FOORT Documentation

Daniel Mayerson

November 2022

Welcome to the documentation for FOORT, which will make using FOORT but a small efFOORT!

1 Installation

1.1 Libconfig library

1.1.1 Ubuntu systems

- `sudo apt-get update -y`
  to make sure `apt-get` is up to date.
- `sudo apt-get install -y libconfig-dev`
  to install the package.
- `export LD_LIBRARY_PATH=/usr/local/lib`
  If necessary, change `/usr/local/lib` to your actually library folder.

1.1.2 Arch systems

Thanks to Suvendu!!

- `sudo pacman -Syu libconfig`
  to install the package.

1.2 Compiling

To compile under Windows with VS, open the `FOORT.sln` file with VS. (VS 2022 was used to create FOORT.)

To compile under Linux, in the `FOORT` subfolder (where all code files are), there is a `makefile`, so running the command `make` should compile FOORT if `g++` and `libconfig` are correctly installed. (The command `make clean` should remove FOORT and/or all intermediate object `.o` files that are created while compiling.)

Note: Under some Linux systems, it is possible that the `LDFLAGS` setting needs to be changed to:

```
LDFLAGS = -lm -lstdc++ -L$(LD_LIBRARY_PATH) -lconfig++
```

before it will compile correctly. Once compiled, FOORT will hopefully run correctly as long as `LD_LIBRARY_PATH` has been set correctly (see above under installation of libconfig).

There is also a file `makefile_precompiledmode`. This can be used to compile FOORT without installing libconfig, *but only if* configuration mode has been turned off in the FOORT source file. To do this, comment out (or remove entirely) the line

`#define CONFIGURATION_MODE`
at the beginning of `Config.h`. FOORT will then not need libconfig to compile, and will run use the precompiled settings that are set in `Main.cpp` in the function `LoadPrecompiledOptions()`.

2 Configuration Options

2.1 `Metric`

This heading is where the metric is selected and all its options set. The first option to set is the type of metric:

- `Name = "MetricName";`
  where MetricName is one of the following:
  - `Kerr`: Kerr in normal Boyer-Lindquist coordinates, with the units chosen such that $M = 1$.
  - `FlatSpace`: flat space in spherical coordinates.

  By default (if this setting is not specified in the configuration), the Kerr metric will be selected.

Depending on which metric has been selected, there can be additional options to be specified:

- `Kerr`:
  - `a = real;`
    where `real` is a number between -1 and 1; FOORT will give a warning if this is not satisfied. (Default: 0.5.)
  - `RLogScale = bool;`
    where `bool` is `true` or `false`. If `true`, then the Kerr metric will be evaluated in the logarithmic radial coordinate $u = \log r$; this leads to better integration around the horizon. (Default: `false`.)
- `FlatSpace`: No further options to specify.
- `Rasheed-Larsen`:
  - `m = real; a = real; p = real; q = real;`
    These specify the parameters for the Rasheed-Larsen black hole metric. Note that the parameters must satisfy $p, q \geq 2m$, $a^2 < m^2$ and $m > 0$; FOORT will give a warning if this is not satisfied. The parameters (which all have dimensions of length) will always be rescaled so that the mass of the black hole is $M = 1$ (i.e. they are measured in units of the black hole mass); note that $M = (p + q)/4$. The horizon is at $r_+ = m + \sqrt{m^2 - a^2}$. (Defaults: $m = 1, a = 0.5, p = q = 2$.)
  - `RLogScale = bool;`
    Same logarithmic radius as for the Kerr metric, see above.

2.2 Source

This heading is where you determine a source for the geodesic equation (i.e. a force on the right-hand side for the geodesic equation, making the geodesic fail to be an actual geodesic). Currently, only `NoSource` is implemented (i.e. no rhs to the geodesic equation):

- `Source = "SourceName";`
  where currently SourceName can only be `NoSource` (and there are no further options to specify).

