FOORT

1.0

Generated by Doxygen 1.12.0

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 Config Namespace Reference

Typedefs

- using ConfigCollection = ConfigReader::ConfigCollection
- using SettingError = std::invalid_argument

Functions

- void InitializeScreenOutput (const ConfigCollection &theCfg)
- std::unique_ptr< Metric > GetMetric (const ConfigCollection &theCfg)

  Config::GetMetric(): Use configuration to create the correct Metric with specified parameters.
- std::unique_ptr< Source > GetSource (const ConfigCollection &theCfg, const Metric ∗const theMetric)

  Config::GetSource(): Use configuration to create the correct Source with specified parameters.
- void InitializeDiagnostics (const ConfigCollection &theCfg, DiagBitflag &alldiags, DiagBitflag &valdiag, const Metric ∗const theMetric)

  Config::InitializeDiagnostics(): Use configuration to set the Diagnostics bitflag appropriately; initialize all DiagnosticOptions for all Diagnostics that are turned on; and set bitflag for diagnostic to be used for coarseness evaluating in Mesh.
- void InitializeTerminations (const ConfigCollection &theCfg, TermBitflag &allterms, const Metric ∗const theMetric)

  Config::InitializeTerminations(): Use configuration to set the Termination bitflag appropriately; and initialize all TerminationOptions for all Terminations that are turned on.
- std::unique_ptr< ViewScreen > GetViewScreen (const ConfigCollection &theCfg, DiagBitflag valdiag, const Metric ∗const theMetric)

  Config::GetViewScreen(): Use configuration to create the ViewScreen object; with options set according to the configuration.
- std::unique_ptr< Mesh > GetMesh (const ConfigCollection &theCfg, DiagBitflag valdiag)

  Config::GetMesh(): Use configuration to create the Mesh object; with options set according to the configuration. Config::GetViewScreen() calls this when creating the ViewScreen object.
- GeodesicIntegratorFunc GetGeodesicIntegrator (const ConfigCollection &theCfg)

  Config::GetGeodesicIntegrator(): Returns a pointer to the integrator function to be used as specified in the configuration file.
- std::unique_ptr< GeodesicOutputHandler > GetOutputHandler (const ConfigCollection &theCfg, DiagBitflag alldiags, DiagBitflag valdiag, std::string FirstLineInfo)

  Config::GetOutputHandler(): Creates the GeodesicOutputHandler object with options specified according to the configuration file, for handling of geodesic outputs.

Variables

- constexpr auto Output_Important_Default = OutputLevel::Level_0_WARNING
- constexpr auto Output_Other_Default = OutputLevel::Level_1_PROC

5.1.1 Typedef Documentation

5.1.1.1 ConfigCollection

```
using Config::ConfigCollection = ConfigReader::ConfigCollection
```

5.1.1.2 SettingError

```
using Config::SettingError = std::invalid_argument
```

5.1.2 Function Documentation

5.1.2.1 GetGeodesicIntegrator()

```
GeodesicIntegratorFunc Config::GetGeodesicIntegrator (
            const ConfigCollection & theCfg)
```

Config::GetGeodesicIntegrator(): Returns a pointer to the integrator function to be used as specified in the configuration file.

5.1.2.2 GetMesh()

```
std::unique_ptr< Mesh > Config::GetMesh (
            const ConfigCollection & theCfg,
            DiagBitflag valdiag)
```

Config::GetMesh(): Use configuration to create the Mesh object; with options set according to the configuration. Config::GetViewScreen() calls this when creating the ViewScreen object.

5.1.2.3 GetMetric()

```
std::unique_ptr< Metric > Config::GetMetric (
            const ConfigCollection & theCfg)
```

Config::GetMetric(): Use configuration to create the correct Metric with specified parameters.

5.1.2.4 GetOutputHandler()

```
std::unique_ptr< GeodesicOutputHandler > Config::GetOutputHandler (
            const ConfigCollection & theCfg,
            DiagBitflag alldiags,
            DiagBitflag valdiag,
            std::string FirstLineInfo)
```

Config::GetOutputHandler(): Creates the GeodesicOutputHandler object with options specified according to the configuration file, for handling of geodesic outputs.

5.1.2.5 GetSource()

```
std::unique_ptr< Source > Config::GetSource (
             const ConfigCollection & theCfg,
             const Metric *const theMetric)
```

Config::GetSource(): Use configuration to create the correct Source with specified parameters.

5.1.2.6 GetViewScreen()

```
std::unique_ptr< ViewScreen > Config::GetViewScreen (
             const ConfigCollection & theCfg,
             DiagBitflag valdiag,
             const Metric *const theMetric)
```

Config::GetViewScreen(): Use configuration to create the ViewScreen object; with options set according to the configuration.

5.1.2.7 InitializeDiagnostics()

```
void Config::InitializeDiagnostics (
             const ConfigCollection & theCfg,
             DiagBitflag & alldiags,
             DiagBitflag & valdiag,
             const Metric *const theMetric)
```

Config::InitializeDiagnostics(): Use configuration to set the Diagnostics bitflag appropriately; initialize all DiagnosticOptions for all Diagnostics that are turned on; and set bitflag for diagnostic to be used for coarseness evaluating in Mesh.

5.1.2.8 InitializeScreenOutput()

```
void Config::InitializeScreenOutput (
             const ConfigCollection & theCfg)
```

5.1.2.9 InitializeTerminations()

```
void Config::InitializeTerminations (
             const ConfigCollection & theCfg,
             TermBitflag & allterms,
             const Metric *const theMetric)
```

Config::InitializeTerminations(): Use configuration to set the Termination bitflag appropriately; and initialize all TerminationOptions for all Terminations that are turned on.

5.1.3 Variable Documentation

5.1.3.1 Output_Important_Default

```
auto Config::Output_Important_Default = OutputLevel::Level_0_WARNING  [constexpr]
```

5.1.3.2 Output_Other_Default

```
auto Config::Output_Other_Default = OutputLevel::Level_1_PROC  [constexpr]
```

5.2 ConfigReader Namespace Reference

Classes

- class ConfigCollection
- class ConfigReaderException

Variables

- const long long MaxStreamSize {std::numeric_limits<std::streamsize>::max()}

5.2.1 Variable Documentation

5.2.1.1 MaxStreamSize

```
const long long ConfigReader::MaxStreamSize {std::numeric_limits<std::streamsize>::max()}
```

5.3 Integrators Namespace Reference

Functions

- std::string GetFullIntegratorDescription ()
- real GetAdaptiveStep (Point curpos, OneIndex curvel)
- void IntegrateGeodesicStep_RK4 (Point curpos, OneIndex curvel, Point &nextpos, OneIndex &nextvel, real &stepsize, const Metric ∗theMetric, const Source ∗theSource)
- void IntegrateGeodesicStep_Verlet (Point curpos, OneIndex curvel, Point &nextpos, OneIndex &nextvel, real &stepsize, const Metric ∗theMetric, const Source ∗theSource)

Variables

- constexpr real delta_nodiv0 = 1e-20
- real Derivative_hval {1e-7}
- std::string IntegratorDescription {"RK4"}
- real epsilon {0.03}
- real SmallestPossibleStepsize {1e-12}
- real VerletVelocityTolerance {0.001}

5.3.1 Function Documentation

5.3.1.1 GetAdaptiveStep()

```
real Integrators::GetAdaptiveStep (
            Point curpos,
            OneIndex curvel)
```

5.3.1.2 GetFullIntegratorDescription()

```
std::string Integrators::GetFullIntegratorDescription ()
```

5.3.1.3 IntegrateGeodesicStep_RK4()

```
void Integrators::IntegrateGeodesicStep_RK4 (
            Point curpos,
            OneIndex curvel,
            Point & nextpos,
            OneIndex & nextvel,
            real & stepsize,
            const Metric * theMetric,
            const Source * theSource)
```

5.3.1.4 IntegrateGeodesicStep_Verlet()

```
void Integrators::IntegrateGeodesicStep_Verlet (
            Point curpos,
            OneIndex curvel,
            Point & nextpos,
            OneIndex & nextvel,
            real & stepsize,
            const Metric * theMetric,
            const Source * theSource)
```

5.3.2 Variable Documentation

5.3.2.1 delta_nodiv0

```
real Integrators::delta_nodiv0 = 1e-20 [constexpr]
```

5.3.2.2 Derivative_hval

```
real Integrators::Derivative_hval {1e-7} [inline]
```

5.3.2.3 epsilon

```
real Integrators::epsilon {0.03} [inline]
```

5.3.2.4 IntegratorDescription

```
std::string Integrators::IntegratorDescription {"RK4"} [inline]
```

5.3.2.5 SmallestPossibleStepsize

```
real Integrators::SmallestPossibleStepsize {1e-12}  [inline]
```

5.3.2.6 VerletVelocityTolerance

```
real Integrators::VerletVelocityTolerance {0.001}  [inline]
```

5.4 tk Namespace Reference

Namespaces

- namespace internal

5.5 tk::internal Namespace Reference

5.6 Utilities Namespace Reference

Classes

- class Timer

Functions

- std::string GetTimeStampString ()

  Other functions in Utilities.
- std::vector< std::string > GetDiagNameStrings (DiagBitflag alldiags, DiagBitflag valdiag)
- std::string GetFirstLineInfoString (const Metric ∗theMetric, const Source ∗theSource, DiagBitflag alldiags, DiagBitflag valdiag, TermBitflag allterms, const ViewScreen ∗theView)

5.6.1 Function Documentation

5.6.1.1 GetDiagNameStrings()

```
std::vector< std::string > Utilities::GetDiagNameStrings (
            DiagBitflag alldiags,
            DiagBitflag valdiag)
```

5.6.1.2 GetFirstLineInfoString()

```
std::string Utilities::GetFirstLineInfoString (
            const Metric * theMetric,
            const Source * theSource,
            DiagBitflag alldiags,
            DiagBitflag valdiag,
            TermBitflag allterms,
            const ViewScreen * theView)
```

5.6.1.3 GetTimeStampString()

```
std::string Utilities::GetTimeStampString ()
```

Other functions in Utilities.

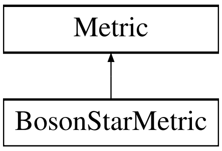
Chapter 6

Class Documentation

6.1 BosonStarMetric Class Reference

```
#include <Metric.h>
```

Inheritance diagram for BosonStarMetric:



Public Member Functions

- BosonStarMetric (bool rLogScale=false)

  BosonStarMetric functions (implementation by Seppe Staelens)
- TwoIndex getMetric_dd (const Point &p) const final
- TwoIndex getMetric_uu (const Point &p) const final
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from Metric

- virtual ∼Metric ()=default
- Metric (bool rlogscale=false)

  Metric (abstract base class) functions.
- virtual ThreeIndex getChristoffel_udd (const Point &p) const
- virtual FourIndex getRiemann_uddd (const Point &p) const
- virtual real getKretschmann (const Point &p) const
- bool getrLogScale () const

Protected Attributes

- tk::spline m_PhiSpline
- tk::spline m_mSpline

Protected Attributes inherited from Metric

- std::vector< int > m_Symmetries {}
- const bool m_rLogScale

6.1.1 Constructor & Destructor Documentation

6.1.1.1 BosonStarMetric()

```
BosonStarMetric::BosonStarMetric (
             bool rLogScale = false)
```

BosonStarMetric functions (implementation by Seppe Staelens)

6.1.2 Member Function Documentation

6.1.2.1 getFullDescriptionStr()

```
std::string BosonStarMetric::getFullDescriptionStr () const  [final], [virtual]
```

Reimplemented from Metric.

6.1.2.2 getMetric_dd()

```
TwoIndex BosonStarMetric::getMetric_dd (
             const Point & p) const  [final], [virtual]
```

Implements Metric.

6.1.2.3 getMetric_uu()

```
TwoIndex BosonStarMetric::getMetric_uu (
             const Point & p) const  [final], [virtual]
```

Implements Metric.

