calculations
- Diagnostic calculations
- Output

Other calculations such as lattice matrix advances, aperture checks, thin lens kicks, etc. have no need for parallel communication and are exactly the same as for a serial run.

3.2.1 Parallel Nuts and Bolts

We use a message passing "master-slave" method for the parallel implementation here. In particular, PVM is used to do the message passing[10] (see reference 6). This package is free, and available for most computing platforms. It has been tested with ORBIT using version 3.4.0. The PVM package must be installed on your computer before you build ORBIT. Then define the environment variable PVM_ROOT to be the installed directory of PVM. Then build ORBIT. If you do not have PVM installed on your system, ORBIT will build fine, but the parallel capability will not work. Also, If you want to run parallel, it is advisable to have a system of CPUs connected with fast (at least 100 Mbs) switching. Parallel space charge calculations on work station clusters connected with 10 Mbs TCP will perform poorly.

Before starting a parallel run, the vitual parallel machine must be set up. This is done by running the pvm daemon. It is possible to construct a "hostfile" containing the names of all the nodes of the parallel computer, and the directories containing the ORBIT executable and the input script file (i.e. where you want to work). Here's an example hostfile (see the PVM manual for details):

```
* wd=/home/jdg/Accelerator/ORBIT/workParallel
ep=/home/jdg/Accelerator/ORBIT/LINUX
alice.sns.ornl.gov
cheshirecat.sns.ornl.gov
```

[10] This package is also used in the driver shell to facilitate parallel optimization and uncertainty analysis. However, the implementation here is independent of that used in the driver, in part to make it easier to switch to MPI if we decide to later.

```
whiterabbit.sns.ornl.gov
caterpillar.sns.ornl.gov
madhatter.sns.ornl.gov
```

The parallel calculation initiation routines are in the Parallel module. This module contains routines for parallel calculation set up and spawing of the child processes. However, most of the actual parallel message passing is in the space charge and diagnostic/output routines. If the calculation is a parallel one, the switch `Parallel::parallelRun` is automatically set to be true (1), and message passing synchronization routines are implemented when needed in the space charge, diagnostic and output modules.

Some parallel settings and output of interest are

User input:

dataEncodeType: set to 0 for full encoding, required for use on heterogeneous parallel computers. set to 1 for no encoding, faster communication and ok for homogeneous parallel computers.

spawnOnParent – swith to contol whether to spawn a child on the same CPU as the parent. Default setting is 0 (don't do it). Set ==1 to spawn a child process on the parent CPU.

Set by ORBIT:

nJobs - This is set in the startParallelRun routine, and contains the number of parallel jobs started.

iAmAChild – == 1if a child process, == 0 if it's the parent process.

Also note that timing a run is different for a parallel run. A useful indicator of run time is the wall clock time. An example of getting this is shown in the example in the section below.

3.2.2 Parallel Calculation Flow and Implementation

3.2.2.1 General parallel calculation flow

Parallel runs must be run with an input script file. The basic structure of a parallel input script is shown schematically in Fig. 3.1. An example script file is shown below also, with the parallel calculation dependent pieces highlighted. The parallel calculation is started by having ORBIT read an input file that is set up to do a parallel calculation, e.g., by typing

```
ORBIT parallel.sc
```

The original process started in this manner will be the parent. Early in the input script, is a call to the `startParallelRun(String &name)` routine. The argument to this routine should be the name of the input file being parsed. This routine will spawn ORBIT on all the other machines active on the virtual machine (see above) and tell them to read the specified input file, which should be the same file as the parent read. Don't worry, if a child calls this routine it will not spawn more ORBIT process, causing an explosive growth of ORBIT processes. Now there are a bunch of ORBITs running on different processors, reading the same input script file. Note the call

to `syncSeeds()`in the example script below. This call ensures that each process uses a unique random number seed, so they do not all track the same herd!

The ring node structure set up is the same as for a serial run. The macro-particle setup is pretty much the same, except do not track any macro-particles on the parent (see the example below). The strategy taken in the present ORBIT parallization is to use the parent to corredinate all the message passing, and not spend time tracking. If you do not like wasting a CPU with no tracking, set the spawnOnParent to 1, and a child ORBIT process will be spawned on the same CPU as the parent.

You can have the calculation proceed as a serial calculation once the macro-particle and node structure is set up. For instance you could loop through some `doTurn` and `showTurnInfo` calls. The output routines are set up so that the parent gathers information from all the children, and only it sends information to a stream. Each process reads the same input script and follows the same calculation path prescribed in the script. Any node calculations that require communication between the child processes have appropriate message passing built in.

An example of a parallel calculation is shown below.

```
////////////////////////////
// Parallel stuff
////////////////////////////

 dataEncodeType = 1;   //   Raw - no encoding.
 startParallelRun("TargProfParallel.sc");
 syncSeeds();

cerr << "Done with Sync seeds, number of // jobs  = " << nJobs << "\n";

// set up file streams as usual
…

// set up macro distributions to sample from as usual
…

   nMacrosPerTurn = 0; // don't launch any macros anywhere!
   nMaxMacroParticles = 216*1158; // this is per node
   if(iAmAChild)
        nMacrosPerTurn = 216;  // if this is a child process launch 216
macros per turn.

   nReals_Macro = 2.0e14/(nMaxMacroParticles * nJobs); // different for
parallel run:

//Set up nodes as usual
…

String t;

if(!iAmAChild) {              // print the starting wall clock time
    dateTime(t);
    fio  << "Starting at " << t << "\n";
    cerr << "Starting at " << t << "\n";
}
```

```
 for (_i=1; _i<=24; _i++)  // Do 600 turns
 {
   doTurn(50);
   showTurnInfo(fio);  // dump turn info to a
   showTurnInfo(cerr);
  }

if(!iAmAChild) {              // print the final wall clock time
    dateTime(t);
    fio << "Done at " << t << "\n";
    cerr<< "Done at " << t << "\n";
}

// Final diagnostics / output as usual
…

quit
```

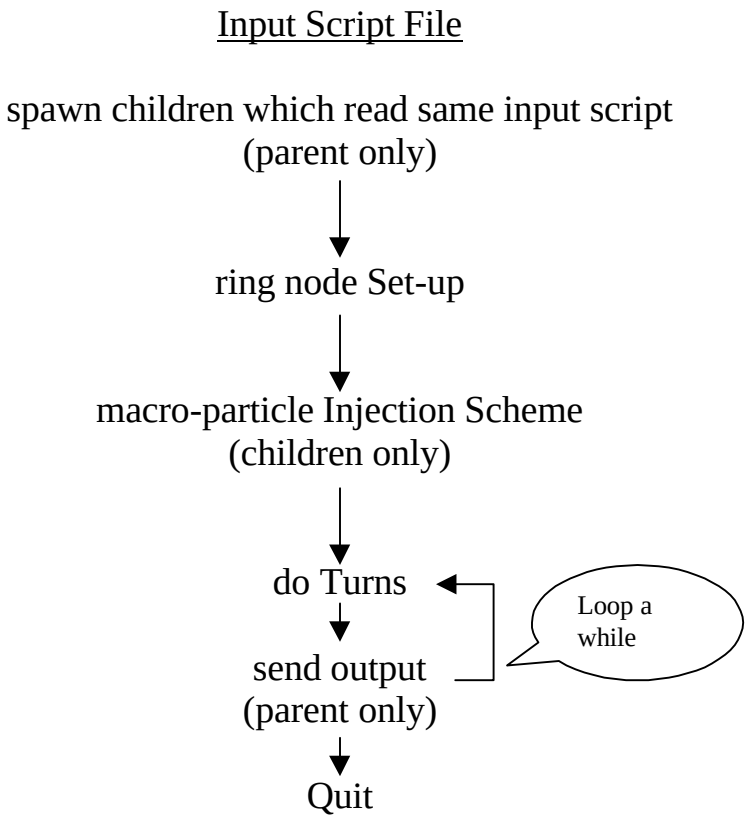


Fig 3.1 General flow of the input file script and calculation of a parallel run.

3.2.2.2 Transverse space charge calculation

The most time critical parallel calculation is the transverse space charge calculation, and as such is described briefly here. As herds are tracked in parallel around a turn on different processors, at each space charge kick node, the collective force must be gathered and communicated between

nodes. Typically this will happen 100's of times per turn. The strategy taken to parallelize this calculation is shown in Fig. 3.2. Note that each child calculates its own FFT of the Greens function and the global charge distribution. This is faster than waiting for one processor to do it, and passing the results.

