XFLAT code modules and implementation details

Compilation and Build Instructions:

XFLAT is a command line application, intended to study neutrino flavor oscillations in supernovae environments. The code is C++ implementation with the hybrid architecture that uses SIMD, OpenMP and MPI for performance acceleration. It is capable to be run on heterogeneous supercomputers and can utilize both traditional CPUs and the newer Intel Many Integrated Core Architecture or Intel MIC (Xeon Phi).

The code contains several modules that can be plugged in or out from the build using the provided Makefile. In addition to modules, other features can also be switched on or off from the Makefile. Features such as the usage of SIMD, OpenMP, and MPI are controllable from the Makefile.

The following code features can be switched on or off from the second line of the provided Makefile starting with CXXFLAGS, and using *-D* switch:

`SIMD` # when defined the code will use SIMD instructions

`OMP` # when defined the code will use OpenMP threads

`MMPI` # when defined the code will use MPI

In addition there are multiple modules that can be changed from the same line of the Makefile. These modules can be categorized as follow:

Geometry related modules:

`SA` # Single Angle supernova module

`MA` # Multi Angle supernova module

`MAA` # Multi Azimutal Angle extended supernova module

`CLN` # Cylindrical module

`PNT` # Plane module

`LIN` # Multi-source line module

IO related modules:

`IOF` # performs file IO for each node

`IOFI` # performs indirect IO in which MIC sends its data to CPU first

An example of the line with the usage of IOF and MAA modules which utilizes SIMD, OpenMP and MPI is shown as below:

```
CXXFLAGS = -O3 -openmp -DIOF -DMAA -DSIMD -DOMP -DMMPI
```

The code is using the NetCDF(either version 3 or 4) library for its current IO modules, however, if the NetCDF4 is used, HDF5 library is also required.

In order to build binary for CPU these commands must be issued:

```
$ cp Makefile.cpu Makefile
$ make all
```

Likewise, in order to build Xeon Phi binary these must be issued:

```
$ cp Makefile.phi Makefile
$ make all
```

The CPU binary is called XFLAT.cpu and the Xeon Phi binary is XFLAT.phi. Please note, if the OpenMP feature is on the -openmp flag also must be added to the compiler flags and if the MPI feature is switched on inside the Makefile, some MPI scripts are required so as to run it on multiple nodes.

In order to optimize the code for CPU one has to add the following flag before compilation:

```
-xHOST
```

, and in order to build the code for MIC, the following flag must be added to compiler flags instead of the above flag:

```
-mmic
```

Source Code Directories and Files:

There are two directories that contains header files (*include/*) and source files (*src/*).

The header directory contains these header:

Fenergy.h:

Contains function declarations which are responsible for energy spectra. These functions are called from *NBeam* module.

Fio.h:

Holds function declarations which are responsible for dumping data onto file and other I/O related taks. Depends on the settings, different source files can implement those functions.

Global.h:

Encloses several global settings and constant.

Matt.h:

Matter profile and its related procedures.

NBeam.h:

Encloses Neutrino Beam class declaration. This class contains all the variables and functions for calculating neutrino beam interactions and evolutions. Each neutrino beam in the system must instantiate this class. This module serves as the lower layer module for other upper layer modules.

NBGroup.h:

This module sums up other modules such as Numerical, Physics, and I/O modules as the upper level modules.

Nmr.h:

Holds Numerical related function declaration.

Parser.h:

Encloses Parser class declaration. This class is responsible for taking a config.txt file, parse it, and put the extracted values to the corresponding variables.

Phy.h:

Holds Physics related function declaration. Depends on the geometry, different source files can implement those functions.

Util.h:

Holds auxiliary functions and variables declarations.

The source directory contains these files:

Fenergy.cpp:

Implements the energy spectra functions.

IO_f.cpp:

One of the implementation of *Fio.h* header. In this source file, every node dumps its data directly to its own file.

IO_fi.cpp:

One of the implementation of *Fio.h* header. Due to limited I/O capability of the current generation of Intel MIC, Xeon Phi nodes send their data to corresponding CPUs first, the CPU dumps both their own data and the Phi's data to file.

main.cpp:

Holds the main function of the application.

Matt.cpp:

Implements matter profile functions.

NBeam.cpp:

Implements Neutrino Beam class.

NBGroup.cpp:

Contains functions for initializing and finalizing other modules.