6.1.3 Member Data Documentation

6.1.3.1 m_mSpline

```
tk::spline BosonStarMetric::m_mSpline  [protected]
```

6.1.3.2 m_PhiSpline

```
tk::spline BosonStarMetric::m_PhiSpline [protected]
```

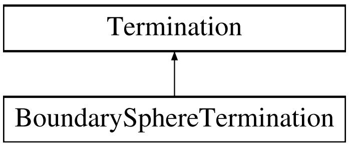
The documentation for this class was generated from the following files:

- FOORT/src/Metric.h
- FOORT/src/Metric.cpp

6.2 BoundarySphereTermination Class Reference

```
#include <Terminations.h>
```

Inheritance diagram for BoundarySphereTermination:



Public Member Functions

- BoundarySphereTermination (Geodesic ∗const theGeodesic)
- Term CheckTermination () final

  BoundarySphereTermination functions.
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from Termination

- Termination ()=delete
- Termination (Geodesic ∗const theGeodesic)
- virtual void Reset ()

  Termination (abstract base class) functions.
- virtual ∼Termination ()=default

Static Public Attributes

- static std::unique_ptr< BoundarySphereTermOptions > TermOptions

Additional Inherited Members

Protected Member Functions inherited from Termination

- bool DecideUpdate (largecounter UpdateNSteps)

Protected Attributes inherited from Termination

- Geodesic ∗const m_OwnerGeodesic
- largecounter m_StepsSinceUpdated {}

6.2.1 Constructor & Destructor Documentation

6.2.1.1 BoundarySphereTermination()

```
BoundarySphereTermination::BoundarySphereTermination (
            Geodesic *const theGeodesic)  [inline]
```

6.2.2 Member Function Documentation

6.2.2.1 CheckTermination()

```
Term BoundarySphereTermination::CheckTermination ()  [final], [virtual]
```

BoundarySphereTermination functions.

Implements Termination.

6.2.2.2 getFullDescriptionStr()

```
std::string BoundarySphereTermination::getFullDescriptionStr () const  [final], [virtual]
```

Implements Termination.

6.2.3 Member Data Documentation

6.2.3.1 TermOptions

```
std::unique_ptr< BoundarySphereTermOptions > BoundarySphereTermination::TermOptions  [static]
```

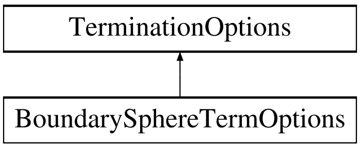
The documentation for this class was generated from the following files:

- FOORT/src/Terminations.h
- FOORT/src/Config.cpp
- FOORT/src/Terminations.cpp

6.3 BoundarySphereTermOptions Struct Reference

```
#include <Terminations.h>
```

Inheritance diagram for BoundarySphereTermOptions:



Public Member Functions

- BoundarySphereTermOptions (real theRadius, bool therLogScale, largecounter Nsteps)

Public Member Functions inherited from TerminationOptions

- TerminationOptions (largecounter Nsteps)
- virtual ∼TerminationOptions ()=default

Public Attributes

- const real SphereRadius
- const bool rLogScale

Public Attributes inherited from TerminationOptions

- const largecounter UpdateEveryNSteps

6.3.1 Constructor & Destructor Documentation

6.3.1.1 BoundarySphereTermOptions()

```
BoundarySphereTermOptions::BoundarySphereTermOptions (
             real theRadius,
             bool therLogScale,
             largecounter Nsteps)  [inline]
```

6.3.2 Member Data Documentation

6.3.2.1 rLogScale

```
const bool BoundarySphereTermOptions::rLogScale
```

6.3.2.2 SphereRadius

```
const real BoundarySphereTermOptions::SphereRadius
```

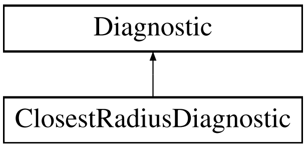
The documentation for this struct was generated from the following file:

- FOORT/src/Terminations.h

6.4 ClosestRadiusDiagnostic Class Reference

```
#include <Diagnostics.h>
```

Inheritance diagram for ClosestRadiusDiagnostic:



Public Member Functions

- ClosestRadiusDiagnostic (Geodesic ∗const theGeodesic)
- void Reset () final

    ClosestRadiusDiagnostic functions.
- void UpdateData () final
- std::string getFullDataStr () const final
- std::vector< real > getFinalDataVal () const final
- real FinalDataValDistance (const std::vector< real > &val1, const std::vector< real > &val2) const final
- std::string getNameStr () const final
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from Diagnostic

- Diagnostic ()=delete
- Diagnostic (Geodesic ∗const theGeodesic)
- virtual ∼Diagnostic ()=default

Static Public Attributes

- static std::unique_ptr< ClosestRadiusOptions > DiagOptions

Private Attributes

- real m_ClosestRadius {-1}

Additional Inherited Members

Protected Member Functions inherited from Diagnostic

- bool DecideUpdate (const UpdateFrequency &myUpdateFrequency)

Protected Attributes inherited from Diagnostic

- Geodesic ∗const m_OwnerGeodesic
- largecounter m_StepsSinceUpdated {}

6.4.1 Constructor & Destructor Documentation

6.4.1.1 ClosestRadiusDiagnostic()

```
ClosestRadiusDiagnostic::ClosestRadiusDiagnostic (
            Geodesic *const theGeodesic)  [inline]
```

6.4.2 Member Function Documentation

6.4.2.1 FinalDataValDistance()

```
real ClosestRadiusDiagnostic::FinalDataValDistance (
            const std::vector< real > & val1,
            const std::vector< real > & val2) const  [final], [virtual]
```

Implements Diagnostic.

6.4.2.2 getFinalDataVal()

```
std::vector< real > ClosestRadiusDiagnostic::getFinalDataVal () const  [final], [virtual]
```

Implements Diagnostic.

6.4.2.3 getFullDataStr()

```
std::string ClosestRadiusDiagnostic::getFullDataStr () const  [final], [virtual]
```

Implements Diagnostic.

6.4.2.4 getFullDescriptionStr()

```
std::string ClosestRadiusDiagnostic::getFullDescriptionStr () const  [final], [virtual]
```

Reimplemented from Diagnostic.

6.4.2.5 getNameStr()

```
std::string ClosestRadiusDiagnostic::getNameStr () const  [final], [virtual]
```

Implements Diagnostic.

6.4.2.6 Reset()

```
void ClosestRadiusDiagnostic::Reset ()  [final], [virtual]
```

ClosestRadiusDiagnostic functions.

Reimplemented from Diagnostic.

6.4.2.7 UpdateData()

```
void ClosestRadiusDiagnostic::UpdateData ()  [final], [virtual]
```

Implements Diagnostic.

6.4.3 Member Data Documentation

6.4.3.1 DiagOptions

```
std::unique_ptr< ClosestRadiusOptions > ClosestRadiusDiagnostic::DiagOptions  [static]
```

6.4.3.2 m_ClosestRadius

```
real ClosestRadiusDiagnostic::m_ClosestRadius {-1}  [private]
```

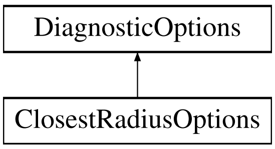
The documentation for this class was generated from the following files:

- FOORT/src/Diagnostics.h
- FOORT/src/Config.cpp
- FOORT/src/Diagnostics.cpp

6.5 ClosestRadiusOptions Struct Reference

```
#include <Diagnostics.h>
```

Inheritance diagram for ClosestRadiusOptions:



Public Member Functions

- ClosestRadiusOptions (bool rlog, UpdateFrequency thefrequency)

Public Member Functions inherited from DiagnosticOptions

- DiagnosticOptions (UpdateFrequency thefrequency)
- virtual ∼DiagnosticOptions ()=default

Public Attributes

- const bool RLogScale

Public Attributes inherited from DiagnosticOptions

- const UpdateFrequency theUpdateFrequency

6.5.1 Constructor & Destructor Documentation

6.5.1.1 ClosestRadiusOptions()

```
ClosestRadiusOptions::ClosestRadiusOptions (
            bool rlog,
            UpdateFrequency thefrequency)  [inline]
```

6.5.2 Member Data Documentation

6.5.2.1 RLogScale

```
const bool ClosestRadiusOptions::RLogScale
```

The documentation for this struct was generated from the following file:

- FOORT/src/Diagnostics.h

6.6 ConfigReader::ConfigCollection Class Reference

```
#include <ConfigReader.h>
```

Classes

- struct ConfigSetting

Public Member Functions

- bool Exists (std::string_view SettingName) const
- int NrSettings () const
- bool IsCollection (std::string_view SettingName) const
- bool IsCollection (int SettingIndex) const
- const ConfigCollection & operator[ ] (std::string_view CollectionName) const
- const ConfigCollection & operator[ ] (int CollectionIndex) const
- template<class OutputType >
  bool LookupValue (std::string_view SettingName, OutputType &theOutput) const
- template<class OutputType >
  bool LookupValue (int SettingIndex, OutputType &theOutput) const
- bool LookupValueInteger (std::string_view SettingName, int &theOutput) const
- bool LookupValueInteger (std::string_view SettingName, long &theOutput) const
- bool LookupValueInteger (std::string_view SettingName, long long &theOutput) const
- bool LookupValueInteger (std::string_view SettingName, unsigned int &theOutput) const
- bool LookupValueInteger (std::string_view SettingName, unsigned long &theOutput) const
- bool LookupValueInteger (std::string_view SettingName, unsigned long long &theOutput) const
- template<class OutputType >
  bool LookupValueInteger (int SettingIndex, OutputType &theOutput) const
- void DisplayCollection (std::ostream &OutputStream, int Indent=0) const
- bool ReadFile (const std::string &FileName)

Private Types

- using ConfigSettingValue

Private Member Functions

- int GetSettingIndex (std::string_view SettingName) const
- void DisplaySetting (std::ostream &OutputStream, int SettingIndex, int Indent) const
- void DisplayTabs (std::ostream &OutputStream, int NrTabs) const
- void ReadCollection (std::ifstream &InputFile)
- void ReadSettingName (std::ifstream &InputFile, std::string &theName)
- void ReadSettingSpecificChar (std::ifstream &InputFile, char theChar) const
- void ReadSettingValue (std::ifstream &InputFile, ConfigSettingValue &theValue)

Private Attributes

- std::vector< ConfigSetting > m_Settings {}

6.6.1 Member Typedef Documentation

6.6.1.1 ConfigSettingValue

```
using ConfigReader::ConfigCollection::ConfigSettingValue  [private]
```

Initial value:

```
 std::variant<bool,
                                                         int, long, long long,
                                                         double,
                                                         std::string,
                                                         std::unique_ptr<ConfigCollection>>
```

6.6.2 Member Function Documentation

6.6.2.1 DisplayCollection()

```
void ConfigCollection::DisplayCollection (
             std::ostream & OutputStream,
             int Indent = 0) const
```

6.6.2.2 DisplaySetting()

```
void ConfigCollection::DisplaySetting (
             std::ostream & OutputStream,
             int SettingIndex,
             int Indent) const  [private]
```

6.6.2.3 DisplayTabs()

```
void ConfigCollection::DisplayTabs (
             std::ostream & OutputStream,
             int NrTabs) const  [private]
```

6.6.2.4 Exists()

```
bool ConfigCollection::Exists (
            std::string_view SettingName) const
```

6.6.2.5 GetSettingIndex()

```
int ConfigCollection::GetSettingIndex (
            std::string_view SettingName) const  [private]
```

6.6.2.6 IsCollection() [1/2]

```
bool ConfigCollection::IsCollection (
            int SettingIndex) const
```

6.6.2.7 IsCollection() [2/2]

```
bool ConfigCollection::IsCollection (
            std::string_view SettingName) const
```

6.6.2.8 LookupValue() [1/2]

```
template<class OutputType >
bool ConfigReader::ConfigCollection::LookupValue (
            int SettingIndex,
            OutputType & theOutput) const
```