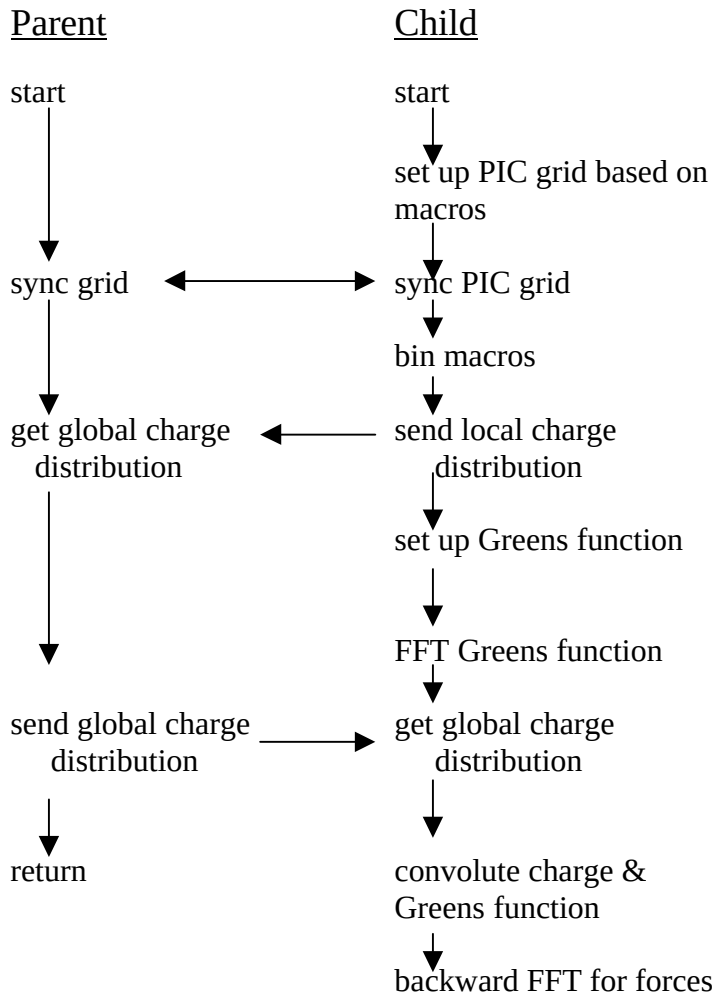


Fig. 3.2 Parallel flow logic for the transverse space charge calculation.

3.2.2.3 Capabilities NOT parallelized yet:

- All moment diagnostics
- StatLat diagnostics
- Cannonical coordinate diagnostic
- Poincarre moment tracking diagnostic

4.1 Transfer Matrix Module

Transfer matrices are a fundamental mechanism used to transport macro-particles from one point in a ring lattice to another. A `TransMatrixBase` class is provided to contain transfer matrix information. This is a base class, and there are two sub-classes, a first order transfer matrix (`TransMatrix1`) and a second order transfer matrix (`TransMatrix2`) At present, these matrices must be generated externally.

4.1.1 First order transfer matrices:

To directly add a first order transfer matrix to the list of ring `nodes`, the following routine from the `TransMat` module can be used:

```
Void addTransferMatrix(const String &name, const Integer &order,
    const RealMatrix &R,  const Real &bx, const Real &by,
    const Real &ax, const Real &ay, const Real &ex, const Real
    &epx, const Real &l)
```

Here `order` is the `_oindex` node value (i.e. where in the ring it is situated), `R` is the 6x6 1st order transfer matrix, `bx(y)` is the horizontal (vertical) beta value at the beginning of the element [m], `ax(y)` is the horizontal (vertical) alpha value at the end of the element, `ex` is the horizontal dispersion [m] value at the end of the element, `epx` is the horizontal dispersion derivative (m) value at the end of the element, and `l` is the length of the element [m].

4.1.1.1 Using externally generated transfer matrices:

A transfer matrix can be added directly from the Shell with a call to the `addTransferMatrix` routine, but this is a rather cumbersome process. Two routines are provided which read externally generated files to get the Transfer Matrix information for an entire ring:

```
Void readDIMADFile(const String &s1 )
Void readMADFile(const String &s, const String &s) .
```

The arguments of these routines are the names of the file to read. Each of these routines reads the specified file, parses it appropriately to find the transfer matrices, and makes the appropriate calls to the `addTransferMatrix` routine. The transfer matrices are assigned order indices starting with 10 for the first transfer matrix and incremented by 10 each following one. This allows room to subsequently insert additional nodes between the transfer matrix nodes.

4.1.1.2 Using DIMAD to generate the lattice transfer matrices:

The `readDIMADFile` routine expects to read a file produced by the code DIMAD [3], in the same format as that used by ACCSIM [2] to enter the transfer matrices. For example the call

```
readDIMADFile("SNSLattice.dmo");
```

will read a file named `SNSLattice.dmo` (created by the code DIMAD [3]) and add a transfer matrix for each matrix produced by the sequence at the end of the DIMAD input. The procedure for creating the DIMAD file is to first construct a standard lattice file which generates the usual listing of Twiss parameters. Then each internodal transfer matrix must be explicitly printed by using the DIMAD input file sequence:

```
print
interval marker1 marker2 99 end,
-11 1,
```

where `marker1` and `marker2` are predefined markers in the lattice. This means that unique markers must be included everywhere in the lattice where a transfer matrix is desired. This process works well for situations when many ring elements are lumped together and there are only a limited number of transfer matrices desired (~ 10's of matrices). When tracking needs to be subdivided into many small steps, as needed with transverse space charge, 100's of transfer matrices are needed and the generation of the DIMAD input is tedious and error prone. The following described use of MAD [4] generated transfer matrices is recommended in this case.

4.1.1.3 Using MAD to generate the lattice transfer matrices:

The `readMADFile` routine expects to read two files generated by MAD. The first is a TWISS file and the second file contains the transfer matrices. The format of the command is

```
readMADFile("SNSTwiss.out", "SNSTM.out");
```

There is one piece of information that should be input independently from the above call, namely the transition gamma. E.g., also include a line like:

```
gammaTrans = 4.93;
```

The Twiss file can be generated by including a line in the MAD input file like:

```
TWISS, SAVE, TAPE=SNSTwiss.out
```

And the first order transfer matrix file "`SNSTM.out`" is generated by including a line like

```
SELECT, FLAG=FIRST, RANGE=#S/#E
```

in the MAD input file, and by redirecting the output to the file `SNSTM.out`. This is a convenient way to produce the transfer matrices between each element of a Ring lattice, without going through the cumbersome DIMAD set up procedure.

For both the MAD and DIMAD file reading routines, each transfer matrix member is added with an `_oindex` value incremented by 10 from the previous transfer matrix (the initial transfer matrix `_oindex` value is 10). Thus there is room between each transfer matrices to add more `Nodes` (like a space charge kick, diagnostic, foil, etc.).

4.1.1.4 The first order Transfer Matrix Node calculations

Presently the code simply advances the macro-particles of a specified herd through the linear transfer matrix. The same transport equations as used in ACCSIM [2] are employed here. Also, as discussed below, some tune shift calculations are optionally provided here.

4.1.2 Second Order Transfer Matrices:

This capability is experimental at present.

4.1.3 Tune Shift Calculations:

Note: this set of calculations will likely be moved to a diagnostic module in the future.

Tune shifts calculations are possible for a herd. This is useful when transverse space charge kicks are employed, or a second order matrix advance is used (otherwise the particles just have the bare lattice tune). There are two tune shift calculations provided. One method involves following the phase advance of each macro-particle as it is advanced through each transfer matrix. As long as the are enough transfer matrices included to allow tracking steps small compared to the betatron period, this technique will allow computation of the absolute tune of each particle.

This method is slow and should not be employed during long runs, but rather at the end of a run or during a "snapshot turn". The following input file fragment shows how the tunes could be calculated and dumped to a file during a run:

```
doTurn(1000);              // do 1000 turns of the mainHerd
useSimpleTuneCalc=False;   // default value anyway, but let's be clear
startTuneCalc();           // start calculating tunes
doTurn(1);                 // calculate the tunes over one turn.
stopTuneCalc();            // stop calculating the tunes, and tally the
                           // tunes for each particle.

OFstream fio("tunes.out", ios::out);   // make a file to dump tunes to

dumpTAndA(mainHerd, fio); // print the tunes and actions to fio

// plot tunes to screen, for those with the PLPLOT package built in:

scatterplot = True;
xyPlot(yTune, xTune, xLabel="nu-x", yLabel="nu-y", mark=dot);

doTurn(999);               // do more turns without calculating the tunes.
```

If the switch `useSimpleTuneCalc` is set to 1 (or `True`), the betatron phase is checked only at the beginning and end of a turn, and the phase advance is calculated from these two quantities. In this case, only the fractional tune (between 0 and 1) is calculated. Otherwise each particle tune is calculated as described above. This method is much faster than the above method. However, when calculating the tunes with `useSimpleTuneCalc` non-zero, several restrictions apply. First, additional macro-particles can not be added to the herd while the tunes are being calculated.