Nmr.cpp:

Implements the neutrino evolution algorithm.

Parser.cpp:

Implements the Parser class.

Phy_CLN.cpp:

This is one of the several Physics modules. This module implements the cylindrical geometry.

Phy_MA.cpp:

This module implements supernova multi-angle geometry.

Phy_MAA.cpp:

This module contains the extended supernova physics. It implements the multi-zenith angle multi-azimuthal angle geometry.

Phy_PLN.cpp:

Implements the Plain geometry.

Phy_SA.cpp:

Contains single-angle supernova physics implementation.

Util.cpp:

This modules holds many general variables and are used in several other modules. It also implements several auxiliary functions.

Inside the *config.txt* file:

In order to run the program, a configuration file has to pass as argument to the program. This file contains several settings related to the behavior of the application and many values for initializing program's variables.

Each keyword in the file must be starts in a new line. Keywords are fixed and cannot be change unless the corresponding keyword in the *Parser.cpp* class implementation is changed accordingly. They contains a fixed character string and ends in a '=' and after a white space its value is stored:

Keyword1= value1

Comments can be added to the configuration file as well. They starts with '#' and continues until reaching the new line. Therefore, they can be added after the values of a keyword, or can be added in a separate line:

 # comment1

Keyword1= value1 # extra comment

Please note that if an expected variable is not initialized in the configuration file, its initial value will be undefined.

The current keywords can be categorized in two different categories: Those that can control the general behavior of the application (mostly related to I/O tasks) and those keywords which initialize physics related variables.

General keywords are listed as follow:

dumpMode=

This keyword expect an integer that set the way data dump onto a file. The values must be read in binary mode. Put 0 (also 0 in binary mode) for the value means no dumping data. Putting 1 (also 1 in binary mode) is the first mode which means dumping a whole snapshot which contains all the wave-function values. The value 2 (10 in binary mode) means dumping only the weighted average over energy bins onto file. The next value can be 4 (100 in binary mode) and so on. Note that in this way values can be combined together as flags in binary mode. Therefore, putting 3 (11 in binary mode) means doing I/O in both the first mode (01 in binary) and second mode (10 in binary).

filePrefix=

This keyword takes a string as the general name for files. In the program, some other strings may attach to it as well. For example when the data dump mode is 1, the “Snapshot” string is also attach to it. Moreover, if multiple files is generated, a counter number starts from 0 is also attached afterwards. Finally, in the MPI runtime, each node will attach its own id to the end of the file (separated from the rest of string with ‘_’).

newFile_step=

Takes an integer value indicating that after the specified I/O job iterations, a new file has to be generated. In this way it is possible to prevent creating a huge file.

sync_step=

Indicated after a specific data dumping iterations, the file has to be synchronized with the disk. In this way it is possible to prevent data loss.

r_step1=

If for each iteration in the evolution loop, data are to be dumped on file, the performance will drop dramatically. In addition, there is no need to perform I/O task for every iteration in the loop as the difference between the two consistent iterations will be negligible. Therefore, the float number of this keyword indicates only after advancing the distance in Km equals to its value, the data can be dumped to file. Hence, there is a distance equals to this keyword’s value between each snapshot. This, keyword only controls the I/O mode of 1, for the second I/O mode he keyword is *r_step2*.

t_step1=

This keyword is the same as the above one. The I/O is only performed after this keyword’s value in seconds. Similar to the previous keyword, it only controls the first I/O mode. For the second I/O mode the keyword is *t_step2*.

itr_step1=

The same as the previous keywords. I/O is performed after every the number of iterations equals to this keyword’s value. It only affects the behavior of the first I/O mode. For the second I/O mode the keyword is *itr_step2*.

start_beam=

end_beam=

These two keywords indicates the starting and ending indices of neutrino beams that the current node has to compute on. For example, the distribution of 1000 beams over two identical nodes can be done in this way: for the first node *start_beam= 0* and *end_beam= 500* and for the second node *start_beam= 500* and *end_beam= 1000* and the 500th beam is the first beam of the second node. The distribution of beams over nodes is depend on the Physics module, for example for the regular supernova environment, the zenith angle beams are distributed over nodes but for the other modules, depends on the geometry the distribution can be over other types of beams such as azimuthal angle beams. If the value of these two keywords is a negative number, in order to find each node's beams range, before distributing the load over nodes, a benchmark code is performed on each one of them and based on the computational capability of each node, the starting and ending indices are defined.