6.6.2.9 LookupValue() [2/2]

```
template<class OutputType >
bool ConfigReader::ConfigCollection::LookupValue (
            std::string_view SettingName,
            OutputType & theOutput) const
```

6.6.2.10 LookupValueInteger() [1/7]

```
template<class OutputType >
bool ConfigReader::ConfigCollection::LookupValueInteger (
            int SettingIndex,
            OutputType & theOutput) const
```

6.6.2.11 LookupValueInteger() [2/7]

```
bool ConfigCollection::LookupValueInteger (
            std::string_view SettingName,
            int & theOutput) const
```

6.6.2.12 LookupValueInteger() [3/7]

```
bool ConfigCollection::LookupValueInteger (
            std::string_view SettingName,
            long & theOutput) const
```

6.6.2.13 LookupValueInteger() [4/7]

```
bool ConfigCollection::LookupValueInteger (
            std::string_view SettingName,
            long long & theOutput) const
```

6.6.2.14 LookupValueInteger() [5/7]

```
bool ConfigCollection::LookupValueInteger (
            std::string_view SettingName,
            unsigned int & theOutput) const
```

6.6.2.15 LookupValueInteger() [6/7]

```
bool ConfigCollection::LookupValueInteger (
            std::string_view SettingName,
            unsigned long & theOutput) const
```

6.6.2.16 LookupValueInteger() [7/7]

```
bool ConfigCollection::LookupValueInteger (
            std::string_view SettingName,
            unsigned long long & theOutput) const
```

6.6.2.17 NrSettings()

```
int ConfigCollection::NrSettings () const
```

6.6.2.18 operator[]() [1/2]

```
const ConfigCollection & ConfigCollection::operator[] (
            int CollectionIndex) const
```

6.6.2.19 operator[]() [2/2]

```
const ConfigCollection & ConfigCollection::operator[] (
            std::string_view CollectionName) const
```

6.6.2.20 ReadCollection()

```
void ConfigCollection::ReadCollection (
            std::ifstream & InputFile)  [private]
```

6.6.2.21 ReadFile()

```
bool ConfigCollection::ReadFile (
            const std::string & FileName)
```

6.6.2.22 ReadSettingName()

```
void ConfigCollection::ReadSettingName (
            std::ifstream & InputFile,
            std::string & theName)  [private]
```

6.6.2.23 ReadSettingSpecificChar()

```
void ConfigCollection::ReadSettingSpecificChar (
            std::ifstream & InputFile,
            char theChar) const  [private]
```

6.6.2.24 ReadSettingValue()

```
void ConfigCollection::ReadSettingValue (
            std::ifstream & InputFile,
            ConfigSettingValue & theValue)  [private]
```

6.6.3 Member Data Documentation

6.6.3.1 m_Settings

```
std::vector<ConfigSetting> ConfigReader::ConfigCollection::m_Settings {}  [private]
```

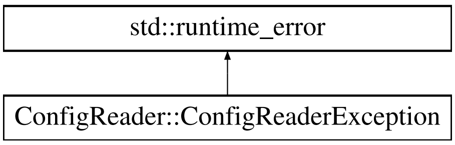
The documentation for this class was generated from the following files:

- FOORT/src/ConfigReader.h
- FOORT/src/ConfigReader.cpp

6.7 ConfigReader::ConfigReaderException Class Reference

```
#include <ConfigReader.h>
```

Inheritance diagram for ConfigReader::ConfigReaderException:



Public Member Functions

- ConfigReaderException (const std::string &error, std::vector< int > settingtrace={})
- std::vector< int > trace () const

Private Attributes

- std::vector< int > m_settingtrace

6.7.1 Constructor & Destructor Documentation

6.7.1.1 ConfigReaderException()

```
ConfigReader::ConfigReaderException::ConfigReaderException (
            const std::string & error,
            std::vector< int > settingtrace = {})  [inline]
```

6.7.2 Member Function Documentation

6.7.2.1 trace()

```
std::vector< int > ConfigReader::ConfigReaderException::trace () const  [inline]
```

6.7.3 Member Data Documentation

6.7.3.1 m_settingtrace

```
std::vector<int> ConfigReader::ConfigReaderException::m_settingtrace  [private]
```

The documentation for this class was generated from the following file:

- FOORT/src/ConfigReader.h

6.8 ConfigReader::ConfigCollection::ConfigSetting Struct Reference

Public Attributes

- std::string SettingName
- ConfigSettingValue SettingValue

6.8.1 Member Data Documentation

6.8.1.1 SettingName

```
std::string ConfigReader::ConfigCollection::ConfigSetting::SettingName
```

6.8.1.2 SettingValue

```
ConfigSettingValue ConfigReader::ConfigCollection::ConfigSetting::SettingValue
```

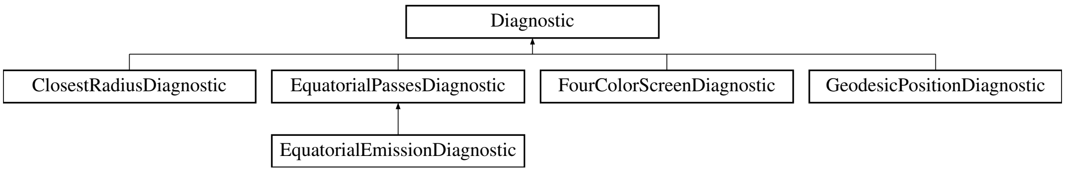
The documentation for this struct was generated from the following file:

- FOORT/src/ConfigReader.h

6.9 Diagnostic Class Reference

```
#include <Diagnostics.h>
```

Inheritance diagram for Diagnostic:


Diagnostic
ClosestRadiusDiagnostic
EquatorialPassesDiagnostic
FourColorScreenDiagnostic
GeodesicPositionDiagnostic
EquatorialEmissionDiagnostic

Public Member Functions

- Diagnostic ()=delete
- Diagnostic (Geodesic ∗const theGeodesic)
- virtual void Reset ()

  Diagnostic (abstract base class) functions.
- virtual ∼Diagnostic ()=default
- virtual void UpdateData ()=0
- virtual std::string getFullDataStr () const =0
- virtual std::vector< real > getFinalDataVal () const =0
- virtual real FinalDataValDistance (const std::vector< real > &val1, const std::vector< real > &val2) const =0
- virtual std::string getNameStr () const =0
- virtual std::string getFullDescriptionStr () const

Protected Member Functions

- bool DecideUpdate (const UpdateFrequency &myUpdateFrequency)

Protected Attributes

- Geodesic ∗const m_OwnerGeodesic
- largecounter m_StepsSinceUpdated {}

6.9.1 Constructor & Destructor Documentation

6.9.1.1 Diagnostic() [1/2]

```
Diagnostic::Diagnostic ()  [delete]
```

6.9.1.2 Diagnostic() [2/2]

```
Diagnostic::Diagnostic (
            Geodesic *const theGeodesic)  [inline]
```

6.9.1.3 ∼Diagnostic()

```
virtual Diagnostic::∼Diagnostic ()  [virtual], [default]
```

6.9.2 Member Function Documentation

6.9.2.1 DecideUpdate()

```
bool Diagnostic::DecideUpdate (
            const UpdateFrequency & myUpdateFrequency)  [protected]
```

6.9.2.2 FinalDataValDistance()

```
virtual real Diagnostic::FinalDataValDistance (
            const std::vector< real > & val1,
            const std::vector< real > & val2) const  [pure virtual]
```

Implemented in ClosestRadiusDiagnostic, EquatorialEmissionDiagnostic, EquatorialPassesDiagnostic, FourColorScreenDiagnostic, and GeodesicPositionDiagnostic.

6.9.2.3 getFinalDataVal()

```
virtual std::vector< real > Diagnostic::getFinalDataVal () const  [pure virtual]
```

Implemented in ClosestRadiusDiagnostic, EquatorialEmissionDiagnostic, EquatorialPassesDiagnostic, FourColorScreenDiagnostic, and GeodesicPositionDiagnostic.

6.9.2.4 getFullDataStr()

```
virtual std::string Diagnostic::getFullDataStr () const  [pure virtual]
```

Implemented in ClosestRadiusDiagnostic, EquatorialEmissionDiagnostic, EquatorialPassesDiagnostic, FourColorScreenDiagnostic, and GeodesicPositionDiagnostic.

6.9.2.5 getFullDescriptionStr()

```
std::string Diagnostic::getFullDescriptionStr () const  [virtual]
```

Reimplemented in ClosestRadiusDiagnostic, EquatorialEmissionDiagnostic, EquatorialPassesDiagnostic, FourColorScreenDiagnostic, and GeodesicPositionDiagnostic.

6.9.2.6 getNameStr()

```
virtual std::string Diagnostic::getNameStr () const  [pure virtual]
```

Implemented in ClosestRadiusDiagnostic, EquatorialEmissionDiagnostic, EquatorialPassesDiagnostic, FourColorScreenDiagnostic, and GeodesicPositionDiagnostic.

6.9.2.7 Reset()

```
void Diagnostic::Reset ()  [virtual]
```

Diagnostic (abstract base class) functions.

Reimplemented in ClosestRadiusDiagnostic, EquatorialEmissionDiagnostic, EquatorialPassesDiagnostic, FourColorScreenDiagnostic, and GeodesicPositionDiagnostic.

6.9.2.8 UpdateData()

```
virtual void Diagnostic::UpdateData ()  [pure virtual]
```

Implemented in ClosestRadiusDiagnostic, EquatorialEmissionDiagnostic, EquatorialPassesDiagnostic, FourColorScreenDiagnostic, and GeodesicPositionDiagnostic.

6.9.3 Member Data Documentation

6.9.3.1 m_OwnerGeodesic

```
Geodesic* const Diagnostic::m_OwnerGeodesic  [protected]
```

6.9.3.2 m_StepsSinceUpdated

```
largecounter Diagnostic::m_StepsSinceUpdated {}  [protected]
```

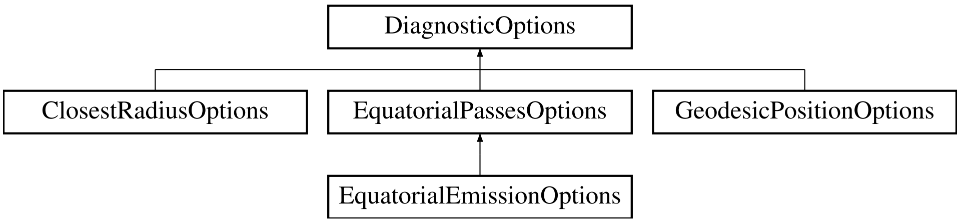
The documentation for this class was generated from the following files:

- FOORT/src/Diagnostics.h
- FOORT/src/Diagnostics.cpp

6.10 DiagnosticOptions Struct Reference

```
#include <Diagnostics.h>
```

Inheritance diagram for DiagnosticOptions:



Public Member Functions

- DiagnosticOptions (UpdateFrequency thefrequency)
- virtual ∼DiagnosticOptions ()=default

Public Attributes

- const UpdateFrequency theUpdateFrequency

6.10.1 Constructor & Destructor Documentation

6.10.1.1 DiagnosticOptions()

```
DiagnosticOptions::DiagnosticOptions (
            UpdateFrequency thefrequency) [inline]
```

6.10.1.2 ∼DiagnosticOptions()

```
virtual DiagnosticOptions::∼DiagnosticOptions () [virtual], [default]
```

6.10.2 Member Data Documentation

6.10.2.1 theUpdateFrequency

```
const UpdateFrequency DiagnosticOptions::theUpdateFrequency
```

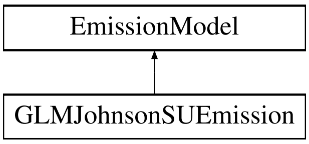
The documentation for this struct was generated from the following file:

- FOORT/src/Diagnostics.h

6.11 EmissionModel Struct Reference

```
#include <DiagnosticsEmission.h>
```

Inheritance diagram for EmissionModel:



Public Member Functions

- virtual ∼EmissionModel ()=default
- virtual real GetEmission (const Point &p) const =0
- virtual std::string getFullDescriptionStr () const

  Emission model functions.