Secondly, only one turn should be done between the `startTuneCalc()` and `stopTuneCalc()` calls. This is to ensure a unique solution. If you would like to average the tune calculation over several turns, with this method, do something like:

```
doTurn(1000);              // do 1000 turns of the mainHerd
useSimpleTuneCalc=True;    // Only get the fractional tune
for(_i=1; _i <= 10; _i++) //  Average tunes over 10 turns.
                           // Note: _i is a built-in SuperCode Integer

{
   startTuneCalc();        // start calculating tunes
   doTurn(1);              // calculate the tunes over one turn.
   stopTuneCalc();         // stop calculating the tunes, and tally the
}                          // tunes for each particle.
```

WARNING: Do not try to calculate the tune with closed orbit bumps on, unless the closed orbit bumps to not have any transfer matrices between them. For example, the following sequence is OK:

StartTuneCalc()
UpBump
Foil
DownBump
Bunch of transfer matrices
StopTuneCalc()

The following sequence would give incorrect tunes:

StartTuneCalc()
Foil
DownBump
Bunch of transfer matrices
UpBump
StopTuneCalc()

4.2 Injection Module (Foil)

This module includes capabilities for creating particles from specified distributions. It also contains a Foil node class. One can generate a specified number of macro-particles at the start of a run and track them, or automatically inject a specified number of macro-particles at the foil at each turn. Also foil scattering can be used.

4.2.1 Specifying the injected particles distribution types.

Macro-particles can be sampled from prescribed distribution types. The phase space distributions which are sampled by the `InjectParts` routine are specified by calls to the routines:

```
Void addXInitializer(const String &n, Subroutine subX)
```

```
Void addYInitializer(const String &n, Subroutine subY)
Void addLongInitializer(const String &n, Subroutine subL)
```

These routines tell the `InjectParts` routine what routines to use to sample the horizontal, vertical and longitudinal distributions respectively. For example, "`subX`" will be called to provide the *x, x'* values for a new macro-particle. The values provided by the specified sampling routines are displacements (the *dx, dx',dy, dy', dE and df)* which are assigned to the variables `dXInj` (mm), `dXPInj` (mrad), `dYInj` (mm), `dYPInj` (mrad), `deltaE` (GeV), and `phi` (rad). These displacements are added to the central coordinates of the injected beam (x_0, x'_0, y_0, y'_0) which are given by the quantities `x0Inj, xP0Inj, y0Inj, and yP0Inj` respectively. The central coordinates are provided to facilitate injection offset from the closed orbit, if desired. You can write your own initialization routine in the input shell script, or simply refer to one of the built in routines discussed below.

4.2.2 Built-in distribution types

4.2.2.1 "Joho" – binomial forms:

Some built-in routines are available for the purpose of randomly sampling common distribution types. The `JohoXDist` , `JohoYDist` and `JohoLongDist` routines supply values sampled from "Joho" horizontal , vertical and longitudinal distributions respectively, where the Joho distribution is a general binomial form taken from Ref. 2. In fact, this entire distribution type follows directly from ACCSIM [2]. For all the "Joho" distributions types, tails can be added. An example setup for using `JohoXDist` is:

```
x0Inj = 30.;  xP0Inj = 0.;      // horizontal center of the injected
                                //   distributiom (mm, mrad)
alphaXInj = 0.5;betaXInj = 10.; // lattice parameters for the injected
                                //   distribution,  beta in m
epsXLimInj = 10.;               // Limiting emittance of injected
                                //    distribution (pi-mm-mrad)
MXJoho = 3;                     // binomial factor.
addXInitializer("Truncated Gaussian X", JohoXDist);
```

It is possible to add tails to the "Joho" distributions. For example the following additional specifications:

```
xTailFraction = 0.3; // 30% of the particles are in the "tail"
xTailFactor = 2.2;   // The "tail" dist is
                     // 2.2 times > than the normal dist.
```

specifies that 30% of the particles created will have an emittance 2.2 times that of the nominally specified value (with `epsXLimInj`). The vertical distributions can be set up similarly by substituting "y" for "x" above. There is an option to use a longitudinal Joho distribution also, but here the distribution extent is determined by specifying the limiting longitudinal phase angle (`phiLimInj` in degrees) and the limiting energy spread (`dELimInj` in GeV). For example:

```
MLJoho = 1;     //  uniform longitudinal.
PhiLimInj = 100.;  // injection bucket is from –100 < phi(degrees) < 100
```

```
   addXInitializer("Uniform Long", JohoLDist);
```

A `UniformLongDist()` routine supplies samples from a uniform longitudinal distribution in energy and phase. Also the `GULongDist` can be used for longitudinal distributions which are uniform in f and gaussian (optionally truncated) in energy. See the Injection.mod and Injection.cc files for a complete description of the built-in sampling routines.

Note that the user can provide her own, say, horizontal distribution sampling routine in an input script file, and simply point to it with the "`addXInitializer`" call, without recompiling. For example,

```
// randomly assign the horizontal distribution to the discrete points:
// x = -5., -4., … 5. mm, and x′ = 0 mrad

   Void newHorDist()
   {
     Integer seed=1;
     dXInj = -5.  + Integer(11.*ran1(seed)); // ran1 is a built-in random
                                             // number generator (between 0,1)
     dXPInj = 0.;
  }
addXInitializer("Discrete X", newHorDist);
```

4.2.3 Continuous injection at a foil

The Injection module contains a `Foil` class which includes information about the foil. Generally a Foil node will be at the beginning of the ring, but it doesn't have to be. A Foil node can be included with a call to the routine:

```
Void addFoil(const String &name, const Integer &order,
    const Real &xMin,  const Real &xMax, const Real &yMin,
    const Real &yMax, const Real &thick)
```

Here name is a name of the node, order is the `_oindex` Node value, `xMin(xMax)` is the minimum (maximum) horizontal foil extent in mm, `yMin(yMax)` is the minimum (maximum) vertical foil extent in mm, and `thick` is the foil thickness ($\mu g/cm^2$). For example, the call:

```
   addFoil ("Foil", 2, 20., 30., 20., 100., 300.);
```

inserts a Foil node at point 2 in the ring, with $20 < x(mm) < 30$, and $20 < y(mm) < 100$, and with a thickness of 300 ($\mu g/cm^2$). Each time the `Foil` node is encountered the routine `InjectParts(nMacrosPerTurn)` is called. This routine will pick `nMacrosPerTurn` macro-particles from the prescribed distributions (see above), and add them to the `mainHerd` ensemble until a maximum of `nMaxMacroParticles` macro-particles have been injected.

4.2.4 The Foil Node calculations

In the `_nodeCalcutor` routine, particles are injected as described above(if the `mainHerd` is being pushed through the ring). Also if foil scattering is active, some beam energy dependent quantities are calculated. In the `_updatePartAtNode` routine, foil traversal events are tallied for each macro-particle. Also if foil scattering is active, each macro-particle is scattered if it intersects the foil.

4.2.4.1 Foil Scattering

If the variable `useFoilScattering == 1 (or True)`, foil scattering is modeled, using the ACCSIM single scattering model described in Ref. 2. In this case the following quantities are used:

```
foilZ = 6;        // default charge number
foilAMU = 12.01;  // default mass AMU
rhoFoil = 2.265;  // default foil density (g/cm3)
muScatter = 1.35; // default effective atomic radius parameter
                  // (see ACCSIM manual).
```

which can be overridden by the user in the input script.

Foil information can be dumped to a stream using:

```
showFoil(fio);
```

where `fio` is a defined Ostream such as "`cout`" for the console. One can manually cause the foil parameters to be updated with the call:

```
miscFoilCalcs();
```

and subsequently dump specific foil information to a stream. A common foil quantity of interest is `avgFoilHits`, which is the total number of foil traversals divided by the total number of macro-particles. See the file Injection.mod for more information on foil parameters.

4.2.5 Creating a distribution using built-in initializers without a Foil

A set of macro-particles for the mainHerd can be created using the initializtion routines discussed above without using a foil. The `addXInitializer, addYInitializer,` and `addLongInitializer` routines are called as above to specify the distributions you want to sample from. Then, for example, calling

```
nMaxMacroParticles = 10000;
InjectParts(10000);
```

will add 10000 macro-particles to `mainHerd` , which can subsequently be tracked. Be sure you have first specified which distribution functions to use though (see above).