Tn=

The total execution time of the application in second. Note that the time for memory allocation is not included, thus this time is only the allowable time for executing the main evolution loop.

Ts=

The total number of iteration for the main evolution loop. The program will be finished after reaching the *Tn* seconds or after iterating *Ts* numbers.

Keywords related to physics are listed as below:

eps0=

The error tolerance threshold that indicates the maximum allowable error between two computed wave-functions.

ksi=

A float value between 0 and 1 for controlling the adaptive step size behavior. It is basically a safety factor to ensure success on the next try.

dm2=

Neutrino mass-squared difference.

theta=

The vacuum mixing angle.

R0=

The starting radius in Km.

Rn=

The final radius in Km.

dr=

The initial Δr value in Km. Usually less than 1 Km.

max_dr=

The maximum possible value for ‘dr’. The higher values will be trimmed to this value.

E0=

The starting point of the energy spectra in MeV. Mostly 0 MeV.

E1=

The ending point of the energy spectra in MeV.

Abins=

The number of zenith angle bins. The value is always >=1.

Pbins=

The number of azimuthal angle bins. The value is always >=1.

Ebins=

The number of energy bins over the range of the energy spectra. The value is always >=1.

SPoints=

The number of emission surface points, for the multi emission points systems.

Flvs=

The number of neutrino flavors in the system.

Ye=

The electron fraction or the net number of electrons per baryon.

nb0=

The baryon density at the neutrino sphere.

Rv=

The neutrino sphere radius.

Mns=

The mass of the neutron star in solar mass unit.

gs=

The statistical weight in relativistic particles.

S=

The entropy per baryon.

hNS=

The scale height.

Lve=

Lv_e=

Lvt=

Lv_t=

The energy luminosity for electro, anti-electron, tau, and anti-tau neutrinos in erg/s.

Tve=

Tv_e=

Tvt=

Tv_t=

The neutrino temperature for electro, anti-electron, tau, and anti-tau neutrinos in MeV.

eta_ve=

eta_v_e=

eta_vt=

eta_v_t=

The degeneracy parameter for electro, anti-electron, tau, and anti-tau neutrinos.

Modules' Methods and Variables:

Neutrino Beam module's methods (NBeam.h/NBeam.cpp)

This is one of the lower layer modules. It contains the NBeam class, which holds arrays of wave-function. Each element of those arrays represent an energy bin. The NBeam module has the following functions and class:

- `void init(int flavors, int ebins)`: First, this function calls the energy spectra module's initialization method. Next, depends on the number of flavors, it allocated several arrays for storing energy bins' values. The size of each array is determined by the number of energy bins. Afterwards, it fills up energy arrays based on the normalized value of the energy spectra. In addition, it calculates and stores the vacuum Hamiltonian for each bin.
- `void freemem()`: This method free up all the arrays that were allocated in the initialization function.
- `inline void upd_nu_coef( const double *REST nu, const double *REST anu, const double n_cf, const double an_cf, double *REST ret ) throw()`: This inlined function calculates the difference between the multiplication of the energy spectra functions and the density matrices of a neutrino particle and its anti-particle. Each density matrix is weighted by their coefficient: $(\varrho_{\nu_{\underline{\alpha}}}(q',\vartheta')f_{\nu_{\underline{\alpha}}}(q')\frac{L_{\nu_{\underline{\alpha}}}}{\langle E_{\nu_{\underline{\alpha}}}\rangle} - \varrho^{*}_{\overline{\nu}_{\underline{\alpha}}}(q',\vartheta')f_{\overline{\nu}_{\underline{\alpha}}}(q')\frac{L_{\overline{\nu}_{\underline{\alpha}}}}{\langle E_{\overline{\nu}_{\underline{\alpha}}}\rangle})$
- `class NBeam`: This class represent a single neutrino beam that contains arrays of wave-functions expanding over a range of energy spectra. There are several variables and methods inside this class:

    - The constructor: The argument is the index that represent the type of the particle (electron neutrino, anti-electron neutrino, etc.) for the beam:
      ```
      NBeam(int prtc);
      ```

    - Energy spectra function's setter and getter: Two functions are available to receive the value of energy spectra function. The first method receives an index of energy bin and return the value of energy function based on the received index. The second method returns the pointer for the array of energy spectra. The pointer can be used to set or get each energy bins' value separately:
      ```
      inline double Fv(int e) const;
      inline double* Fv() const;
      ```