6.11.1 Constructor & Destructor Documentation

6.11.1.1 ∼EmissionModel()

```
virtual EmissionModel::∼EmissionModel ()  [virtual], [default]
```

6.11.2 Member Function Documentation

6.11.2.1 GetEmission()

```
virtual real EmissionModel::GetEmission (
            const Point & p) const  [pure virtual]
```

Implemented in GLMJohnsonSUEmission.

6.11.2.2 getFullDescriptionStr()

```
std::string EmissionModel::getFullDescriptionStr () const  [virtual]
```

Emission model functions.

Reimplemented in GLMJohnsonSUEmission.

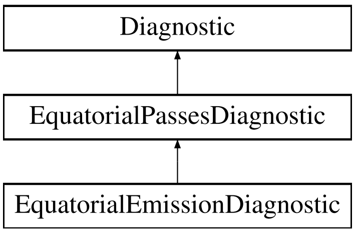
The documentation for this struct was generated from the following files:

- FOORT/src/DiagnosticsEmission.h
- FOORT/src/DiagnosticsEmission.cpp

6.12 EquatorialEmissionDiagnostic Class Reference

```
#include <Diagnostics.h>
```

Inheritance diagram for EquatorialEmissionDiagnostic:



Public Member Functions

- EquatorialEmissionDiagnostic (Geodesic ∗const theGeodesic)
- void Reset () final

  EquatorialEmissionDiagnostic functions.
- void UpdateData () final
- std::string getFullDataStr () const final
- std::vector< real > getFinalDataVal () const final
- real FinalDataValDistance (const std::vector< real > &val1, const std::vector< real > &val2) const final
- std::string getNameStr () const final
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from EquatorialPassesDiagnostic

- EquatorialPassesDiagnostic (Geodesic ∗const theGeodesic)
- void Reset () override

  EquatorialPassesDiagnostic functions.
- void UpdateData () override
- std::string getFullDataStr () const override
- std::vector< real > getFinalDataVal () const override
- real FinalDataValDistance (const std::vector< real > &val1, const std::vector< real > &val2) const override
- std::string getNameStr () const override
- std::string getFullDescriptionStr () const override

Public Member Functions inherited from Diagnostic

- Diagnostic ()=delete
- Diagnostic (Geodesic ∗const theGeodesic)
- virtual ∼Diagnostic ()=default

Static Public Attributes

- static std::unique_ptr< EquatorialEmissionOptions > DiagOptions

Static Public Attributes inherited from EquatorialPassesDiagnostic

- static std::unique_ptr< EquatorialPassesOptions > DiagOptions

Private Attributes

- real m_Intensity {0.0}

Additional Inherited Members

Protected Member Functions inherited from Diagnostic

- bool DecideUpdate (const UpdateFrequency &myUpdateFrequency)

Protected Attributes inherited from EquatorialPassesDiagnostic

- int m_EquatPasses {0}

Protected Attributes inherited from Diagnostic

- Geodesic ∗const m_OwnerGeodesic
- largecounter m_StepsSinceUpdated {}

6.12.1 Constructor & Destructor Documentation

6.12.1.1 EquatorialEmissionDiagnostic()

```
EquatorialEmissionDiagnostic::EquatorialEmissionDiagnostic (
             Geodesic *const theGeodesic)  [inline]
```

6.12.2 Member Function Documentation

6.12.2.1 FinalDataValDistance()

```
real EquatorialEmissionDiagnostic::FinalDataValDistance (
             const std::vector< real > & val1,
             const std::vector< real > & val2) const  [final], [virtual]
```

Implements Diagnostic.

6.12.2.2 getFinalDataVal()

```
std::vector< real > EquatorialEmissionDiagnostic::getFinalDataVal () const  [final], [virtual]
```

Implements Diagnostic.

6.12.2.3 getFullDataStr()

```
std::string EquatorialEmissionDiagnostic::getFullDataStr () const  [final], [virtual]
```

Implements Diagnostic.

6.12.2.4 getFullDescriptionStr()

```
std::string EquatorialEmissionDiagnostic::getFullDescriptionStr () const  [final], [virtual]
```

Reimplemented from Diagnostic.

6.12.2.5 getNameStr()

```
std::string EquatorialEmissionDiagnostic::getNameStr () const  [final], [virtual]
```

Implements Diagnostic.

6.12.2.6 Reset()

```
void EquatorialEmissionDiagnostic::Reset ()  [final], [virtual]
```

EquatorialEmissionDiagnostic functions.

Reimplemented from Diagnostic.

6.12.2.7 UpdateData()

```
void EquatorialEmissionDiagnostic::UpdateData ()  [final], [virtual]
```

Implements Diagnostic.

6.12.3 Member Data Documentation

6.12.3.1 DiagOptions

```
std::unique_ptr< EquatorialEmissionOptions > EquatorialEmissionDiagnostic::DiagOptions  [static]
```

6.12.3.2 m_Intensity

```
real EquatorialEmissionDiagnostic::m_Intensity {0.0}  [private]
```

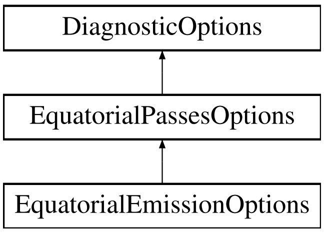
The documentation for this class was generated from the following files:

- FOORT/src/Diagnostics.h
- FOORT/src/Config.cpp
- FOORT/src/Diagnostics.cpp

6.13 EquatorialEmissionOptions Struct Reference

```
#include <Diagnostics.h>
```

Inheritance diagram for EquatorialEmissionOptions:



Public Member Functions

- EquatorialEmissionOptions (real thefudgefactor, int equatupper, std::unique_ptr< EmissionModel > theemission, std::unique_ptr< FluidVelocityModel > thefluidmodel, bool rlog, int theredshiftpower, real thethreshold, UpdateFrequency thefrequency)

Public Member Functions inherited from EquatorialPassesOptions

- EquatorialPassesOptions (real thethreshold, UpdateFrequency thefrequency)

Public Member Functions inherited from DiagnosticOptions

- DiagnosticOptions (UpdateFrequency thefrequency)
- virtual ∼DiagnosticOptions ()=default

Public Attributes

- const real GeometricFudgeFactor
- const int EquatPassUpperBound
- const bool RLogScale
- const int RedShiftPower
- const std::unique_ptr< EmissionModel > TheEmissionModel
- const std::unique_ptr< FluidVelocityModel > TheFluidVelocityModel

Public Attributes inherited from EquatorialPassesOptions

- const real Threshold

Public Attributes inherited from DiagnosticOptions

- const UpdateFrequency theUpdateFrequency

6.13.1 Constructor & Destructor Documentation

6.13.1.1 EquatorialEmissionOptions()

```
EquatorialEmissionOptions::EquatorialEmissionOptions (
             real thefudgefactor,
             int equatupper,
             std::unique_ptr< EmissionModel > theemission,
             std::unique_ptr< FluidVelocityModel > thefluidmodel,
             bool rlog,
             int theredshiftpower,
             real thethreshold,
             UpdateFrequency thefrequency)  [inline]
```

6.13.2 Member Data Documentation

6.13.2.1 EquatPassUpperBound

```
const int EquatorialEmissionOptions::EquatPassUpperBound
```

6.13.2.2 GeometricFudgeFactor

```
const real EquatorialEmissionOptions::GeometricFudgeFactor
```

6.13.2.3 RedShiftPower

```
const int EquatorialEmissionOptions::RedShiftPower
```

6.13.2.4 RLogScale

```
const bool EquatorialEmissionOptions::RLogScale
```

6.13.2.5 TheEmissionModel

```
const std::unique_ptr<EmissionModel> EquatorialEmissionOptions::TheEmissionModel
```

6.13.2.6 TheFluidVelocityModel

```
const std::unique_ptr<FluidVelocityModel> EquatorialEmissionOptions::TheFluidVelocityModel
```

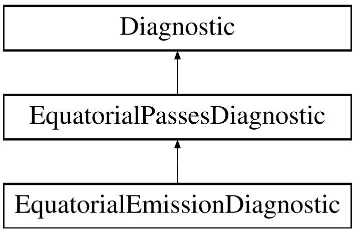
The documentation for this struct was generated from the following file:

- FOORT/src/Diagnostics.h

6.14 EquatorialPassesDiagnostic Class Reference

```
#include <Diagnostics.h>
```

Inheritance diagram for EquatorialPassesDiagnostic:



Public Member Functions

- EquatorialPassesDiagnostic (Geodesic ∗const theGeodesic)
- void Reset () override

  EquatorialPassesDiagnostic functions.
- void UpdateData () override
- std::string getFullDataStr () const override
- std::vector< real > getFinalDataVal () const override
- real FinalDataValDistance (const std::vector< real > &val1, const std::vector< real > &val2) const override
- std::string getNameStr () const override
- std::string getFullDescriptionStr () const override

Public Member Functions inherited from Diagnostic

- Diagnostic ()=delete
- Diagnostic (Geodesic ∗const theGeodesic)
- virtual ∼Diagnostic ()=default

Static Public Attributes

- static std::unique_ptr< EquatorialPassesOptions > DiagOptions

Protected Attributes

- int m_EquatPasses {0}

Protected Attributes inherited from Diagnostic

- Geodesic ∗const m_OwnerGeodesic
- largecounter m_StepsSinceUpdated {}

Private Attributes

- real m_PrevTheta {-1}

Additional Inherited Members

Protected Member Functions inherited from Diagnostic

- bool DecideUpdate (const UpdateFrequency &myUpdateFrequency)

6.14.1 Constructor & Destructor Documentation

6.14.1.1 EquatorialPassesDiagnostic()

```
EquatorialPassesDiagnostic::EquatorialPassesDiagnostic (
            Geodesic *const theGeodesic)  [inline]
```

6.14.2 Member Function Documentation

6.14.2.1 FinalDataValDistance()

```
real EquatorialPassesDiagnostic::FinalDataValDistance (
            const std::vector< real > & val1,
            const std::vector< real > & val2) const  [override], [virtual]
```

Implements Diagnostic.

6.14.2.2 getFinalDataVal()

```
std::vector< real > EquatorialPassesDiagnostic::getFinalDataVal () const  [override], [virtual]
```

Implements Diagnostic.

6.14.2.3 getFullDataStr()

```
std::string EquatorialPassesDiagnostic::getFullDataStr () const  [override], [virtual]
```

Implements Diagnostic.

6.14.2.4 getFullDescriptionStr()

```
std::string EquatorialPassesDiagnostic::getFullDescriptionStr () const  [override], [virtual]
```

Reimplemented from Diagnostic.

6.14.2.5 getNameStr()

```
std::string EquatorialPassesDiagnostic::getNameStr () const  [override], [virtual]
```

Implements Diagnostic.

6.14.2.6 Reset()

```
void EquatorialPassesDiagnostic::Reset ()  [override], [virtual]
```

EquatorialPassesDiagnostic functions.

Reimplemented from Diagnostic.

6.14.2.7 UpdateData()

```
void EquatorialPassesDiagnostic::UpdateData ()  [override], [virtual]
```

Implements Diagnostic.