4.3 Bump Module

The closed orbit of the ring can be artificially altered anywhere in the Ring by introducing an ideal bump. The bump positions can also be a function of time. This is typically done directly before (and after) a `Foil` node to facilitate painting of the ring phase space distributions. The node sub-class used to contain the bumps is the `IdealBump`. WARNING: Bumps should be added in pairs: an "up" bump, and a "down" bump! An `IdealBump` node can be added with the routine:

```
Void addIdealBump(const String &name, const Integer &order,
    const Integer &upDown,  const Subroutine sub)
```

Here `order` is the `_oindex` value specifying where in the ring the bump is. `sub` is the name of a routine that will specify the bump values as a function of time, and `upDown` is a switch indicating whether the bump is moving the closed orbit up (==1) or down (!=1). Note that the same routine should be specified by "`sub`" when adding both the "up" and "down" bumps, to ensure no net movement of the closed orbit. The routine specified by `sub`, which is called to provide the bumps as a function of time is expected to set the variables:

`xIdealBump`	- "The x value of the ideal bump at a point in time (mm)",
`xPIdealBump`	- "The x prime of the ideal bump at a point in time (mrad)",
`yIdealBump`	- "The y value of the ideal bump at a point in time (mm)",
`yPIdealBump`	- "The y prime of the ideal bump at a point in time (mrad)"

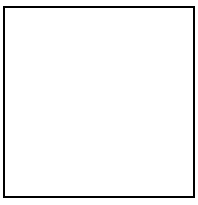
as a function of the variable `time` [msec]. These variables are then used to modify the closed orbit parameters, and accordingly the values of *x,x',y,*and *y'* of each macro-particle.

4.3.1 Using built-in bump forms:

Some bump as a function of time are built in. These are described here.

4.3.1.1 Exponential bump form 1:

An exponentially decaying bump profile is provided in the routine `eFoldBump`. It assumes profiles of the form



where $x_{0(f)}$ is the initial (final) *x* bump position, D_t is the total bump period, and t_x is a normalized time constant. The *x'*, *y* and *y'* bumps are specified in a similar manner in the `eFoldBump()` routine. The variable names for these quantities are described in the Bump.mod file. For example,

```
addIdealBump("upBump",1,1,eFoldBump); // add a closed orbit "up" bump
                    // at node point 1, using the eFoldBump routine.
```

```
addIdealBump("downBump",5,-1,eFoldBump); // add corresponding "down" bump

xBump0 = 20.; xBumpF = 0.;      // bump from 20 to 0 mm horizontal
xPBump0 = 0.; XPBumpF = 0.;     // bump from 0 to 0 mrad
tBump0 = 0.; tBumpF = 0.974;    // bump is on from 0 to 0.974 msec
eFoldTimeX = 4.;          // 4 efolds will occur between tBump0 and tBumpF
```

Similar notation is used for the vertical bump setup with `x(X)` replaced by `y(Y)`. If times > `tBumpF` are used, the final bump values are used (i.e, `xBumpF, xPBumpF, yBumpF, yPBumpF`).

4.3.1.2 Exponential bump form 2:

Exponential decaying bump profiles can be input in the form:

$$x(t) = x_0 + x_1 e^{\frac{-t}{t}}$$

Using the `eFoldBump2` routine. For example:

```
addIdealBump("upBump",1,1,eFoldBump2); // add a closed orbit "up" bump
                  // at node point 1, using the eFoldBump2 routine.
addIdealBump("downBump",5,-1,eFoldBump2); // add corresponding "down" bump

xBump0 = 20.; xBump1 = 10.;     // bump starts at 30 mm horizontal
xPBump0 = 5.; XPBump1 = 2.5;    // bump starts at 7.5 mrad
eFoldTimeX = 4.;                // e-fold time  = 4 msec.
```

Similar notation is used for the vertical bump setup with `x(X)` replaced by `y(Y)`.

4.3.1.3 Interpolate Bump coordinates.

The closed orbit bump coordinates can be prescribed by linear interpolating between input data points using the `interpolateBumps` routine. The procedure is to first size and fill a set of vectors containing the bump points to use for interpolation. Then insert a pair of up/down bumps which use the `interpolateBumps` routine to scale the ideal bump coordinates. The following example illustrates this procedure for a case with two points for interpolation (i.e. a linear bump):

```
sizeBumpPoints(2);    // use 2 bump points

// The time (abscissa) points for the interpolation are
// 0 and 0.5 msec:
bumpTimes(1) = 0.; bumpTimes(2) = 0.50;

yBumpPoints(1) = 16.;   // y is bumped from 16 to 0 mm
yBumpPoints(2) = 0.;
yPBumpPoints(1) = 2.2044;  // y' is bumped from 2.2044 to 0 mrad
yPBumpPoints(2) = 0.;

xBumpPoints = 0.;   //  the x (horizontal) direction
xPBumpPoints = 0.;  //  is not bumped at all here!

// add a closed orbit up bump at the end of the ring:
```

```
addIdealBump("upBump",9999,1, interpolateBumps);
                                                //
// add the closed orbit down bump near the beginning of the ring:
addIdealBump("upBump",3, -1, interpolateBumps);
```

More interpolation points can be included, but be sure to adequately size the bump point storage.

4.3.2 User specified bump form

The user can add a routine to specify any sort of bump profile in the input script file, and simply refer to this routine in the `addIdealBump` call. For example the following set of commands would create and activate the use of trigonometric bump forms :

```
Real bumpFrequency = 1;
Void trigBumps()
{
  // x goes from 10 to 0 mm (0 < t(msec) < pi/4 msec) :
  xIdealBump = 10. * cos(time*2*pi* bumpFrequency);
  // x' goes from -10 to 0 mrad (0 < t(msec) < pi/4 msec) :
  xPIdealBump = -10. * cos(time*2*pi* bumpFrequency);
  // y goes from 0 to 10 mm (0 < t(msec) < pi/4 msec) :
  yIdealBump = 10. * sin(time*2*pi* bumpFrequency);
  // y' goes from 0 to 10 mrad (0 < t(msec) < pi/4 msec) :
  yIdealBump = 10. * sin(time*2*pi* bumpFrequency);
}

// add the new trig closed orbit up bump at the end of the ring:
  addIdealBump("upBump",9999,1, trigBumps);
                                                //
// add the new trig closed orbit down bump near the beginning of the ring:
  addIdealBump("upBump",3, -1, trigBumps);
```

4.4 Acceleration Module

4.4.1 Non-accelerating RF

This node is provided to add RF cavities. The Node sub-class for this purpose is the RFCav class, which inherits the base class `AccelerateBase`. For this class of RF Cavity the synchronous phase is always assumed to be 0. The general capability is to provide the voltage as the sum of an arbitrary number of harmonics, and as a function of time. The user must specify the number of harmonics to be used, vectors containing the amplitudes and phases of each voltage harmonic, and (if the voltage is ramped with time) a routine providing their waveforms. The RF Voltage for a cavity is assumed to be of the form:

$$V_{RF} = \sum_{i=1,n} V_i \sin(h_i f - f_i),$$

where n is the number of RF harmonics used, V_i is the i'th harmonic voltage (kV), h_i is the i'th harmonic number, f_i is the i'th harmonic phase (rad) and t is the time (msec). A steady state RF cavity with an arbitrary number of harmonics can be added with the routine:

```
Void addRFCavity(const String &n, const Integer &o,  Integer &nh,
          RealVector &v,  RealVector &hn,  RealVector &p)
```

The first argument is a name for the node, the second argument is the order number where this RF cavity appears in the Ring, the third argument is the number of harmonic components the RF Voltage waveform has, and the fourth, fifth and sixth arguments are references to Real Vectors containing the RF voltage (kV), harmonic number, and phase offset(rad) of each harmonic component of the RF voltage, respectively. The RealVectors referenced in this call can be created, and initialized, on the fly prior to this call. An example of this is:

```
Integer nHarms = 2;
RealVector RFVolts(nHarms), RFHarmNumber(nHarms), RFPhase(nHrams);
RFPhase = 0.;  // All components have phase = 0;
RFHarmNum(1) = 1; RFHarmNum(2) = 2;  // use 1st and 2nd harmonics
RFVolts(1) = 40.; RFVolts(2) = -20.;  // 1st component = 40 kV, 2nd = -20 kV
addRFCavity("RF Cavity 1", 200, nHarms, RFVolts, RFHarmNumber,. RFPhase);
```

It is also possible to specify a pulsed RF waveform. The following example specifies a ramped RF waveform going from 6 to 10.5 kV between 0 and 0.5 msec, and equal to 10.5 kV thereafter. Only a single harmonic component is used:

```
Integer nRFHarms = 1;
RealVector Volts(nRFHarms), harmNumber(nRFHarms), RFPhase(nRFHrams);
RFPhase = 0.;  // phase = 0;
harmNum(1) = 1; // use only the 1st harmonic
Real tFactor;
Void PSRVolts()
{
 tFactor = (time > 0.5) ? 0.5, time;  // check for flattop
 Volts(1) = 6. + 4.5*tFactor;
}
addRFCavity("RF Cavity 1", 200, nHarms, RFVolts, RFHarmNumber,. RFPhase);
```

The RF waveform can be more complicated than the simple linear ramp example shown above.