    - Wave-function's components setter and getter: Sometimes it requires that the wave-function's components can be accessible based on an index (i.e. 0 for the real part of the first number, 1 for the imaginary part of the first number, 2 for the real part of the second number, 3 for the imaginary part of the second number, etc.), therefore the following methods are provided to make wave-function's components accessible based on an index number:
      ```
      inline const double* psi(int cmpn) const throw();
      ```

```
inline double* psi(int cmpn) throw();
```

- Energy bins' setters and getters: These set of functions return the pointer to the components of wave-function arrays. The pointer can be used to set or get each energy bins' value. The following functions are available for the current version of NBeam class:

```
inline double* Ar();
inline double* Ai();
inline double* Br();
inline double* Bi();
inline const double* Ar() const;
inline const double* Ai() const;
inline const double* Br() const;
inline const double* Bi() const;
```

- Setters and Getters: These set of functions can set the value of each energy bin or return the current value of each energy bin. For each wave-function component, there has to be at least one setter and getter functions. The method's argument is the index of the required energy bin. The following functions are available for the current version of NBeam class:

```
inline double& Ar(const int e);
inline double& Ai(const int e);
inline double& Br(const int e);
inline double& Bi(const int e);
inline const double& Ar(const int e);
inline const double& Ai(const int e);
inline const double& Br(const int e);
inline const double& Bi(const int e);
```

- Density matrix: The density matrix is calculated from the wave-function. A wave-function with two complex components has a 2x2 complex density matrix. Yet, since the second row in the matrix can be constructed from the first row, the provided methods only return the computed first row of the density matrix. The returned value is in a four-element array. There are two possible ways to compute the density matrix, one is by passing an energy bin index, and the other is by passing the wave-function components:

```
inline void density(const int ebin, Res_t ret) const throw();
inline void density(const double ar, const double ai, const double
br, const double bi, Res_t ret) const throw();
```

- Neutrinos' evolution: After computing the Hamiltonian, a method is required to evolve the current wave-functions using the Hamiltonian. There are two methods that can be used for the neutrinos' evolution. The first function, take an energy bin index, delta-radius, the Hamiltonian, and returns the components of the evolved wave-function. The second method takes delta-radius, the Hamiltonian, the components of the current wave-function, and returns the components of the new wave-function:

```
inline void U(  const int e, const double dr, const double h_r0, const double h_i0, const double h_r1, const double h_i1, const double n_ar, const double n_ai, const double n_br, const double n_bi) throw();
inline void U(  const double dr, const double h_r0, const double h_i0, const double h_r1, const double h_i1, const double a_r, const double a_i, const double b_r, const double b_i, double& n_ar, double& n_ai, double& n_br, double& n_bi) const throw();
```

- Summation over energy bins: The summation over all energy bins is necessary in order to calculate the Hamiltonian, thus a function is provided to calculate the sum and store inside the class:
  ```
  void calcHSum() throw();
  ```

- Energy bins summation: This method returns the previously computed summation over energy bins. The returned value is in a four-element array that is the first row of the summation matix:
  ```
  void getHSum(Res_t ret) const throw();
  ```

- Neutrino Beam Evolution: These set of functions take another neutrino beam and based on the neutrino-neutrino background Hamiltonian, matter potential, and neutrinos' mass difference term, evolves the wave-functions:
  ```
  void evolveBinsAvgErr( const NBeam& beam, const int ptc_idx, const double dr, const double *REST hvv, const double hmatt, NBeam& beamAvg, NBeam& beamErr ) throw();
  void evolveBinsHvvAvg( const NBeam& beam, const int ptc_idx, const double dr, const double *REST hvv, const double hmatt, NBeam& beamAvg ) throw();
  void evolveBinsAvg( const NBeam& beam, const int ptc_idx, const double dr, const double *REST hvv, const double hmatt, NBeam& beamAvg ) throw();
  void evolveBinsAvg( const int ptc_idx, const double dr, const double *REST hvv, const double hmatt, NBeam& beamAvg ) throw();
  void evolveBinsHvv( const NBeam& beam, const int ptc_idx, const double dr, const double *REST hvv, const double hmatt ) throw();
  void evolveBins( const NBeam& beam, const int ptc_idx, const double dr, const double *REST hvv, const double hmatt ) throw();
  void evolveBins( const int ptc_idx, const double dr, const double *REST hvv, const double hmatt ) throw();
  ```
  Other than the neutrinos' evolution, some of them perform other task such as calculating the summation of energy bins for the background Hamiltonian or taking the average between two neutrino beams or calculating the maximum error between two neutrino beams. Since function fusing can increase the overall performance, it is recommended to use the fused version of the function to perform more tasks on the same data. The particle