6.14.3 Member Data Documentation

6.14.3.1 DiagOptions

```
std::unique_ptr< EquatorialPassesOptions > EquatorialPassesDiagnostic::DiagOptions  [static]
```

6.14.3.2 m_EquatPasses

```
int EquatorialPassesDiagnostic::m_EquatPasses {0}  [protected]
```

6.14.3.3 m_PrevTheta

```
real EquatorialPassesDiagnostic::m_PrevTheta {-1}  [private]
```

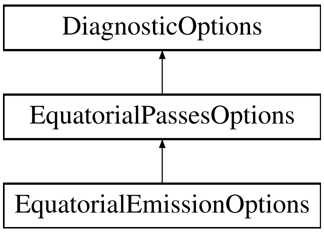
The documentation for this class was generated from the following files:

- FOORT/src/Diagnostics.h
- FOORT/src/Config.cpp
- FOORT/src/Diagnostics.cpp

6.15 EquatorialPassesOptions Struct Reference

```
#include <Diagnostics.h>
```

Inheritance diagram for EquatorialPassesOptions:



Public Member Functions

- EquatorialPassesOptions (real thethreshold, UpdateFrequency thefrequency)

Public Member Functions inherited from DiagnosticOptions

- DiagnosticOptions (UpdateFrequency thefrequency)
- virtual ∼DiagnosticOptions ()=default

Public Attributes

- const real Threshold

Public Attributes inherited from DiagnosticOptions

- const UpdateFrequency theUpdateFrequency

6.15.1 Constructor & Destructor Documentation

6.15.1.1 EquatorialPassesOptions()

```
EquatorialPassesOptions::EquatorialPassesOptions (
            real thethreshold,
            UpdateFrequency thefrequency)  [inline]
```

6.15.2 Member Data Documentation

6.15.2.1 Threshold

```
const real EquatorialPassesOptions::Threshold
```

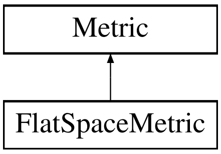
The documentation for this struct was generated from the following file:

- FOORT/src/Diagnostics.h

6.16 FlatSpaceMetric Class Reference

```
#include <Metric.h>
```

Inheritance diagram for FlatSpaceMetric:



Public Member Functions

- FlatSpaceMetric (bool rlogscale=false)

  FlatSpaceMetric functions.
- TwoIndex getMetric_dd (const Point &p) const final
- TwoIndex getMetric_uu (const Point &p) const final
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from Metric

- virtual ~Metric ()=default
- Metric (bool rlogscale=false)

    Metric (abstract base class) functions.
- virtual ThreeIndex getChristoffel_udd (const Point &p) const
- virtual FourIndex getRiemann_uddd (const Point &p) const
- virtual real getKretschmann (const Point &p) const
- bool getrLogScale () const

Additional Inherited Members

Protected Attributes inherited from Metric

- std::vector< int > m_Symmetries {}
- const bool m_rLogScale

6.16.1 Constructor & Destructor Documentation

6.16.1.1 FlatSpaceMetric()

```
FlatSpaceMetric::FlatSpaceMetric (
             bool rlogscale = false)
```

FlatSpaceMetric functions.

6.16.2 Member Function Documentation

6.16.2.1 getFullDescriptionStr()

```
std::string FlatSpaceMetric::getFullDescriptionStr () const  [final], [virtual]
```

Reimplemented from Metric.

6.16.2.2 getMetric_dd()

```
TwoIndex FlatSpaceMetric::getMetric_dd (
             const Point & p) const  [final], [virtual]
```

Implements Metric.

6.16.2.3 getMetric_uu()

```
TwoIndex FlatSpaceMetric::getMetric_uu (
             const Point & p) const  [final], [virtual]
```

Implements Metric.

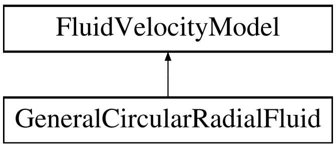
The documentation for this class was generated from the following files:

- FOORT/src/Metric.h
- FOORT/src/Metric.cpp

6.17 FluidVelocityModel Struct Reference

```
#include <DiagnosticsEmission.h>
```

Inheritance diagram for FluidVelocityModel:



Public Member Functions

- FluidVelocityModel (const Metric ∗const theMetric)
- virtual ∼FluidVelocityModel ()=default
- virtual OneIndex GetFourVelocityd (const Point &p) const =0
- virtual std::string getFullDescriptionStr () const

  FluidVelocityModel functions.

Protected Attributes

- const Metric ∗const m_theMetric

6.17.1 Constructor & Destructor Documentation

6.17.1.1 FluidVelocityModel()

```
FluidVelocityModel::FluidVelocityModel (
              const Metric *const theMetric)  [inline]
```

6.17.1.2 ∼FluidVelocityModel()

```
virtual FluidVelocityModel::~FluidVelocityModel ()  [virtual], [default]
```

6.17.2 Member Function Documentation

6.17.2.1 GetFourVelocityd()

```
virtual OneIndex FluidVelocityModel::GetFourVelocityd (
              const Point & p) const  [pure virtual]
```

Implemented in GeneralCircularRadialFluid.

6.17.2.2 getFullDescriptionStr()

```
std::string FluidVelocityModel::getFullDescriptionStr () const  [virtual]
```

FluidVelocityModel functions.

Reimplemented in GeneralCircularRadialFluid.

6.17.3 Member Data Documentation

6.17.3.1 m_theMetric

```
const Metric* const FluidVelocityModel::m_theMetric  [protected]
```

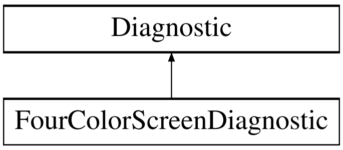
The documentation for this struct was generated from the following files:

- FOORT/src/DiagnosticsEmission.h
- FOORT/src/DiagnosticsEmission.cpp

6.18 FourColorScreenDiagnostic Class Reference

```
#include <Diagnostics.h>
```

Inheritance diagram for FourColorScreenDiagnostic:



Public Member Functions

- FourColorScreenDiagnostic (Geodesic ∗const theGeodesic)
- void Reset () final

  FourColorScreen functions.
- void UpdateData () override
- std::string getFullDataStr () const final
- std::vector< real > getFinalDataVal () const final
- real FinalDataValDistance (const std::vector< real > &val1, const std::vector< real > &val2) const final
- std::string getNameStr () const final
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from Diagnostic

- Diagnostic ()=delete
- Diagnostic (Geodesic ∗const theGeodesic)
- virtual ∼Diagnostic ()=default

Private Attributes

- int m_quadrant {0}

Additional Inherited Members

Protected Member Functions inherited from Diagnostic

- bool DecideUpdate (const UpdateFrequency &myUpdateFrequency)

Protected Attributes inherited from Diagnostic

- Geodesic ∗const m_OwnerGeodesic
- largecounter m_StepsSinceUpdated {}

6.18.1 Constructor & Destructor Documentation

6.18.1.1 FourColorScreenDiagnostic()

```
FourColorScreenDiagnostic::FourColorScreenDiagnostic (
            Geodesic *const theGeodesic)  [inline]
```

6.18.2 Member Function Documentation

6.18.2.1 FinalDataValDistance()

```
real FourColorScreenDiagnostic::FinalDataValDistance (
            const std::vector< real > & val1,
            const std::vector< real > & val2) const  [final], [virtual]
```

Implements Diagnostic.

6.18.2.2 getFinalDataVal()

```
std::vector< real > FourColorScreenDiagnostic::getFinalDataVal () const  [final], [virtual]
```

Implements Diagnostic.

6.18.2.3 getFullDataStr()

```
std::string FourColorScreenDiagnostic::getFullDataStr () const  [final], [virtual]
```

Implements Diagnostic.

6.18.2.4 getFullDescriptionStr()

```
std::string FourColorScreenDiagnostic::getFullDescriptionStr () const  [final], [virtual]
```

Reimplemented from Diagnostic.

6.18.2.5 getNameStr()

```
std::string FourColorScreenDiagnostic::getNameStr () const  [final], [virtual]
```

Implements Diagnostic.

6.18.2.6 Reset()

```
void FourColorScreenDiagnostic::Reset ()  [final], [virtual]
```

FourColorScreen functions.

Reimplemented from Diagnostic.

6.18.2.7 UpdateData()

```
void FourColorScreenDiagnostic::UpdateData ()  [override], [virtual]
```

Implements Diagnostic.

6.18.3 Member Data Documentation

6.18.3.1 m_quadrant

```
int FourColorScreenDiagnostic::m_quadrant {0}  [private]
```

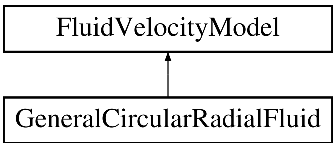
The documentation for this class was generated from the following files:

- FOORT/src/Diagnostics.h
- FOORT/src/Diagnostics.cpp

6.19 GeneralCircularRadialFluid Struct Reference

```
#include <DiagnosticsEmission.h>
```

Inheritance diagram for GeneralCircularRadialFluid:



Public Member Functions

- GeneralCircularRadialFluid (real subKeplerParam, real betar, real betaphi, const Metric ∗const theMetric)
- OneIndex GetFourVelocityd (const Point &p) const final
- std::string getFullDescriptionStr () const final

  FluidVelocityModel functions.

Public Member Functions inherited from FluidVelocityModel

- FluidVelocityModel (const Metric ∗const theMetric)
- virtual ∼FluidVelocityModel ()=default

Private Member Functions

- OneIndex GetCircularVelocityd (const Point &p, bool subKeplerianOn=true) const
- OneIndex GetInsideISCOCircularVelocityd (const Point &p) const
- OneIndex GetRadialVelocityd (const Point &p) const
- void FindISCO ()
- TwoIndex GetChristrRaisedDer (real r) const

Private Attributes

- const real m_subKeplerParam
- const real m_betaR
- const real m_betaPhi
- bool m_ISCOexists {false}
- real m_ISCOr {-1.0}
- real m_ISCOpt {}
- real m_ISCOpphi {}

Additional Inherited Members

Protected Attributes inherited from FluidVelocityModel

- const Metric ∗const m_theMetric

6.19.1 Constructor & Destructor Documentation

6.19.1.1 GeneralCircularRadialFluid()

```
GeneralCircularRadialFluid::GeneralCircularRadialFluid (
            real subKeplerParam,
            real betar,
            real betaphi,
            const Metric *const theMetric)  [inline]
```

6.19.2 Member Function Documentation

6.19.2.1 FindISCO()

```
void GeneralCircularRadialFluid::FindISCO ()  [private]
```

6.19.2.2 GetChristrRaisedDer()

```
TwoIndex GeneralCircularRadialFluid::GetChristrRaisedDer (
             real r) const  [private]
```

6.19.2.3 GetCircularVelocityd()

```
OneIndex GeneralCircularRadialFluid::GetCircularVelocityd (
             const Point & p,
             bool subKeplerianOn = true) const  [private]
```

6.19.2.4 GetFourVelocityd()

```
OneIndex GeneralCircularRadialFluid::GetFourVelocityd (
             const Point & p) const  [final], [virtual]
```

Implements FluidVelocityModel.

6.19.2.5 getFullDescriptionStr()

```
std::string GeneralCircularRadialFluid::getFullDescriptionStr () const  [final], [virtual]
```

FluidVelocityModel functions.

Reimplemented from FluidVelocityModel.