2.3 Diagnostics

Inside the heading `Diagnostics`, each `Diagnostic` has its own subheading. Under their own subheading, the most important options to specify is:

- `On = bool;`
  where `bool` is `true` is you want the Diagnostic to be turned on and `false` otherwise. Only if the Diagnostic is turned on will the other options in the Diagnostic's subheading be considered.

- `UseForMesh = bool;`
  where `bool` is `true` if this is the Diagnostic that should be used in the Mesh to determine "values" to assign to geodesics and "distances" between neigboring geodesics, which the Mesh will use to determine where to refine. Note: only one Diagnostic can have this setting set to `true`!

By default, if no diagnostic sets `UseForMesh` to `true`, `FourColorScreen` will be used if it is turned on. If `FourColorScreen` is not turned on, then all Diagnostic settings revert to the default of only turning on `FourColorScreen`. If more than one Diagnostic has `UseForMesh` set to `true`, the first one to set it will be used (in the order of Diagnostics given below).

The different possible Diagnostic subheadings and further options therein are:

- `FourColorScreen`: This is the four-color screen, assigning an integer 1-4 to the geodesic according to which quadrant of the boundary sphere it has escaped to. Geodesics that do not terminate due to exiting the boundary sphere (due to e.g. timing out or falling into a horizon) will be assigned 0. There are no other settings that `FourColorScreen` takes.

- `GeodesicPosition`: This keeps track of the position of the geodesic along its integration and outputs all those positions; this can be used for e.g. plotting individual geodesic trajectories. The options to specify are:
  - `UpdateFrequency = int;`
    with `int` a number larger or equal to zero. The Diagnostic will only store a position every number of steps as indicated here. If this is 0, the Diagnostic does not update the position during integration, but may update at start and finish according to the following two options. (Default is 1.)
  - `UpdateStart = bool;`
    (Only relevant if `UpdateFrequency = 0`.) Decides whether the Diagnostic stores the position of the geodesic at the very start of integration.
  - `UpdateFinish = bool;`
    (Only relevant if `UpdateFrequency = 0`.) Decides whether the Diagnostic stores the position of the geodesic at the very end of integration.

- `OutputSteps = int;`
  Here, `int` is the (approximate!) number of positions per geodesic that will be outputted to file. If the Diagnostic has stored more than this many positions during the integration of the geodesic, the Diagnostic will select only (approximately) this many steps to output to file, evenly selected among the stored positions. If this is set to 0, it outputs all steps. (Default is 0.)

- `EquatorialPasses`: This keeps track of the number of passes through the equator $\theta = \pi/2$ that the geodesic makes. It returns a positive number for geodesics that have escapes out the boundary sphere, and a negative number for geodesics that eventually fall in the horizon. This Diagnostic also takes the options `UpdateFrequency`, `UpdateStart`, `UpdateFinish` just as `GeodesicPosition` does above. In addition, it takes the option:

  - `Threshold = real;`
    Only consider a geodesic "above" or "below" the equatorial plane if its θ coordinate is `Threshold`$\times\pi/2$ above or below $\pi/2$. This threshold is mainly useful for (near-)equatorial camera positions, since in that case without a threshold there is often a discontinuity in equatorial passes between the top and bottom half of the screen. (Default is 0.01.)

2.4 Terminations

Terminations are set very similar to Diagnostics as discussed above. Each Termination has its own subheading, with most important option to specify:

- `On = bool;`
  which determines if the Termination is used or not.

By default (e.g. if no Terminations are turned on, or there is no `Terminations` subheading), `TimeOut` and `BoundarySphere` will be turned on with the default options set as indicated below.

The different possible Termination subheadings and further options therein are:

- `Horizon`: If the metric has a horizon at a constant r radius, then we want to stop integration before we get to the horizon, since usually the horizon is a coordinate singularity where geodesics spend an infinite amount of coordinate time approaching. (FOORT will check whether the `Metric` is a descendant class of `SphericalHorizonMetric`; if not, this Termination will be turned off.) Options to set are:

  - `Epsilon_Horizon = real;`
    Here, `real` is the percentage of the horizon radius at which to terminate the geodesic above the horizon radius. For example, if $r = r_h$ is the horizon and `real =`ϵ, then integration will be terminated at $r = r_h(1+\epsilon)$. By default, this is set to 0.01.
  - `UpdateFrequency = int;`
    Check this condition every `int` steps of the geodesic integration. By default, this is set to 1.