4.4.2 Accelerating RF

For an accelerating bunch simulation, three quantities are of particular interest: 1) the time profile of the dipole field, 2) the time profile of the accelerating RF voltage, and 3) the time variation of the synchronous phase. Two of these quantities must be specified, and the third solved for. Presently, support is provided for the case when the field and RF voltages are specified.

To use acceleration the following information about the ring must be specified:

```
////////////////////////////
// Make  a Ring
////////////////////////////

  lRing = 299.2;              // ring circumference (m)
  gammaTrans = 5.91;          // transition gamma
  rhoBend = 2. * 32./(twoPi); // dipole bend radius (m)
```

The parameter is provided automatically if you read a lattice file (e.g. if you are doing transverse space tracking). The above snippet was taken from a file that did longitudinal only tracking and did not use a lattice file.

4.4.2.1 Ramped B Acceleration

This option requires the user to provide the time profile of the dipole field and RF voltages. The synchronous phase is solved for. If incompatible field/voltage values are specified, the computation stops with an appropriate message. A ramped B field accelerating node can be added to the calculation with the routine

```
Void addRampedBAccel(const String &name, const Integer &o,
            const Subroutine sub, const Integer &nh,
            RealVector &rfv, RealVector &rfp)
```

As usual for nodes, the first argument is a name for the node and the second is the order value where the node should be placed (around the ring). The third argument is the name of a routine to be called that will provide the current dipole field and RF voltages (see below). The fourth argument is the number of harmonics the RF voltage will use, the fith argument is a reference to the vector containing the RF voltages for each harmonic (which will likely change with time). The last argument is the vector holding the vector of RF phase shifts for each RF voltage (usually this is kept at 0, but these can also be varied if you want to get fancy).

When the calculation gets to the accelerating node, the "_nodeCalculator" first calls the specified routine to update the field and RF voltage. The user can create some vectors to hold the RF voltages and phases and write his own routine to scale these parameters in the driver shell. The field variable that should be provided here is called "`Accelerate::BSynch`" and the independent time variable is "`Ring::time`" .

Specifying a cos form field ramp and use interpolated RF voltage values

However, there are several built-in capabilities for modeling commonly used accelerating cycles. One is to assume sine waveforms for the dipole and interpolate RF voltages from an input set of data. This is done by using the `InterpolateV` routine. Here the dipole filed is assumed to be of the form:

$$B(t) = B_0 - B_1 \cos(2\pi tf).$$

The user must specify the "fixed" field level (B_0, in T with variable `BSynch0`), the "varying" field level (B_1, in T with vaiable `BSynch1`) and the ramp frequency (Hz) with the variable `BSynchFreq`. The RF voltage is assumed to be of the form

$$V_{RF} = \sum_{i=1,n} V_i \sin(if - f_i).$$

The voltage at each node is calculated by interpolating between data read in before the run starts. The A routine getRampedV(String) is provided to read in this data. The argument here is the name of the file to read, and the file format is expected to be of the form:

- the first line contains an integer specifying the number of RF harmonics (*n*)
- each subsequent line contains the following white-space delimited parameters: the time *t* (msec), $V_1(t)$, $f_1(t)$, … $V_n(t)$, $f_n(t)$

The `InterpolateV` routine puts the interpolated values of the RF voltages and phases into predefined vectors `RFVolts` and `RFPhase`, and the number of RF harmonics in `nRFHarmonics`, so be sure to specify these in the call to `addRampedBAccel`.

Here's an example of the implementation of the "ramped-V" acceleration node:

```
////////////////////////////
// Add an Acceleration Node
////////////////////////////

   Ring::harmonicNumber  = 2; // two bunches in ring for this case.

   cerr << "set up RF\n";

  ignoreNegAcel = 1;  // lets inject in falling field and ignore accel
there

  // Set up the cos form dipole field ramp:

  Real Bmin = 0.3124980; Real Bmax = 0.9118358; // in T
  BSync0 = (Bmax + Bmin)/2.;
  BSync1 = (Bmax - Bmin)/2.;
  BSyncFreq = 30.; // 30 Hz ramp

  getRampedV("SNSCosForm1");  // get data to interpolate RF voltage from

  time = -0.7;  // start injection at –0.7 msec

  addRampedBAccel("Accelerate",15 , InterpolateV, nRFHarmonics,
      RFVolts, RFPhase);  // This node goes at location 15.
```

The input file SNSCosForm1 must be lying around, and here what it looks like:

```
1
-0.7            30.     0.
0.              65.     0.
0.0948459       80      0.
0.221275        100     0.
0.389806        120     0.
0.614457        140     0.
.
.
.
14.1111         260     0.
14.963          250     0.
15.8148         240     0.
16.6667         230     0.
```

Using interpolated B and RF voltage values

This case is similar to that above case except that the B field is also calculated by interpolating between input data instead of assuming a *cos* waveform. In this case the `InterpolateBV` routine is specified in the call to `addRampedBAccel`. Also routine (`getRampedBV(String)`) is provided for use with this option that will read in data to interpolate both *B* and *V*. It expects an argument with the filename of a file containing the following format:

- the first line contains an integer specifying the number of RF harmonics (*n*)
- each subsequent line contains the following white-space delimited parameters: the time *t* (msec), *B*(T), $V_1(t)$, $f_1(t)$, … $V_n(t)$, $f_n(t)$
-

The `InterpolateV` routine also puts the interpolated values of the RF voltages and phases into predefined vectors `RFVolts` and `RFPhase`, and the number of RF harmonics in `nRFHarmonics`, so be sure to specify these in the call to `addRampedBAccel`.

Here's an example of the implementation of the "ramped-BV" acceleration node:

```
////////////////////////////
// Add an Acceleration Node
////////////////////////////

   Ring::harmonicNumber  = 2; // Two bunches in ring

   cerr << "set up Acceleration\n";

   getRampedBV("SNSWaveForm5");

   time = 0.;  // start injection  at t=0 msec

   addRampedBAccel("Accelerate",15 , InterpolateBV,nRFHarmonics,
      RFVolts, RFPhase); );  // This node goes at location 15.
```

The input file SNSWaveForm5 must be lying around, and here what it looks like:

```
2
0.              0.3124980       35.     0.     -12.    0.
0.7             0.312498        65      0.     -22.    0.
0.754761        0.312516        75      0.     -25.    0.
0.810562        0.312571        83      0.     -28.    0.
0.910526        0.312764        90     0.      -30.    0.
0.985233        0.312987        100     0.     -33.    0.
1.12105         0.313561        106     0.     -35.    0.
.
.
.
21.9333         0.911684        270     0.     -90.    0.
22              0.911836        265     0.     -88.    0.
```

This case used a dual harmonic RF.

4.4.2.2 Acceleration Output:

To get output about acceleration, try using `showAccelerate(Ostream)`. This routine dumps the time (msec), synchronous particle kinetic energy (GeV), the bunch factor, the relativistic beta, the synchronous phase (deg), the primary harmonic RF voltage, the syncrotron frequency, the bucketHeight (eV), the bunch height (eV), the bucket area (eV-sec), the bunch area (eV-sec) and the maximum *dp/p* (%) (all on a single line). Here's an example of its use:

```
// Files for output:

  runName = "SNSRCS1";
  String name1 = runName + ".out";
  OFstream fio(name1, ios::out);

for (_i=1; _i <=28 ; _i++)  // Do 3500 turns
 {
   doTurn(125);
   showAccelerate(fio);  // send acceleration info to a file
   showAccelerate(cout); // send acceleration info to the screen
 }
```

4.5 Longitudinal Space Charge

A longitudinal space charge node (1) bins the longitudinal beam profile, (2) calculates the longitudinal space charge force, and (3) applies a momentum kick, based on this space charge force to the macro-particles. The longitudinal profile calculated here is also used as a weighting factor for the transverse space charge kicks. Typically, only a few (or likely just one) of these nodes will be used, as the longitudinal profile does not change much during a single turn. There is a base longitudinal space charge node class, which contains the basic members any implementation will need. Presently, there is only one derived longitudinal space charge implementation, namely an FFT one.