6.19.2.6 GetInsideISCOCircularVelocityd()

```
OneIndex GeneralCircularRadialFluid::GetInsideISCOCircularVelocityd (
             const Point & p) const  [private]
```

6.19.2.7 GetRadialVelocityd()

```
OneIndex GeneralCircularRadialFluid::GetRadialVelocityd (
             const Point & p) const  [private]
```

6.19.3 Member Data Documentation

6.19.3.1 m_betaPhi

```
const real GeneralCircularRadialFluid::m_betaPhi [private]
```

6.19.3.2 m_betaR

```
const real GeneralCircularRadialFluid::m_betaR [private]
```

6.19.3.3 m_ISCOexists

```
bool GeneralCircularRadialFluid::m_ISCOexists {false} [private]
```

6.19.3.4 m_ISCOpphi

```
real GeneralCircularRadialFluid::m_ISCOpphi {} [private]
```

6.19.3.5 m_ISCOpt

```
real GeneralCircularRadialFluid::m_ISCOpt {} [private]
```

6.19.3.6 m_ISCOr

```
real GeneralCircularRadialFluid::m_ISCOr {-1.0} [private]
```

6.19.3.7 m_subKeplerParam

```
const real GeneralCircularRadialFluid::m_subKeplerParam [private]
```

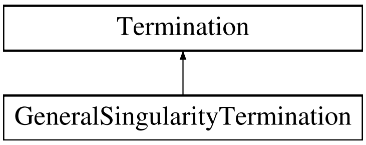
The documentation for this struct was generated from the following files:

- FOORT/src/DiagnosticsEmission.h
- FOORT/src/DiagnosticsEmission.cpp

6.20 GeneralSingularityTermination Class Reference

```
#include <Terminations.h>
```

Inheritance diagram for GeneralSingularityTermination:



Public Member Functions

- GeneralSingularityTermination (Geodesic ∗const theGeodesic)
- Term CheckTermination () final

  GeneralSingularityTermination functions.
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from Termination

- Termination ()=delete
- Termination (Geodesic ∗const theGeodesic)
- virtual void Reset ()

  Termination (abstract base class) functions.
- virtual ∼Termination ()=default

Static Public Attributes

- static std::unique_ptr< GeneralSingularityTermOptions > TermOptions

Private Member Functions

- std::string SingularityToString (int singnr) const

Additional Inherited Members

Protected Member Functions inherited from Termination

- bool DecideUpdate (largecounter UpdateNSteps)

Protected Attributes inherited from Termination

- Geodesic ∗const m_OwnerGeodesic
- largecounter m_StepsSinceUpdated {}

6.20.1 Constructor & Destructor Documentation

6.20.1.1 GeneralSingularityTermination()

```
GeneralSingularityTermination::GeneralSingularityTermination (
            Geodesic *const theGeodesic)  [inline]
```

6.20.2 Member Function Documentation

6.20.2.1 CheckTermination()

```
Term GeneralSingularityTermination::CheckTermination ()  [final], [virtual]
```

GeneralSingularityTermination functions.

Implements Termination.

6.20.2.2 getFullDescriptionStr()

```
std::string GeneralSingularityTermination::getFullDescriptionStr () const  [final], [virtual]
```

Implements Termination.

6.20.2.3 SingularityToString()

```
std::string GeneralSingularityTermination::SingularityToString (
             int singnr) const  [private]
```

6.20.3 Member Data Documentation

6.20.3.1 TermOptions

```
std::unique_ptr< GeneralSingularityTermOptions > GeneralSingularityTermination::TermOptions
[static]
```

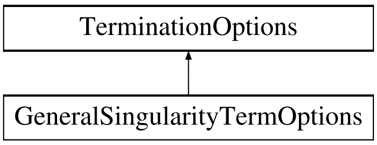
The documentation for this class was generated from the following files:

- FOORT/src/Terminations.h
- FOORT/src/Config.cpp
- FOORT/src/Terminations.cpp

6.21 GeneralSingularityTermOptions Struct Reference

```
#include <Terminations.h>
```

Inheritance diagram for GeneralSingularityTermOptions:



Public Member Functions

- GeneralSingularityTermOptions (std::vector< Singularity > sings, real eps, bool consoleoutputon, bool therlogscale, largecounter Nsteps)

Public Member Functions inherited from TerminationOptions

- TerminationOptions (largecounter Nsteps)
- virtual ∼TerminationOptions ()=default

Public Attributes

- const std::vector< Singularity > Singularities
- const real Epsilon
- const bool OutputToConsole
- const bool rLogScale

Public Attributes inherited from TerminationOptions

- const largecounter UpdateEveryNSteps

6.21.1 Constructor & Destructor Documentation

6.21.1.1 GeneralSingularityTermOptions()

```
GeneralSingularityTermOptions::GeneralSingularityTermOptions (
            std::vector< Singularity > sings,
            real eps,
            bool consoleoutputon,
            bool therlogscale,
            largecounter Nsteps)  [inline]
```

6.21.2 Member Data Documentation

6.21.2.1 Epsilon

```
const real GeneralSingularityTermOptions::Epsilon
```

6.21.2.2 OutputToConsole

```
const bool GeneralSingularityTermOptions::OutputToConsole
```

6.21.2.3 rLogScale

```
const bool GeneralSingularityTermOptions::rLogScale
```

6.21.2.4 Singularities

```
const std::vector<Singularity> GeneralSingularityTermOptions::Singularities
```

The documentation for this struct was generated from the following file:

- FOORT/src/Terminations.h

6.22 Geodesic Class Reference

```
#include <Geodesic.h>
```

Public Member Functions

- Geodesic ()=delete
- Geodesic (const Geodesic &)=delete
- Geodesic & operator= (const Geodesic &)=delete
- Geodesic (const Metric ∗const theMetric, const Source ∗const theSource, DiagBitflag diagbit, DiagBitflag valdiagbit, TermBitflag termbit, GeodesicIntegratorFunc theIntegrator)
- void Reset (ScreenIndex scrindex, Point initpos, OneIndex initvel)

    Geodesic (and descendant classes) functions.
- Term Update ()
- Term getTermCondition () const
- Point getCurrentPos () const
- OneIndex getCurrentVel () const
- real getCurrentLambda () const
- ScreenIndex getScreenIndex () const
- std::vector< std::string > getAllOutputStr () const
- std::vector< real > getDiagnosticFinalValue () const

Private Attributes

- Term m_TermCond {Term::Uninitialized}
- Point m_CurrentPos {}
- OneIndex m_CurrentVel {}
- real m_curLambda {0.0}
- ScreenIndex m_ScreenIndex {}
- const Metric ∗const m_theMetric
- const Source ∗const m_theSource
- const DiagnosticUniqueVector m_AllDiagnostics
- const TerminationUniqueVector m_AllTerminations
- const GeodesicIntegratorFunc m_theIntegrator

6.22.1 Constructor & Destructor Documentation

6.22.1.1 Geodesic() [1/3]

```
Geodesic::Geodesic ( ) [delete]
```

6.22.1.2 Geodesic() [2/3]

```
Geodesic::Geodesic (
            const Geodesic & ) [delete]
```

6.22.1.3 Geodesic() [3/3]

```
Geodesic::Geodesic (
             const Metric *const theMetric,
             const Source *const theSource,
             DiagBitflag diagbit,
             DiagBitflag valdiagbit,
             TermBitflag termbit,
             GeodesicIntegratorFunc theIntegrator)  [inline]
```

6.22.2 Member Function Documentation

6.22.2.1 getAllOutputStr()

```
std::vector< std::string > Geodesic::getAllOutputStr () const
```

6.22.2.2 getCurrentLambda()

```
real Geodesic::getCurrentLambda () const
```

6.22.2.3 getCurrentPos()

```
Point Geodesic::getCurrentPos () const
```

6.22.2.4 getCurrentVel()

```
OneIndex Geodesic::getCurrentVel () const
```

6.22.2.5 getDiagnosticFinalValue()

```
std::vector< real > Geodesic::getDiagnosticFinalValue () const
```

6.22.2.6 getScreenIndex()

```
ScreenIndex Geodesic::getScreenIndex () const
```

6.22.2.7 getTermCondition()

```
Term Geodesic::getTermCondition () const
```

6.22.2.8 operator=()

```
Geodesic & Geodesic::operator= (
             const Geodesic & )  [delete]
```

6.22.2.9 Reset()

```
void Geodesic::Reset (
            ScreenIndex scrindex,
            Point initpos,
            OneIndex initvel)
```

Geodesic (and descendant classes) functions.

6.22.2.10 Update()

```
Term Geodesic::Update ()
```

6.22.3 Member Data Documentation

6.22.3.1 m_AllDiagnostics

```
const DiagnosticUniqueVector Geodesic::m_AllDiagnostics  [private]
```

6.22.3.2 m_AllTerminations

```
const TerminationUniqueVector Geodesic::m_AllTerminations  [private]
```

6.22.3.3 m_curLambda

```
real Geodesic::m_curLambda {0.0}  [private]
```

6.22.3.4 m_CurrentPos

```
Point Geodesic::m_CurrentPos {}  [private]
```

6.22.3.5 m_CurrentVel

```
OneIndex Geodesic::m_CurrentVel {}  [private]
```

6.22.3.6 m_ScreenIndex

```
ScreenIndex Geodesic::m_ScreenIndex {}  [private]
```

6.22.3.7 m_TermCond

```
Term Geodesic::m_TermCond {Term::Uninitialized}  [private]
```

6.22.3.8 m_theIntegrator

```
const GeodesicIntegratorFunc Geodesic::m_theIntegrator [private]
```

6.22.3.9 m_theMetric

```
const Metric* const Geodesic::m_theMetric [private]
```

6.22.3.10 m_theSource

```
const Source* const Geodesic::m_theSource [private]
```

The documentation for this class was generated from the following files:

- FOORT/src/Geodesic.h
- FOORT/src/Geodesic.cpp

6.23 GeodesicOutputHandler Class Reference

```
#include <InputOutput.h>
```

Public Member Functions

- GeodesicOutputHandler ()=delete
- GeodesicOutputHandler (std::string FilePrefix, std::string TimeStamp, std::string FileExtension, std::vector< std::string > DiagNames, largecounter nroutputstocache=LARGECOUNTER_MAX - 1, largecounter geodperfile=LARGECOUNTER_MAX, std::string firstlineinfo="")

  GeodesicOutputHandler functions.
- void PrepareForOutput (largecounter nrOutputToCome)
- void NewGeodesicOutput (largecounter index, std::vector< std::string > theOutput)
- void OutputFinished ()
- std::string getFullDescriptionStr () const

Private Member Functions

- void WriteCachedOutputToFile ()
- void OpenForFirstTime (const std::string &filename)
- std::string GetFileName (int diagnr, unsigned short filenr) const

Private Attributes

- const std::string m_FilePrefix
- const std::string m_TimeStamp
- const std::string m_FileExtension
- const std::vector< std::string > m_DiagNames
- const bool m_PrintFirstLineInfo
- const std::string m_FirstLineInfoString
- const largecounter m_nrOutputsToCache {}
- const largecounter m_nrGeodesicsPerFile {}
- bool m_WriteToConsole {false}
- largecounter m_PrevCached {0}
- largecounter m_CurrentGeodesicsInFile {0}
- unsigned short m_CurrentFullFiles {0}
- std::vector< std::vector< std::string > > m_AllCachedData {}

6.23.1 Constructor & Destructor Documentation

6.23.1.1 GeodesicOutputHandler() [1/2]

```
GeodesicOutputHandler::GeodesicOutputHandler ()  [delete]
```

6.23.1.2 GeodesicOutputHandler() [2/2]

```
GeodesicOutputHandler::GeodesicOutputHandler (
             std::string FilePrefix,
             std::string TimeStamp,
             std::string FileExtension,
             std::vector< std::string > DiagNames,
             largecounter nroutputstocache = LARGECOUNTER_MAX - 1,
             largecounter geodperfile = LARGECOUNTER_MAX,
             std::string firstlineinfo = "")
```

GeodesicOutputHandler functions.