- `BoundarySphere`: A (fictitious) boundary sphere at a constant radius far away from the object; if the geodesic escapes *outside* this sphere, we stop integration. Options to set:

- `SphereRadius = real;`
  This radius `real` is the large radius at which to terminate integration. By default, set to 1000.
  Warning: if using a logarithmic r coordinate for a metric with horizon (see above), then adjust this accordingly! For example, for a boundary sphere radius of $r_b = 1000$ using the usual r coordinate, when using a logarithmic r coordinate, `SphereRadius = 6.9077` is the appropriate value!

- `UpdateFrequency = int;`
  This is analogous to the same setting under `Horizon` above. (Default is 1.)

- `TimeOut`: Terminate the geodesic integration if it has taken too many integration steps. This sometimes happens if a geodesic travels too close to a coordinate singularity. Options to set are:

  - `MaxSteps = int;`
    The maximum number of integration steps allowed before the geodesic terminates. Default is 10000.

  - `UpdateFrequency = int;`
    Analogous to the same setting under `Horizon` above. Default is 1.
    Warning: if this is not set to 1, then the effective maximum number of integration steps that the geodesic is allowed before being terminated is `MaxSteps*UpdateFrequency`

2.5 `ViewScreen`

Under this subheading, all options are set that determine the location, size, orientation, etc. of the viewing screen. The camera is assumed to be a spherical point-sized camera with a certain aperture; the "screen size" is then determined by this aperture — the "screen" is taken to be an square in the plane going through $r = 0$ and perpendicular to the viewing direction.

The options that can be specified are:

- `Position = { t = real; r = real; theta = real; phi = real; }`
  This specifies the position of the camera. (Default is $(0, 1000, \pi/2, 0)$.)
  Warning: If the initial radius of the camera is *outside* of the boundary sphere radius set in the `Horizon` Termination, then all geodesics will terminate immediately after zero steps!

- `Direction = { t = real; r = real; theta = real; phi = real; }`
  This specifies the direction that the camera is pointing in.
  Warning: Currently, the camera is *always* pointing straight towards $r = 0$ (i.e. the direction vector is $(0, -1, 0, 0)$); this option is effectively ignored.

- `ScreenSize = { x = real; y = real; }`
  This sets the screen size. The screen is centered around $r = 0$. For example, when the Kerr metric is selected, units are such that $M = 1$; when $a = 0$, the shadow of the Schwarzschild black hole is a circle with radius $\sqrt{27}$ on the viewing screen. (Default: $(10, 10)$.)

Finally, there is a subheading `Mesh`, under which the Mesh settings should be set. The Mesh is what determines which pixels on the viewing screen are integrated. The options under the `Mesh` subheading are:

- `Type = "MeshName";`
  This indicates the type of Mesh to be used. The possibilities for MeshName are:

  – `SimpleSquareMesh`: This is a square mesh with equal number of pixels in length and width, and pixels evenly distributed over the viewing screen.

  – `InputCertainPixelsMesh`: On a square mesh (such as defined in `SimpleSquareMesh`), have the user input some pixels to be integrated. This can be useful for e.g. integrating only a few geodesics with the Diagnostic `GeodesicPosition` turned on, to be able to follow and plot these few geodesics along their trajectories.

  – `SquareSubdivisionMesh`: This Mesh starts in a first integration iteration with a simple square mesh (such as in `SimpleSquareMesh`), and then further subdivides certain squares into smaller squares in following iterations. It uses the Diagnostic that has `UseForMesh` turned on (see above) to determine "values" and "distances" between geodesics/pixels, which is used to determine which pixel squares to subdivide.

When no Mesh is set with the `Type` option, the default is a `SimpleSquareMesh` (with additional default options as listed below).