4.5.1 FFT Longitudinal Space Charge

An FFT longitudinal space charge kick node can be added anywhere in the ring with the `addFFTLSpaceCharge` routine. This routine takes 6 arguments:

1 A name for the node (String)
2 The index order used to place the node (Integer)
3 A vector containing the *Z/n* wall longitudinal coupling impedance (Complex vector)
4 The wall/beam radius ratio (Real)
5 A flag indicating whether kick values interpolated from the phase bins should be used (if ==1 an interpolation scheme is used, otherwise a kick is calculated for each macro-particle's explicit phase). (Integer)
6 A minimum number of macro-particles required before the calculation is done. Typically setting this number to ~ 100 prevents spurious numerical events. (Integer).

Also the number of longitudinal bins to use is specified independently as the integer `nLongBins`. The wall impedance input vector for argument 3 has values for each harmonic number, and should

be dimensioned to at least the number of longitudinal bins/2 and is in units of Ohms. The real part of element *i* contains the resistive impedance in Ohms /(harmonic number *i*) , and the imaginary part of the vector contains the reactive part of the impedance (Ohms) /(harmonic number i). Setting this vector = 0, is equivalent to ignoring wall impedance effects.

As an example of including an FFT longitudinal space charge node consider:

```
//////////////////////////////////////////
// Add an FFT Longitudinal Space Charge Node
// without wall impedances
//////////////////////////////////////////

  ComplexVector ZImped(16);
  ZImped = Complex(0.,0.);    // Since it is ==0, there will be no
                              // wall impedance effects included.
  nLongBins = 32;             // Use 32 longitudinal bins (It's good to use
                              // a power of 2 for the FFT implementation)
  Real b_a = 2.;              // Use this for wall/beam radius
  Integer useAvg = 0;         // Let's not use the interpolation method
  Integer nMacroLSCMin = 100; // Require at least 100 macros present before
                              // trying to use this.

// Now let's add it between the nodes with indexes of 10 and 20:

  addFFTLSpaceCharge("LSC1", 17, ZImped, b_a, useAvg,nMacroLSCMin);
```

Here's an example of including an FFT longitudinal space charge node with wall impedance:

```
//////////////////////////////////////////
// Add an FFT Longitudinal Space Charge Node
// with wall impedances
//////////////////////////////////////////

  ComplexVector ZImped(16);
  ZImped(1) = Complex(115,53); // real part of 1st harmonic is 115 Ohm,
                               // imaginary part is 53 Ohms,
  ZImped(2) = Complex(110,67); // real part of 2nd harmonic is 220 Ohm,
                               // imaginary part is 134 Ohms,
  ZImped(3) = Complex(130,53); // and so on.
  ZImped(4) = Complex(160,27);
  ZImped(5) = Complex(48,12);
  ZImped(6) = Complex(45,7);
  ZImped(7) = Complex(43,6);
  ZImped(8) = Complex(43,5);
  ZImped(9) = Complex(44,4);
  ZImped(10) = Complex(45,3);
  ZImped(11) = Complex(45,3);
  ZImped(12) = Complex(45,3);
  ZImped(13) = Complex(45,3);
  ZImped(14) = Complex(45,3);
  ZImped(15) = Complex(45,3);
  ZImped(16) = Complex(45,3);
  nLongBins = 32;             // Use 32 longitudinal bins (It's good to use
                              // a power of 2 for the FFT implementation)
  Real b_a = 2.;              // Use this for wall/beam radius
  Integer useAvg = 0;         // Let's not use the interpolation method
  Integer nMacroLSCMin = 100; // Require at least 100 macros present before
```

```
                                // trying to use this.
// Now let's add it between nodes with indexes 10 and 20:
  addFFTLSpaceCharge("LSC1", 17, ZImped, b_a, useAvg,nMacroLSCMin);
```

4.6 Transverse Space Charge

The transverse space charge nodes provide a transverse kick due to the space charge force. There is a base class called TransSC. There are presently three derived kinds of transverse space charge nodes: (1) pair-wise sum, (2) brute-force Particle-In-Cell (PIC), and (3) and FFT PIC nodes. To use any of these meaningfully it is necessary to add many of these nodes around a ring, including at least one node per scale length of the beam shape variation and > 10 per betatron oscillation. The easiest way to add these nodes is with the routines described below, which automatically insert a node after each transfer matrix included in the lattice file. This requires generation of a lattice file with sufficient fineness. As described above in Section 3.2.1, this is easier using MAD than with DIMAD.

4.6.1 Pair-wise sum

The pair-wise sum transverse space charge method simply calculates the Coulomb force on one particle by summing the force over all other particles. A smoothing parameter (ε_s) is included in the calculation as an additional length used in calculating the particle separation distances. Typically one uses a length very small compared to the beam transverse size, in which case it only reduces the Coulomb force between near-neighbors. This smoothing parameter can prevent spurious kicks. No binning is done with this scheme. This scheme is simple, but requires ~ $(n_p)^2$ operations to calculate the kicks, where n_p is the number of macro-particles. This is exceedingly slow, and has only been used for checking other faster methods. It is not recommended for normal usage, but none-the-less may be implemented by:

```
    Real epsSmooth = 0.5;             // smoothing parameter in mm
    Integer nMacroMin = 100;          // minimum number of Macros to
                                      //   accumulate before doing SC calc.
    addPWSTransSCSet(epsSmooth, nMacroMin); // add the space charge node set.
```

4.6.2 Brute-Force PIC

A straight-forward PIC implementation is provided with the brute-force PIC Class. At each of these nodes, the macro-particles are binned on a prescribed X-Y grid, the force at each grid point is calculated using the binned particle distribution, and the force on each particle is calculated by a bi-linear interpolation from the grid. The grid extent is determined by first calculating the macro-particle extreme locations, and by making a rectilinear grid extending a prescribed amount beyond the particle extent. A smoothing parameter is also used in calculating the force between grid points, similar to that described above (see Ref 5.). This smoothing parameter can be used if the bins are sparsely populated, and should have magnitude comparable to the bin spacing. However,

it is recommended to use enough macro-particles to populate the bins to at least 10 particles per bin, and let the smoothing parameter $\rightarrow 0$.

The dominant calculation time for this method is typically the force calculations on the grid, which is ~ $(n_{bin})^4$ operations. While this is faster than the pair-wise sum calculation described above, for bin sizes >10, the FFT method described below is faster. A set of brute-force PIC transverse space charge nodes may be added as:

```
   Real epsSmooth = 0.5;   // smoothing parameter in mm
   Integer nMacroMin = 100; // minimum number of Macros to
                        //   accumulate before doing SC calc.
   Real gridFac = 1.1;  // Make grid extend 10% beyond macro-particles
   Integer nxBins = 10;       // Number of horizontal bins
   Integer nyBins = 10;       // Number of vertical bins

// add the brute-force PIC space charge node set.

   addBFTransSCSet(nxBins, nyBins, epsSmooth, nMacroMin, gridFac);
```

4.6.3 FFT-PIC

The fastest method of calculating the transverse space charge effects is with a set of FFT PIC nodes. These nodes are similar to the brute-force PIC in that the calculation proceeds by first binning macro-particles to a grid, then calculating the force distribution on the grid, and finally by interpolating the force from the grid back to each macro-particle. The difference is that an FFT method is used to calculate the force on the grid using the binned particle distribution. The grid is determined automatically, by first calculating the macro-particle extrema location, and by making a rectilinear grid centered at the average macro-particle (x,y) location, but extending twice as far as any particle. This buffer of empty bins is provided to prevent any false force contributions from the FFT method. This grid extent is not adjustable. A smoothing parameter similar to that used in the brute-force PIC method is also provided. Namely, A smoothing parameter is also used in the force calculation between grid points, as described in Ref 5. If a positive value is entered for the smoothing parameter, it is taken to be a length relative to a grid size. If a negative value is entered the smoothing parameter is taken to the absolute value of this number (in mm). This smoothing parameter can be used if the bins are sparsely populated, and should be comparable to the bin spacing. However, it is recommended to use enough macro-particles to populate the bins to at least 10 particles per bin, and let the smoothing parameter $\rightarrow 0$.

As with the brute-force PIC method, the dominant computation is in calculating the space charge forces across the grid. Since this method uses an FFT implementation the computation requires only ~ 2 n_{bin} $(n_{bin})^2$ operations.

NOTE: since an FFT method is employed here, the most efficient usage is to enter a number of bins that is a power of 2. The method will work for an arbitrary number of bins, but will likely proceed less efficiently.