6.23.2 Member Function Documentation

6.23.2.1 GetFileName()

```
std::string GeodesicOutputHandler::GetFileName (
             int diagnr,
             unsigned short filenr) const  [private]
```

6.23.2.2 getFullDescriptionStr()

```
std::string GeodesicOutputHandler::getFullDescriptionStr () const
```

6.23.2.3 NewGeodesicOutput()

```
void GeodesicOutputHandler::NewGeodesicOutput (
            largecounter index,
            std::vector< std::string > theOutput)
```

6.23.2.4 OpenForFirstTime()

```
void GeodesicOutputHandler::OpenForFirstTime (
            const std::string & filename)  [private]
```

6.23.2.5 OutputFinished()

```
void GeodesicOutputHandler::OutputFinished ()
```

6.23.2.6 PrepareForOutput()

```
void GeodesicOutputHandler::PrepareForOutput (
            largecounter nrOutputToCome)
```

6.23.2.7 WriteCachedOutputToFile()

```
void GeodesicOutputHandler::WriteCachedOutputToFile ()  [private]
```

6.23.3 Member Data Documentation

6.23.3.1 m_AllCachedData

```
std::vector<std::vector<std::string> > GeodesicOutputHandler::m_AllCachedData {}  [private]
```

6.23.3.2 m_CurrentFullFiles

```
unsigned short GeodesicOutputHandler::m_CurrentFullFiles {0}  [private]
```

6.23.3.3 m_CurrentGeodesicsInFile

```
largecounter GeodesicOutputHandler::m_CurrentGeodesicsInFile {0}  [private]
```

6.23.3.4 m_DiagNames

```
const std::vector<std::string> GeodesicOutputHandler::m_DiagNames  [private]
```

6.23.3.5 m_FileExtension

```
const std::string GeodesicOutputHandler::m_FileExtension  [private]
```

6.23.3.6 m_FilePrefix

```
const std::string GeodesicOutputHandler::m_FilePrefix  [private]
```

6.23.3.7 m_FirstLineInfoString

```
const std::string GeodesicOutputHandler::m_FirstLineInfoString  [private]
```

6.23.3.8 m_nrGeodesicsPerFile

```
const largecounter GeodesicOutputHandler::m_nrGeodesicsPerFile {}  [private]
```

6.23.3.9 m_nrOutputsToCache

```
const largecounter GeodesicOutputHandler::m_nrOutputsToCache {}  [private]
```

6.23.3.10 m_PrevCached

```
largecounter GeodesicOutputHandler::m_PrevCached {0}  [private]
```

6.23.3.11 m_PrintFirstLineInfo

```
const bool GeodesicOutputHandler::m_PrintFirstLineInfo  [private]
```

6.23.3.12 m_TimeStamp

```
const std::string GeodesicOutputHandler::m_TimeStamp  [private]
```

6.23.3.13 m_WriteToConsole

```
bool GeodesicOutputHandler::m_WriteToConsole {false}  [private]
```

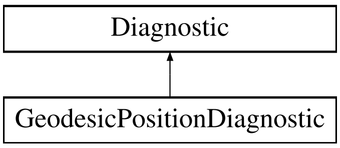
The documentation for this class was generated from the following files:

- FOORT/src/InputOutput.h
- FOORT/src/InputOutput.cpp

6.24 GeodesicPositionDiagnostic Class Reference

```
#include <Diagnostics.h>
```

Inheritance diagram for GeodesicPositionDiagnostic:



Public Member Functions

- GeodesicPositionDiagnostic (Geodesic *const theGeodesic)
- void Reset () final

  GeodesicPositionDiagnostic functions.
- void UpdateData () final
- std::string getFullDataStr () const final
- std::vector< real > getFinalDataVal () const final
- real FinalDataValDistance (const std::vector< real > &val1, const std::vector< real > &val2) const final
- std::string getNameStr () const final
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from Diagnostic

- Diagnostic ()=delete
- Diagnostic (Geodesic *const theGeodesic)
- virtual ~Diagnostic ()=default

Static Public Attributes

- static std::unique_ptr< GeodesicPositionOptions > DiagOptions

Private Attributes

- std::vector< Point > m_AllSavedPoints {}

Additional Inherited Members

Protected Member Functions inherited from Diagnostic

- bool DecideUpdate (const UpdateFrequency &myUpdateFrequency)

Protected Attributes inherited from Diagnostic

- Geodesic *const m_OwnerGeodesic
- largecounter m_StepsSinceUpdated {}

6.24.1 Constructor & Destructor Documentation

6.24.1.1 GeodesicPositionDiagnostic()

```
GeodesicPositionDiagnostic::GeodesicPositionDiagnostic (
             Geodesic *const theGeodesic)  [inline]
```

6.24.2 Member Function Documentation

6.24.2.1 FinalDataValDistance()

```
real GeodesicPositionDiagnostic::FinalDataValDistance (
             const std::vector< real > & val1,
             const std::vector< real > & val2) const  [final], [virtual]
```

Implements Diagnostic.

6.24.2.2 getFinalDataVal()

```
std::vector< real > GeodesicPositionDiagnostic::getFinalDataVal () const  [final], [virtual]
```

Implements Diagnostic.

6.24.2.3 getFullDataStr()

```
std::string GeodesicPositionDiagnostic::getFullDataStr () const  [final], [virtual]
```

Implements Diagnostic.

6.24.2.4 getFullDescriptionStr()

```
std::string GeodesicPositionDiagnostic::getFullDescriptionStr () const  [final], [virtual]
```

Reimplemented from Diagnostic.

6.24.2.5 getNameStr()

```
std::string GeodesicPositionDiagnostic::getNameStr () const  [final], [virtual]
```

Implements Diagnostic.

6.24.2.6 Reset()

```
void GeodesicPositionDiagnostic::Reset ()  [final], [virtual]
```

GeodesicPositionDiagnostic functions.

Reimplemented from Diagnostic.

6.24.2.7 UpdateData()

```
void GeodesicPositionDiagnostic::UpdateData ()  [final], [virtual]
```

Implements Diagnostic.

6.24.3 Member Data Documentation

6.24.3.1 DiagOptions

```
std::unique_ptr< GeodesicPositionOptions > GeodesicPositionDiagnostic::DiagOptions  [static]
```

6.24.3.2 m_AllSavedPoints

```
std::vector<Point> GeodesicPositionDiagnostic::m_AllSavedPoints {}  [private]
```

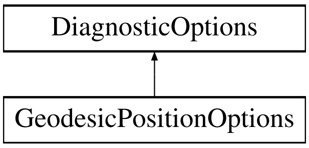
The documentation for this class was generated from the following files:

- FOORT/src/Diagnostics.h
- FOORT/src/Config.cpp
- FOORT/src/Diagnostics.cpp

6.25 GeodesicPositionOptions Struct Reference

```
#include <Diagnostics.h>
```

Inheritance diagram for GeodesicPositionOptions:



Public Member Functions

- GeodesicPositionOptions (largecounter outputsteps, UpdateFrequency thefrequency)

Public Member Functions inherited from DiagnosticOptions

- DiagnosticOptions (UpdateFrequency thefrequency)
- virtual ∼DiagnosticOptions ()=default

Public Attributes

- const largecounter OutputNrSteps

Public Attributes inherited from DiagnosticOptions

- const UpdateFrequency theUpdateFrequency

6.25.1 Constructor & Destructor Documentation

6.25.1.1 GeodesicPositionOptions()

```
GeodesicPositionOptions::GeodesicPositionOptions (
            largecounter outputsteps,
            UpdateFrequency thefrequency)  [inline]
```

6.25.2 Member Data Documentation

6.25.2.1 OutputNrSteps

```
const largecounter GeodesicPositionOptions::OutputNrSteps
```

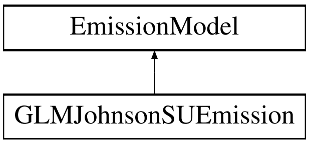
The documentation for this struct was generated from the following file:

- FOORT/src/Diagnostics.h

6.26 GLMJohnsonSUEmission Struct Reference

```
#include <DiagnosticsEmission.h>
```

Inheritance diagram for GLMJohnsonSUEmission:



Public Member Functions

- GLMJohnsonSUEmission (real mu, real gamma, real sigma)
- real GetEmission (const Point &p) const final
- std::string getFullDescriptionStr () const final

  Emission model functions.

Public Member Functions inherited from EmissionModel

- virtual ∼EmissionModel ()=default

Private Attributes

- const real m_mu
- const real m_gamma
- const real m_sigma

6.26.1 Constructor & Destructor Documentation

6.26.1.1 GLMJohnsonSUEmission()

```
GLMJohnsonSUEmission::GLMJohnsonSUEmission (
            real mu,
            real gamma,
            real sigma)  [inline]
```

6.26.2 Member Function Documentation

6.26.2.1 GetEmission()

```
real GLMJohnsonSUEmission::GetEmission (
            const Point & p) const  [final], [virtual]
```

Implements EmissionModel.

6.26.2.2 getFullDescriptionStr()

```
std::string GLMJohnsonSUEmission::getFullDescriptionStr () const  [final], [virtual]
```

Emission model functions.

Reimplemented from EmissionModel.

6.26.3 Member Data Documentation

6.26.3.1 m_gamma

```
const real GLMJohnsonSUEmission::m_gamma  [private]
```

6.26.3.2 m_mu

```
const real GLMJohnsonSUEmission::m_mu  [private]
```

6.26.3.3 m_sigma

```
const real GLMJohnsonSUEmission::m_sigma [private]
```

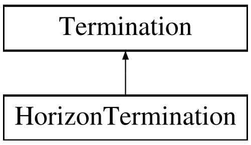
The documentation for this struct was generated from the following files:

- FOORT/src/DiagnosticsEmission.h
- FOORT/src/DiagnosticsEmission.cpp

6.27 HorizonTermination Class Reference

```
#include <Terminations.h>
```

Inheritance diagram for HorizonTermination:



Public Member Functions

- HorizonTermination (Geodesic ∗const theGeodesic)
- Term CheckTermination () final

  HorizonTermination functions.
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from Termination

- Termination ()=delete
- Termination (Geodesic ∗const theGeodesic)
- virtual void Reset ()

  Termination (abstract base class) functions.
- virtual ∼Termination ()=default

Static Public Attributes

- static std::unique_ptr< HorizonTermOptions > TermOptions

Additional Inherited Members

Protected Member Functions inherited from Termination

- bool DecideUpdate (largecounter UpdateNSteps)

Protected Attributes inherited from Termination

- Geodesic ∗const m_OwnerGeodesic
- largecounter m_StepsSinceUpdated {}

6.27.1 Constructor & Destructor Documentation

6.27.1.1 HorizonTermination()

```
HorizonTermination::HorizonTermination (
            Geodesic *const theGeodesic)  [inline]
```

6.27.2 Member Function Documentation

6.27.2.1 CheckTermination()

```
Term HorizonTermination::CheckTermination ()  [final], [virtual]
```

HorizonTermination functions.

Implements Termination.

6.27.2.2 getFullDescriptionStr()

```
std::string HorizonTermination::getFullDescriptionStr () const  [final], [virtual]
```

Implements Termination.

6.27.3 Member Data Documentation

6.27.3.1 TermOptions

```
std::unique_ptr< HorizonTermOptions > HorizonTermination::TermOptions  [static]
```

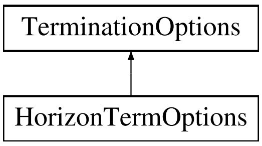
The documentation for this class was generated from the following files:

- FOORT/src/Terminations.h
- FOORT/src/Config.cpp
- FOORT/src/Terminations.cpp

6.28 HorizonTermOptions Struct Reference

```
#include <Terminations.h>
```

Inheritance diagram for HorizonTermOptions:



Public Member Functions

- HorizonTermOptions (real theHorizonRadius, bool therLogScale, real theAtHorizonEps, largecounter Nsteps)

Public Member Functions inherited from TerminationOptions

- TerminationOptions (largecounter Nsteps)
- virtual ∼TerminationOptions ()=default

Public Attributes

- const real HorizonRadius
- const real AtHorizonEps
- const bool rLogScale

Public Attributes inherited from TerminationOptions

- const largecounter UpdateEveryNSteps

6.28.1 Constructor & Destructor Documentation

6.28.1.1 HorizonTermOptions()

```
HorizonTermOptions::HorizonTermOptions (
            real theHorizonRadius,
            bool therLogScale,
            real theAtHorizonEps,
            largecounter Nsteps)  [inline]
```

6.28.2 Member Data Documentation

6.28.2.1 AtHorizonEps

```
const real HorizonTermOptions::AtHorizonEps
```

6.28.2.2 HorizonRadius

```
const real HorizonTermOptions::HorizonRadius
```

6.28.2.3 rLogScale

```
const bool HorizonTermOptions::rLogScale
```

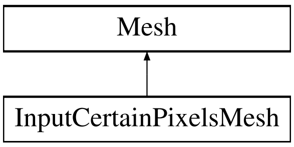
The documentation for this struct was generated from the following file:

- FOORT/src/Terminations.h

6.29 InputCertainPixelsMesh Class Reference

```
#include <Mesh.h>
```

Inheritance diagram for InputCertainPixelsMesh:



Public Member Functions

- InputCertainPixelsMesh ()=delete
- InputCertainPixelsMesh (const InputCertainPixelsMesh &)=delete
- InputCertainPixelsMesh (largecounter totalPixels, DiagBitflag valdiag)

  InputCertainPixelsMesh functions.
- largecounter getCurNrGeodesics () const final
- void getNewInitConds (largecounter index, ScreenPoint &newunitpoint, ScreenIndex &newscreenindex) const final
- void GeodesicFinished (largecounter index, std::vector< real > finalValues) final
- void EndCurrentLoop () final
- bool IsFinished () const final
- std::string getFullDescriptionStr () const final

  Mesh (abstract base class) functions.