Depending on which Mesh is selected with the `Type` option, there are different additional options to set:

- `SimpleSquareMesh`:

  – `TotalPixels = int;`
  This specifies the total number of pixels in the entire grid. If this is not a perfect square, it will be rounded downwards to the nearest perfect square (because each row and column must have an integer number of pixels). (Default: 10000.)

- `InputCertainPixelsMesh`:

  – `TotalPixels = int;`
  This is the same option as for `SimpleSquareMesh`, since the user will be inputting pixels to integrate which live on such a square grid. (Default: 10000.)

- `SquareSubdivisionMesh`:

  – `InitialPixels = int;`
  This is similar to the option `TotalPixels` for `SimpleSquareMesh`: the option `InitialPixels` determines the initial square (equally spaced) pixels to be integrated in the first integration loop. (Again, if it is not a perfect square, it will be rounded down to the nearest perfect square.) (Default: 100.)

  – `MaxPixels = int;`
  This is the maximum number of pixels that can be integrated *in total*, i.e. over *all* integration loops combined. If this is set to 0, then there is no maximum number (integration will continue until the maximum subdivision has been reached or every pixel's weight is zero). (Default: 100.)

  – `MaxSubdivide = int;`
  This sets the maximum number of times a square of pixels can be subdivided. Note that the initial grid has subdivision level 1. (Default: 1, which is also what will be used if an invalid number smaller than 1 is given.)
  Note that, when `InitialPixels =` n^2 and `MaxSubdivide =` d, the total square grid size of pixels will then be given by $((n-1)\cdot 2^{d-1}+1)^2$.

– `IterationPixels = int;`
  At each new integration loop, the Mesh is allowed to select `IterationPixels` squares that it wishes to subdivide; this means that at most 5×`IterationPixels` new geodesics will be integrated in each integration loop (except the first iteration); there could be less geodesics integrated if some of the pixels necessary to subdivide a square already existed previously. (Default: 100.)
– `InitialSubdivisionToFinal = bool;`
  Normally, the Mesh selects only squares to subdivide with non-zero "weight", that is, squares where the "distance" (as determined by the value Diagnostic) is non-zero. However, if this option is set to `true`, then the Mesh will want to continue subdividing *any* square once it has been subdivided at least once (after the initial grid). (Default: `false`.)

2.6 Integrator

Under this subheading, the integration scheme is chosen that is used to integrate the geodesic equation, and other options set. The integration scheme is set by the option:

- `Type = "IntegratorName";`
  Currently, only `RK4` can be used as IntegratorName (this is also the default). This specifies the fourth order Runge-Kutta integration scheme.

There is one other option to set here:

- `StepSize = real;`
  This sets the *basic* step size (i.e. change in affine parameter) for a single integration step. This is adjusted dynamically according to the current position and velocity of the geodesic. By default, this is 0.03.

2.7 Output

Under this heading, you can configure how the geodesics' will be written to file. You can also set the level of detail outputted to the screen (console) during the running of the program:

- `ScreenOutputLevel = int;`
  Here, `int` must be an integer between 0 and 4 (inclusive). 0 means no output to the screen at all except important warnings; 4 is all possible messages including any debug messages. (Default is 4.)

The options that are available to set that determine the format of the file name(s) to which the output is written are:

- `FilePrefix = "prefix";`
  All output files will start with "prefix". If this is not specified, FOORT will do all output to the console.

- `FileExtension = "extension";`
  All output files will end with ".extension". If this is not specified or given as an empty string, the output files will have no extension.

- `TimeStamp = bool;`
  `bool` is true if every output file contains a time stamp (with the local system time and date of when FOORT starts).

The files created will then have names such as "prefix_TIME_DiagName.extension" or "prefix_TIME_DiagName_n.extension" for $n > 1$, where "DiagName" is a short (space-less) name of the Diagnostic whose data is outputted in this file. There will be a different file for each Diagnostic turned on. n is the number of file written, which is only part of the file name for $n > 1$. The "TIME" timestamp is of the form "yymmdd-hhmmss" (year - month - day, hour - minute - second). Note that FOORT will create any directory structures necessary to create the file.