A set of FFT PIC transverse space charge nodes may be added as:

```
   Real epsSmooth = 0.;       // don't use smoothing
   Integer nMacroMin = 100;   // minimum number of Macros to
                          //   accumulate before doing SC calc.
   Integer nxBins = 64;    // Number of horizontal bins
   Integer nyBins = 64;          // Number of vertical bins

// add the FFT PIC space charge node set.

   addFFTTransSCSet(nxBins, nyBins, epsSmooth, nMacroMin);
```

4.7 Thin Lens

Experimental.

4.8 Aperture

An aperture node can be placed anywhere in the ring to either (a) count particle hits beyond a certain position (transparent), or (b) remove macro-particles from the tracking if they extend beyond a certain position (non-transparent). The aperture position in the ring is determined by its node index value. Generally the mechanism for determining what index value to use is to first read in the transfer matrix set. An aperture can be placed after or before any of the transfer matrix nodes. If you are not sure which transfer matrix node corresponds to the longitudinal ring position you want to put the aperture at, read in the transfer matrix and do a "`showRing(cout)`" and a "`showTransMatrix(cout)`" to help figure out the node index where you would like to place the aperture. As each transfer matrix index is incremented by 10 from the previous transfer matrix index, just pick an index value between the two transfer matrices you are interested in. You can always do another "`showRing(cout)`" after adding the aperture node to see if you have it right.

4.8.1 Rectangular Aperture

Presently this is the only kind of aperture. The maximum and minimum positions are specified. An example of adding a rectangular aperture is:

```
////////////////////////////////////////
// Add an aperture
//  Put it after the 1st transfer matrix (node=11)
//  The betax here is 9.2 m and the betay is 9.4 m
//  I know the sigma of the injected beam emittance is 10 mm-mrad.
//   Let's figure the aperture postitions from these values.
//   The last argument is to make it non-transparent, i.e. lose particles.
//
////////////////////////////////////////

  Real xMin, xMax, yMin, yMax;

  xMin = -sqrt(9.2 * 10.* 5.); // minimum horizontal acceptance (mm)
  xMax = sqrt(9.2 * 10.* 5.);  // maximum horizontal acceptance (mm)
  yMin = -sqrt(9.4 * 10.* 5.); // minimum vertical acceptance (mm)
  yMax = sqrt(9.4 * 10.* 5.);  // maximum vertical acceptance (mm)
  Integer tranparent = 0;

  addRectAperture("Aperture1", 11, xMin,xMax, yMin, yMax, transparent);
```

For those who prefer to do setup calculations offline, the above could also be done with:

```
addRectAperture("Aperture1", 11, -21.448, 21.448, -21.679, 21.679, 0);
```

Note the transparent switch. If it is set to 0 (or False), the aperture will stop anything that hits it, remove it from its herd and store its information in the LostOnes object (see the section on the MacroPart class). If the transparent switch is set to 1 (or True), only particle hits are calculated at the aperture.

4.8.2 Getting Aperture Output

At some point you may want to see what the aperture has collected. The `showApertures` routine will dump some information about all apertures in the ring:

```
showApertures(cout);   //   show some aperture info to the screen
```

This output includes the aperture dimensions and how many particles have hit it. It is also possible to get a dump of all the lost particles for a herd with the `routine Particles:: dumpLostParts` . For example:

```
OFstream fio("aperture.dmp", ios::out);
dumpLostParts(mainHerd, fio);  // dump details of lost particles to
                               // the file "aperture.dmp" .
```

The first line of this file contains a description of the information dumped for each lost particle. Then a separate line is printed for each lost macro-particle containing the phase parameters x (mm), x' (mrad), y (mm) , y' (mrad), f (rad), DE (GeV), the longitudinal ring position where the macro-particle was lost (m), the node number where the macro-particle was lost, and the turn number when the macro-particle was lost.

5 Miscellaneous Modules

5.1 Output Module

Many modules contain some built in output capabilities of their own, which are described in the sections above. Additionally, some general information can be output from the Shell as desired by the user. Output capabilities to the screen, plots and files are described in Reference 1. These output capabilities can generally be done from the Shell, without requiring the need to add source code.

However, several canned text output routines are provided for commonly used output capabilities.

```
Void showNodes(Ostream &os)
                - "Routine to show all Node info to stream os";
Void showTransMatrix(Ostream &os)
                - "Routine to show TransferMatrix info to stream os";
Void showRing(Ostream &os)
                - "Routine to show all the ring information to stream os";
Void showTurnInfo(Ostream &os)
                - "Routine to show misc. general turn information";
Void showInject(Ostream &os)
             - "Sends Injection information to an Ostream.";
Void showFoil(Ostream &os)
             - "Sends Foil information to an Ostream.";
Void showTiming(Ostream &os, const Real &et)
             - "Writes CPU timing info to an Ostream.";
Void showStart(Ostream &os)
             - "Writes basic run input info to an Ostream.";
```

Note: the argument to these routines can either be:

(1) `cout` - long buffered output to the screen,
(2) `cerr` - short buffered output to the screen, or
(3) any user defined stream.

A user defined stream called "`fio`" to a file called "`fileout`" can be created on the fly by

```
OFstream fio("fileout", ios::out);
```

This stream will delete any previously defined file called "`fileout`". The command

```
OFstream fio("fileout", ios::app);
```

is similar, but will append the prescribed output to whatever exists (if anything) in a file called "`fileout`". See Reference 1 for more information on general inputting/outputting information from the Shell.

The string `runName` can be defined at the beginning of a run. It is used in the showStart output and on the plots (see below) for labeling output.

5.2 Plots Module

The Plots module contains routines to produce phase space plots. It only is available if the code was built with the PLPLOT environment variable defined to the installation directory of the PLPLOT package (see section 1.1). Plots can be generated interactively on the screen or dumped to a postscript file.

5.2.1 Built-in Plots

A number of commonly used accelerator physics plots are built in and described below.

5.2.1.1 Plotting to an X window

The horizontal, vertical, and longitudinal phase spaces of a herd can be plotted to an X-window display with the commands:

```
plotHorizontal(mainHerd);  //  plot the horizontal phase distribution
                           //  of the main Herd

plotVertical(mainHerd);    //  plot the horizontal phase distribution
                           //  of the main Herd

plotLongitudinal(mainHerd);//  plot the longitudinal phase distribution
                           //  of the main Herd
```

The real space distribution can be plotted to an X-window display with the command

```
plotXY(mainHerd);          //  plot the real space distribution
                           // of the main Herd
```

The longitudinal phase distribution along with the longitudinal space charge (if it has been calculated) can be plotted to an X-window display with the command

```
plotLongSet(mainHerd);     // plot the longitudinal space distribution
                           // of the main Herd and the long. space charge
```

The 3 phase space plots and the real space plot can all be plotted together (4 plots per page) on an X-Window display with the command:

```
plotXWin(mainHerd);        // plot the all phase space distributions +
                           // real space dist. of the main Herd
```

5.2.1.2 Plotting to a postscript file

The above built-in plots can also be sent to a postscript file. The commands to plot to a file called `test.ps` are:

```
plotLongSetPS(mainHerd, “test.ps”);   // plot the longitudinal space
plotXYPS(mainHerd, “test.ps”);        //  plot the real space distribution
plotHorizontalPS(mainHerd, “test.ps”);//  plot the horizontal phase
plotVerticalPS(mainHerd, “test.ps”);  //  plot the vertical phase
```

The 3 phase space plots and the real space plot can all be plotted together (4 plots per page) on a postscript file with the command:

```
plotPS(mainHerd, “testPlot.ps”); // plot the all phase space distributions
                                 // + real space dist. of the main Herd to
                                 // to a postscript file called testPlot.ps
```

5.2.1.3 Plot Settings

Any of the following quantities can be set while running, either in an input script file, or from the shell prompt.

Histogram Bins

Each of the phase space plots also includes a histogram inset. The number of bins used in the horizontal and vertical distributions is given by `nTransBins` (default = 32). The number of bins used in the energy spread distribution distributions is given by `nDeltaEBins` (default = 32). The number of bins used in the longitudinal phase is given by the number of bins used in the longitudinal space charge calculation: `nLongBins` (default = 32).