index (0 for electron neutrino, 1 for anti-electron neutrino, 2 for mu neutrino, 3 for anti-mu neutrino, etc.), the delta-radius, the neutrino-neutrino background and the matter potential are passed as arguments too.

- Average of two beams: There is also a function that take a neutrino beam, then take the average between it and the current beam, and store the result inside the class:
  ```
  void addAvg( const NBeam& beam ) throw();
  ```

- Find the maximum error between two neutrino beams: In order to detect whether or not the current delta-radius is enough for calculations, the maximum error between two neutrino beams must be computed. There are two functions that take a neutrino beam and calculate and return the maximum error over all energy bins. The second function also compute the summation over energy bins which can be used in future for the Hamiltonian computations:
  ```
  double calcErr( const NBeam& beam ) const throw();
  double calcErrHvv( const NBeam& beam ) throw();
  ```

- Return maximum error: If the maximum error is computed in one of the fused function before, this function only returns it:
  ```
  double& getErr() throw();
  ```

Numeric module's methods (Nmr.h/Nmr.cpp)

This module is responsible for the numeric algorithm, and can be replaced with the other numerical modules using different algorithms. There are several functions in this module that have to be implemented:

- `int init(int len)`: This function is the first function to be called for this module. It takes an integer argument that is the length of the neutrino beams (#of trajectory beams X #of particles). Afterwards, it allocates the memory for the entire neutrino beam's arrays. The number of arrays may be varied and depends on the algorithm. Next, it calls the `initBeam()` function of the physics module for each of those arrays, to initialize them.
- `void freemem()`: Calls the `freeBeam()` function of the physics module then perform neutrino beams' deallocation.
- `int evolutionLoop() throw()`: This function contains the main neutrinos' evolution loop. In fact, it implements the numeric algorithm. If MPI is enabled, there are several points that nodes exchange data. First, inside the main evolution loop, the master node sends the termination condition, current radius, and delta-radius to all of the nodes. Next, it calculates the distance of the advancement based on the number of middle points. Then, the program calculates the matter density profile and afterwards, the Hamiltonian computations are performed. The current algorithm continues with the neutrino beam evolutions, error calculations, and if the program is in the multi-node mode, all the nodes exchange maximum local errors in order to find maximum global error. Next, if the maximum error is less that a predefined error threshold an IO module's function is called to perform any necessary IO operations. Finally, it goes through the next iteration.

Physics modules' methods (Phy.h/Phy_*.cpp)

Currently, there are several physics related module each having different geometry and physics. All of them have to implement at least all the methods inside the header file (Phy.h). These general methods are as the following:

- `void init()`: Initilization of the physics module such as memory allocations in a specific order and arrays inits.
- `void freemem()`: Deallocate all the allocated memories for this module
- `int beamLen()`: Returns the total length of the current beam. It can be used from within the other modules to find out about the neutrino's beam length for allocation purposes (e.g. theta_bins x phi_bins x num_of_particles)
- `int getDim()`: Returns the number of data dimensions of the plugged-in physics module, which is used from the I/O modules to format the data for NetCDF file (e.g. 6 = [r, theta, phi, num_of_particles, wavefunction's components, e])
- `void getDimInfo(std::string str[])`: Returns an array of names of data dimension (e.g. 6 = ["r", "theta", "phi", "prtcl", "comp", "ebin"])
- `size_t*startDim()`: Returns an array of starting point of each data dimension, to be used by I/O module (e.g. [current_rad, 0, 0, 0, 0, 0])
- `size_t* countDim()`: Returns an array of the length of each data dimension to determine what te size of the current snapshot is, to be used by I/O module (e.g. [1, theta_bins_num, phi_bins_num, num_of_particles, num_of_components, num_of_energyBins])
- `int& startBeamIdx()`: Returns the start number for the dimension that is distributed over nodes, for the current node (e.g. 0 for the first node, 500 for the second node, 1000 for the third node, 2000 for the fourth node)
- `int& endBeamIdx()`: Returns the length of the dimension that is distributed over nodes, for the current node (e.g. 500 for the first node, 1000 for the second node, 2000 for the third node, 3000 for the fourth node)
- `int firstDimLen()`: Returns the size of the dimension on which the problem size is distributed over nodes (e.g. depends on the module can be theta_bins, or phi_bins, …)
- `void initBeam(NBeam* beam)`: Takes a NBeam class array and initializes each class's internal arrays (Particles and Ebins) accordingly by calling each class's constructor.
- `void freeBeam(NBeam* beam)`: Takes a NBeam class array Call deconstructor of each class.
- `void calcAngleBins(const double r, const int step_num)`: Usually calculates and caches the cosine bins at radius 'r' and for different step numbers (current point, mid-point, full-point). Depends on the module may calculates other angle bins as well.
- `void calcDeltaLs(const double dr, const int cur_pnt, const int s_pnt, const int e_pnt)`: Calculates and caches 'dl' for each angle bins. The 'dr' is the radius difference, the cur_pnt is the point at which the 'dl' is calculated and the last two points are those at which the average is taken. (i.e. dl[cur_pnt] = dr / (.5 * (cos[s_pnt]+cos[e_pnt]))
- `void newHvv(double*& hvv)`: Allocates memory for an array of Hamiltonians;
- `void deleteHvv(double*& hvv)`: Deletes allocated memory for Hamiltonian array.
- The following methods, take an input array of NBeam objects, evolve them into an output array of NBeam objects using the matter profile and Hamiltonian array 'hvv'. They may perform other

tasks such as partiall summation of Hvv over energy bins, or taking the average of two NBeam arrays and store them into the last parameter:

- `void evolve(const nbm::NBeam *REST ibeam, const int pnt, const double *REST hvv, const double hmatt, nbm::NBeam *REST obeam) throw();`
- `void evolveHvv(const nbm::NBeam *REST ibeam, const int pnt, const double *REST hvv, const double hmatt, nbm::NBeam *REST obeam) throw();`
- `void evolveAvg(const nbm::NBeam *REST ibeam, const int pnt, const double *REST hvv, const double hmatt, nbm::NBeam *REST obeam, nbm::NBeam *REST obeamAvg) throw();`
- `void evolveHvvAvg(const nbm::NBeam *REST ibeam, const int pnt, const double *REST hvv, const double hmatt, nbm::NBeam *REST obeam, nbm::NBeam *REST obeamAvg) throw();`
- `void evolveAvgErr(const nbm::NBeam *REST ibeam, const int pnt, const double *REST hvv, const double hmatt, nbm::NBeam *REST obeam, nbm::NBeam *REST obeamAvg, nbm::NBeam *REST obeamErr) throw();`
- `void evolveAvg(const int pnt, const double *REST hvv, const double hmatt, nbm::NBeam *REST iobeam, nbm::NBeam *REST obeamAvg) throw();`

- `void avgBeam(const NBeam *REST ibeam, NBeam *REST obeam)`: This function only takes the average of the two input beams and store the result into the second beam.

I/O module's methods (Fio.h/IO_f.cpp, IO_fi.cpp)

There are two I/O modules available. When the first one (IO_f.cpp) is plugged-in, each node will dump data onto its own file using NetCDF. However, since I/O performance is very poor on Xeon Phi and may cause a serious bottleneck on heterogeneous environments, another module is provided (IO_fi.cpp) that indirectly send Xeon Phi data to the corresponding CPU. Therefore, CPU is responsible to dump its own data as well as Xeon Phi. Here are the public methods for the I/O modules:

- `void init(int file_counter=0)`: Initializes the module. It also takes a counter number which indicates the number of generated files so far. This is because sometimes it is not possible to store all the data onto a single file. Thus, this module can be called from I/O module for initializing another file.
- `void freemem()`: Deallocate all the allocated memories for this module.
- `void fillInitData(NBeam* nubeam)`: It is possible to resume the code using the previously generated data file. In that case, this function is called and it takes an array of NBeam objects to initializes it using the provided data file. The data file can be the second argument to the code.
- `void dumpToFile(const NBeam *REST nubeam, const int itr, const double r)`: This is the main function for dumping data. It takes an array of NBeam objects as well as the current iteration and radius. If the current iteration or radius have reached some thresholds, it dumps data onto file.