The options that further specify details of the actual output written to files are:

- `GeodesicsToCache = int;`
  Indicates how many geodesics should be cached (stored in memory) before writing everything in memory to file. By default, this is essentially infinite. (It is actually the maximum number storeable in a `unsigned long`, which is $\sim 10^{10}$).

- `GeodesicsPerFile = int;`
  Indicates how many geodesics should be written per file. If more than this number of geodesics are integrated, additional files with filenames ending in `_n` (before the specified file extension) will be created. By default, is essentially infinite (just as `GeodesicsToCache` above). (If 0 is specified, the default will be used.)

- `FirstLineInfo = bool;`
  If set to `true`, will output on the first line of every output file a string that gives descriptive data of all the objects used in integration (Metric, Source, ViewScreen, etc.)

3 Processing Output

(what is outputted in the files etc)

4 Procedures to Add New Objects

For every object to add, there are a certain number of tasks to be done. Per object, these are enumerated below. Within the source files, these add points are also indicated and sample source code is given. These steps below and the add points in the code make adding a new object not a big efFOORT at all!

4.1 Adding a new Metric

A. *At the end of `Metric.h`*: Add the declaration for your new `Metric` class here, inheriting publically from the base `Metric` class (or from `SphericalHorizonMetric`, if your metric has a horizon at a constant radius). It is good practice to make the class (and all overriding virtual functions) `final`, unless further descendant `Metric` classes are possible. It is also good practice to make the member variables (metric parameters) `private` and `const` (and initialized in a constructor) since the metric should not change after initialization.

   It is necessary to override the basic metric getter functions `getMetric_dd` and `getMetric_uu`, which return the metric with indices down or up. It is also recommended to overload `GetDescriptionString()`, although this has been already implemented in the base class `Metric`.

The implementations (definitions) of these functions can then go in `Metric.cpp` (or a different source code file).

B. *In `Config.cpp`, in function `Config::GetMetric()`*: You must add a new `else if` clause checking for if your new metric type has been specified. If it has, then proceed to look up any other parameters necessary for the metric before creating a new instance of it in the variable `TheMetric`.

4.2 Adding a new Diagnostic

A.1. *In the middle of `Diagnostics.h`, after all other `Diagnostic` class declarations*: Declare your new `Diagnostic` class here, inheriting publically from `Diagnostic`. As with `Metric`, it is good practice to make this new `Diagnostic` and all its overriding functions `final`. The definitions of the member functions of your class can then be given in `Diagnostics.cpp` (or a different source code file). The necessary functions to override are:

- A constructor that passes along the `const` pointer to the (owner) `Geodesic` to the base class constructor.
- `UpdateData()`: here your `Diagnostic` updates itself according to the current state of its owner `Geodesic`.
- `getFullDataStr()`: this is the full output string of the `Diagnostic` for its owner `Geodesic`, as should be written to a file.
- `getFinalDataVal()`: this returns a vector of real numbers that indicates the final "value" that should be associated to its owner `Geodesic`.
- `FinalDataValDistance(...)`: this returns a "distance" between two geodesics based on their two final "values" (as given by `getFinalDataVal()`. This distance is used by adaptive mesh(es) to decide where to refine the mesh.
- `getNameStr()`: This returns a short string without spaces that will be appended to the file name, typically just a short name identifying the `Diagnostic`.
- `getFullDescriptionStr()`: This returns a longer, descriptive string (with spaces allowed) of the `Diagnostic` and any relevant options that are selected. This will e.g. be outputted to the screen at runtime to indicate which Diagnostics are turned on.

Finally, if the `Diagnostic` needs to specify options, a declaration of its `DiagnosticOptions` should be given here. Note that this is a `std::unique_ptr` and is `static`: the reason is that these options get set once (at the start of the program), and then remain the same — i.e. for all instances of the `Diagnostic` that belong to different instances of `Geodesics`.