Plot Ranges

The plot extents are given by:

```
xMinPlot   - Minimum x for plots [mm]
xMaxPlot   - Maximum x for plots [mm]
xPMinPlot  - Minimum x-prime for plots [mrad]
xPMaxPlot  - Maximum x-prime for plots [mrad]
yMinPlot   - Minimum y for plots [mm]
yMaxPlot   - Maximum y for plots [mm]
yPMinPlot  - Minimum y-prime for plots [mrad]
yPMaxPlot  - Maximum y-prime for plots [mrad]
dEMinPlot  - Minimum value for dE plots [GeV]
dEMaxPlot  - Maximum value for dE plots [GeV]
phiMinPlot - Minimum value for the phase [rad]
phiMaxPlot - Maximum value for the phase [rad]
LSCMinPlot - Minimum value for Long. space charge plots [kV]
LSCMaxPlot - Maximum value for Long. space charge plots [kV]
```

Macro-particle plotting density

The density of macro-particles plotted on any plot is controlled by the variable `plotFreq`. One out of every `plotFreq` macro-particles is plotted. It is advisable to set this Integer variable to be > 1 for herd sizes > 10,000.

Plot labeling:

By default, the plots are labeled with the run name (define the string "`runName`") the turn number. This labeling can be suppressed, by setting `verbosePlot = 0`;

5.2.3 General plotting

The driver shell includes additional general plotting capability. See the section on plotting in Reference 1. This is useful for quick looks at vectors, etc. The code is distributed with the X-window plotting driver active, but if you can get the Tcl-dp library running on your system, use of this driver offers much superior interactive plotting flexibility.

6. References.

1. Haney, S.W., “Using and Programming the SUPERCODE”, UCRL-ID-118982, Oct. 24, 1994.
2. Jones, F. W. “Users Guide to ACCSIM “, TRI-DN-90-17, June 1990.
3. R.V. Servranckx, R.L. Brown, L. Schachinger, D. Douglas, “User’s Guide to the Program DIMAD”, SLAC report 285 UC-28 (A), May 1985.
4. H. Grote, C. Iselin, “The MAD Program”, CERN/SL/90-13 (AP), Rev.5 April 29, 1996.
5. J. A. Holmes, J. D. Galambos, D-O Jeon, D. K. Olsen, M. Blaskiewicz, A. U. Luccio, and J. J. Beebe-Wang, “Dynamic Space Charge Calculations for High Intensity Beams in Rings”, presented at the International Computational Accelerator Physics Conference, Monterey, CA, September 1998.
6. PVM (Parallel Virtual Machine) http://www.epm.ornl.gov/pvm/ .

Apendix 1. Description of the ORBIT Base Classes.

Table A.1. Description of the Synchronous particle class.

```
////////////////////////////////////////////////////////////////////////
//
// CLASS NAME
//    SyncPart
//
// INHERITANCE RELATIONSHIPS
//    SyncPart -> Object -> IOSystem
//
// USING/CONTAINING RELATIONSHIPS
//    Object (U)
//
// DESCRIPTION
//    A class for storing SyncPart
//
// PUBLIC MEMBERS
//    SyncPart:      Constructor for making SyncPart objects
//    ~SyncPart:     Destructor for the SyncPart class.
//    _mass          Mass (AMU)
//    _charge        charge number
//    _eKinetic      kinetic energy (GeV)
//    _e0            rest mass (GeV)
//    _eTotal        total energy (GeV)
//    _betaSync      v/v_light
//    _gammaSync     E/E_0
//    _dppFac        conversion factor from dE to dp/p
//    _phiCoef       conversion from length to angle phase
//
// PROTECTED MEMBERS
//  None
// PRIVATE MEMBERS
//    None.
//
////////////////////////////////////////////////////////////////////////
```

Table A.2 Description of the macro-particle class.

```
////////////////////////////////////////////////////////////////////////
//
// CLASS NAME
//    MacroPart
//
// INHERITANCE RELATIONSHIPS
//    MacroPart -> Object -> IOSystem
//
// USING/CONTAINING RELATIONSHIPS
//    Object (U)
//
// DESCRIPTION
//    A class for storing macro  particle info. The macro particles
//    abstraction is for a "herd" of macroparticles. Note, we puposely
//    to not store all atributes, statistics etc. about the macropartcles
//    in this class to keep the size as small as possible. Only the basic
//    proprties are stored here. The hope is that keeping the size small
//    will increase the likelyhood of the more-often used stuff getting
//    into cache.
//
// PUBLIC MEMBERS
//    MacroPart:        Constructor for making MacroPart objects
//    ~MacroPart:       Destructor for the MacroPart class.
// Real Vector:
//    _x                 x  (horizontal) position (mm)
//    _xp                x prime position (mrad)
//    _y                 y  (vertical) position (mm)
//    _yp                y prime position (mrad)
//    _phi               phase angle relative to synchronous particle (rad)
//    _deltaE            energy offset (GeV)
//    _dp_p              momentum dp/p
//    _fractLPosition    Fractional position in longitudinal bin
//    _LPosFactor        Longitudinal weighting factor = local line density
//                       over the average line density.
// Integer Vector
//    _LPositionIndex    Longitudinal bin index
//    _foilHits          Number of foil traversals.
//    _xBin              Integer vector storing the horizontal bin location
//    _yBin              Integer vector storing the vertical bin location
// Real Vector
//    _xFractBin         Fractional postion within a horizontal bin
//    _yFractBin         Fractional postion within a vertical bin
// Real
//    _xMin              Min x of herd (mm)
//    _xMax              Max x of herd (mm)
//    _yMin              Min y of herd (mm)
//    _yMax              Max y of herd (mm)
//    _phiMin            Minimum longitudinal phase of herd (rad)
//    _phiMax            Maximum longitudinal phase of herd (rad)
//    _dEMin             Minimum deltaE of herd (GeV)
//    _dEMax             Maximum deltaE of herd (GeV)
//    _bunchFactor       longitudinal (average/peak) density ratio
// Integer
//    _nMacros           Number of macro particles presently in herd.
//    _feelsHerds        Switch to control interaction with other herds
```

```
//                       0 (default) = no interaction with other herds
//                       1 = feels, but doesn't push other herds
//    _longBinningDone   flag to indicate if longitudinally binned yet
//
// Node Info particular to the herd:
//    _currentNode       the current Ring node the herd is at
//    _nTurnsDone        number of turns the herd has completed
//    _nPartTurnsDone    integral of the number of turns * number of particles
//
//  Void routines:
//    _reSize           - resize (increase) the herd vectors to new size
//    _findXYPExtrema   - find the min/max _x,_y,_phi,extents
//    _findDEExtrema    - find the min/max  deltaE extents
//    _insertMacroPart  - routines to add a macroparticle to a herd.
//    _addLostMacro     - routine to move a single particle from the herd to
//                      - the LostMacroParts object.
//
// SyncPart &
//    _syncPart          Reference to the synchronous particle object
//
// LostMacroParts
//    _lostOnes         - place to store lost macroparticle information.
//
// PROTECTED MEMBERS
//  None
// PRIVATE MEMBERS
//    None.
//
////////////////////////////////////////////////////////////////////////////
```

Table A.3 Description of the Node class members.

```
////////////////////////////////////////////////////////////////////////
//
// CLASS NAME
//    Node
//
// INHERITANCE RELATIONSHIPS
//    Node -> Object -> IOSystem
//
// USING/CONTAINING RELATIONSHIPS
//    Object (U), BSpline (C)
//
// DESCRIPTION
//    An abstract class for storing Node information.
//
// PUBLIC MEMBERS
//    Node:             Constructor for making Node objects
//    ~Node:            Destructor for the Node class.
// String:
//    _name             Name of node
//  Integer:
//    _oindex           Order index
//  Real:
//    _position         Poistion in ring (m)
//    _length           Length of node (m)
//  Void:
//    _nodeCalculator   Routine to do preliminary calculation for this Node.
//    _updatePartAtNode Routine to call to do macroparticle updates
//    _nameOut          Routine to return the node name
//
// PROTECTED MEMBERS
//  None
// PRIVATE MEMBERS
//    None.
//
////////////////////////////////////////////////////////////////////////
```

Table A.4 Description of the DiagnosticBase class members.

```
////////////////////////////////////////////////////////////////////////
//
// CLASS NAME
//   DiagnosticBase
//
// INHERITANCE RELATIONSHIPS
//    DiagnosticBase -> Object
//
// USING/CONTAINING RELATIONSHIPS
//    None.
//
// DESCRIPTION
//    This is a base class for storing Diagnostic information.
//
// PUBLIC MEMBERS
//
//  _mp - reference to the macro particle herd we want info on
//
//  Virtual routines:
//
//  _diagCalculator  - Routine that does the diagnostic calculation
//  _showDiagnostic  - Routine to show information to a stream
//                   - typically used for "human readable" output
//  _dumpDiagnostic  - Routine to dump data to a stream
//                   - typically used for creating file output for processing
//
// PROTECTED MEMBERS
//  None
// PRIVATE MEMBERS
//    None.
//
////////////////////////////////////////////////////////////////////////
```