A.2. *(Optional) At the end of `Diagnostics.h`, after all other `DiagnosticOptions` struct declarations*: Declare and define your new `DiagnosticOptions` struct here, if your `Diagnostic` needs additional options over the standard `UpdateFrequency` ones provided by the base struct `DiagnosticOptions`. As indicated in the code, all member variables should be `const` but `public`, and initialized in the constructor. Make sure your constructor passes along the `UpdateFrequency` struct information to the base struct constructor.

As mentioned above, the `DiagnosticOptions` are `static` members of their owning `Diagnostic` class; they get set at the beginning of the program and are afterwards immutable.

B. *At the beginning of* ***`Diagnostics.h`*** *(at the definitions of the bitflags)*: Define a new `DiagBitflag` for your new `Diagnostic`. Make sure to use a bitflag that has not been used before! (All defined bitflags are here, so it should be clear what has been used already.)

C. *At the beginning of* ***`Diagnostics.cpp`****, in the definition of* ***`CreateDiagnosticVector`***: add an appropriate `if` clause checking to see if `diagflags` contains the newly defined `DiagBitflag` (which was defined in point B. above), and if so adds an instance of the `Diagnostic` to the `Diagnostic` vector being created. If it is additionally the value `Diagnostic` (in `valdiag`), then rotate the resulting vector such that it is at the front of the vector.

D.1. *At the beginning of* ***`Config.cpp`***: If your Diagnostic carries a `static DiagnosticOptions struct` (whether it is a new descendant of `DiagnosticOptions` or not), then it must be declared here!

D.2. *In* ***`Config.cpp`****, in function* ***`Config::InitializeDiagnostics()`***: Add an `if` clause here checking if your Diagnostic is turned on in the configuration file. If it is, intialize the relevant options and set the bitflags appropriately.

4.3 Adding a new Termination

A.1. *At the end of the declarations of the* ***`Termination`*** *classes in* ***`Terminations.h`***: Declare your new `Termination` class here, inheriting publically from `Termination`. The definitions of the member functions of your class can then be given in `Terminations.cpp` (or a different source code file). The necessary functions to override are:

- A constructor that passes along the `const` pointer to the (owner) `Geodesic` to the base class constructor.
- `CheckTermination()`: which checks if the termination condition has been reached; if not, returns `Term::Continue`, if so, returns the new termination condition (see below in B.2.).
- `getFullDescriptionStr()`: This returns a descriptive string (with spaces allowed) of the `Termination` and any relevant options that are selected. This will e.g. be outputted to the screen at runtime to indicate which Terminations are turned on.

Finally, most likely the `Terminations` needs to specify options, so a declaration of its `TerminationOptions` should be given here. Note that this is a `std::unique_ptr` and is `static`: as for `DiagnosticOptions`, these options get set once (at the start of the program), and then remain the same — i.e. for all instances of the `Termination` that belong to different instances of `Geodesics`.

A.2. *(Optional) At the end of* ***`Terminations.h`***: Declare and define your new `TerminationOptions` struct here, if necessary, inheriting publically from `TerminationOptions`. Just as for `DiagnosticOptions`, all member variables should be `const` but `public`, and initialized in the constructor. Make sure your constructor passes along the `UpdateFrequency` struct information to the base struct constructor.

B.1. *At the beginning of `Terminations.h` (at the definitions of the bitflags)*: Define a new `TermBitflag` for your new `Termination`.

B.2. *In the definition of `enum class Term` in `Terminations.h`*: Add a new termination condition in this class that can be set by your new `Termination`.

C. *At the beginning of `Terminations.cpp`, in the definition of `CreateTerminationVector`*: add an appropriate `if` clause checking to see if `termflags` contains the newly defined `TermBitflag` (which was defined in point B.1 above), and if so adds an instance of the `Termination` to the `Termination` vector being created.

D.1. *At the beginning of `Config.cpp`*: If your Termination carries a `static TerminationOptions struct` (whether it is a new descendant of `TerminationOptions` or not), then it must be declared here!

D.2. *In `Config.cpp`, in function `Config::InitializeTerminations()`*: Add an `if` clause here checking if your Termination is turned on in the configuration file. If it is, intialize the relevant options and set the bitflags appropriately.