Public Member Functions inherited from Mesh

- Mesh (DiagBitflag valdiag)
- virtual ∼Mesh ()=default

Private Attributes

- const pixelcoord m_RowColumnSize
- largecounter m_TotalPixels {0}
- std::vector< ScreenIndex > m_PixelsToIntegrate {}
- bool m_Finished {false}

Additional Inherited Members

Protected Attributes inherited from Mesh

- const std::unique_ptr< const Diagnostic > m_DistanceDiagnostic

6.29.1 Constructor & Destructor Documentation

6.29.1.1 InputCertainPixelsMesh() [1/3]

```
InputCertainPixelsMesh::InputCertainPixelsMesh ()  [delete]
```

6.29.1.2 InputCertainPixelsMesh() [2/3]

```
InputCertainPixelsMesh::InputCertainPixelsMesh (
            const InputCertainPixelsMesh & )  [delete]
```

6.29.1.3 InputCertainPixelsMesh() [3/3]

```
InputCertainPixelsMesh::InputCertainPixelsMesh (
            largecounter totalPixels,
            DiagBitflag valdiag)
```

InputCertainPixelsMesh functions.

6.29.2 Member Function Documentation

6.29.2.1 EndCurrentLoop()

```
void InputCertainPixelsMesh::EndCurrentLoop ()  [final], [virtual]
```

Implements Mesh.

6.29.2.2 GeodesicFinished()

```
void InputCertainPixelsMesh::GeodesicFinished (
            largecounter index,
            std::vector< real > finalValues)  [final], [virtual]
```

Implements Mesh.

6.29.2.3 getCurNrGeodesics()

```
largecounter InputCertainPixelsMesh::getCurNrGeodesics () const  [final], [virtual]
```

Implements Mesh.

6.29.2.4 getFullDescriptionStr()

```
std::string InputCertainPixelsMesh::getFullDescriptionStr () const  [final], [virtual]
```

Mesh (abstract base class) functions.

Reimplemented from Mesh.

6.29.2.5 getNewInitConds()

```
void InputCertainPixelsMesh::getNewInitConds (
             largecounter index,
             ScreenPoint & newunitpoint,
             ScreenIndex & newscreenindex) const  [final], [virtual]
```

Implements Mesh.

6.29.2.6 IsFinished()

```
bool InputCertainPixelsMesh::IsFinished () const  [final], [virtual]
```

Implements Mesh.

6.29.3 Member Data Documentation

6.29.3.1 m_Finished

```
bool InputCertainPixelsMesh::m_Finished {false}  [private]
```

6.29.3.2 m_PixelsToIntegrate

```
std::vector<ScreenIndex> InputCertainPixelsMesh::m_PixelsToIntegrate {}  [private]
```

6.29.3.3 m_RowColumnSize

```
const pixelcoord InputCertainPixelsMesh::m_RowColumnSize  [private]
```

6.29.3.4 m_TotalPixels

```
largecounter InputCertainPixelsMesh::m_TotalPixels {0}  [private]
```

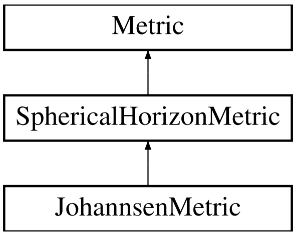
The documentation for this class was generated from the following files:

- FOORT/src/Mesh.h
- FOORT/src/Mesh.cpp

6.30 JohannsenMetric Class Reference

```
#include <Metric.h>
```

Inheritance diagram for JohannsenMetric:



Public Member Functions

- JohannsenMetric ()=delete
- JohannsenMetric (real aParam, real alpha13Param, real alpha22Param, real alpha52Param, real eps3Param, bool rLogScale=false)

  JohannsenMetric functions (implementation by Seppe Staelens)
- TwoIndex getMetric_dd (const Point &p) const final
- TwoIndex getMetric_uu (const Point &p) const final
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from SphericalHorizonMetric

- SphericalHorizonMetric ()=delete
- SphericalHorizonMetric (real HorizonRadius, bool rLogScale)

  SphericalHorizonMetric functions.
- real getHorizonRadius () const

Public Member Functions inherited from Metric

- virtual ∼Metric ()=default
- Metric (bool rlogscale=false)

  Metric (abstract base class) functions.
- virtual ThreeIndex getChristoffel_udd (const Point &p) const
- virtual FourIndex getRiemann_uddd (const Point &p) const
- virtual real getKretschmann (const Point &p) const
- bool getrLogScale () const

Private Attributes

- const real m_aParam
- const real m_alpha13Param
- const real m_alpha22Param
- const real m_alpha52Param
- const real m_eps3Param

Additional Inherited Members

Protected Attributes inherited from SphericalHorizonMetric

- const real m_HorizonRadius

Protected Attributes inherited from Metric

- std::vector< int > m_Symmetries {}
- const bool m_rLogScale

6.30.1 Constructor & Destructor Documentation

6.30.1.1 JohannsenMetric() [1/2]

```
JohannsenMetric::JohannsenMetric ()  [delete]
```

6.30.1.2 JohannsenMetric() [2/2]

```
JohannsenMetric::JohannsenMetric (
            real aParam,
            real alpha13Param,
            real alpha22Param,
            real alpha52Param,
            real eps3Param,
            bool rLogScale = false)
```

JohannsenMetric functions (implementation by Seppe Staelens)

6.30.2 Member Function Documentation

6.30.2.1 getFullDescriptionStr()

```
std::string JohannsenMetric::getFullDescriptionStr () const  [final], [virtual]
```

Reimplemented from Metric.

6.30.2.2 getMetric_dd()

```
TwoIndex JohannsenMetric::getMetric_dd (
            const Point & p) const  [final], [virtual]
```

Implements Metric.

6.30.2.3 getMetric_uu()

```
TwoIndex JohannsenMetric::getMetric_uu (
            const Point & p) const  [final], [virtual]
```

Implements Metric.

6.30.3 Member Data Documentation

6.30.3.1 m_alpha13Param

```
const real JohannsenMetric::m_alpha13Param  [private]
```

6.30.3.2 m_alpha22Param

```
const real JohannsenMetric::m_alpha22Param  [private]
```

6.30.3.3 m_alpha52Param

```
const real JohannsenMetric::m_alpha52Param  [private]
```

6.30.3.4 m_aParam

```
const real JohannsenMetric::m_aParam  [private]
```

6.30.3.5 m_eps3Param

```
const real JohannsenMetric::m_eps3Param  [private]
```

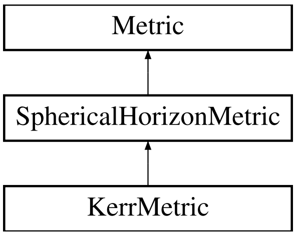
The documentation for this class was generated from the following files:

- FOORT/src/Metric.h
- FOORT/src/Metric.cpp

6.31 KerrMetric Class Reference

```
#include <Metric.h>
```

Inheritance diagram for KerrMetric:



Public Member Functions

- KerrMetric ()=delete
- KerrMetric (real aParam, bool rLogScale=false, real mParam=1.)

  KerrMetric functions.
- TwoIndex getMetric_dd (const Point &p) const final
- TwoIndex getMetric_uu (const Point &p) const final
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from SphericalHorizonMetric

- SphericalHorizonMetric ()=delete
- SphericalHorizonMetric (real HorizonRadius, bool rLogScale)

  SphericalHorizonMetric functions.
- real getHorizonRadius () const

Public Member Functions inherited from Metric

- virtual ∼Metric ()=default
- Metric (bool rlogscale=false)

  Metric (abstract base class) functions.
- virtual ThreeIndex getChristoffel_udd (const Point &p) const
- virtual FourIndex getRiemann_uddd (const Point &p) const
- virtual real getKretschmann (const Point &p) const
- bool getrLogScale () const

Private Attributes

- const real m_aParam
- const real m_mParam

Additional Inherited Members

Protected Attributes inherited from SphericalHorizonMetric

- const real m_HorizonRadius

Protected Attributes inherited from Metric

- std::vector< int > m_Symmetries {}
- const bool m_rLogScale

6.31.1 Constructor & Destructor Documentation

6.31.1.1 KerrMetric() [1/2]

```
KerrMetric::KerrMetric ()  [delete]
```

6.31.1.2 KerrMetric() [2/2]

```
KerrMetric::KerrMetric (
            real aParam,
            bool rLogScale = false,
            real mParam = 1.)
```

KerrMetric functions.

6.31.2 Member Function Documentation

6.31.2.1 getFullDescriptionStr()

```
std::string KerrMetric::getFullDescriptionStr () const  [final], [virtual]
```

Reimplemented from Metric.

6.31.2.2 getMetric_dd()

```
TwoIndex KerrMetric::getMetric_dd (
             const Point & p) const  [final], [virtual]
```

Implements Metric.

6.31.2.3 getMetric_uu()

```
TwoIndex KerrMetric::getMetric_uu (
             const Point & p) const  [final], [virtual]
```

Implements Metric.

6.31.3 Member Data Documentation

6.31.3.1 m_aParam

```
const real KerrMetric::m_aParam  [private]
```

6.31.3.2 m_mParam

```
const real KerrMetric::m_mParam  [private]
```

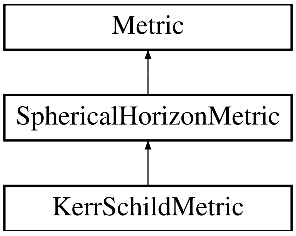
The documentation for this class was generated from the following files:

- FOORT/src/Metric.h
- FOORT/src/Metric.cpp

6.32 KerrSchildMetric Class Reference

```
#include <Metric.h>
```

Inheritance diagram for KerrSchildMetric:



Public Member Functions

- KerrSchildMetric ()=delete
- KerrSchildMetric (real aParam, bool rLogScale=false)

  KerrSchildMetric functions.
- TwoIndex getMetric_dd (const Point &p) const final
- TwoIndex getMetric_uu (const Point &p) const final
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from SphericalHorizonMetric

- SphericalHorizonMetric ()=delete
- SphericalHorizonMetric (real HorizonRadius, bool rLogScale)

  SphericalHorizonMetric functions.
- real getHorizonRadius () const

Public Member Functions inherited from Metric

- virtual ∼Metric ()=default
- Metric (bool rlogscale=false)

  Metric (abstract base class) functions.
- virtual ThreeIndex getChristoffel_udd (const Point &p) const
- virtual FourIndex getRiemann_uddd (const Point &p) const
- virtual real getKretschmann (const Point &p) const
- bool getrLogScale () const

Private Attributes

- const real m_aParam

Additional Inherited Members

Protected Attributes inherited from SphericalHorizonMetric

- const real m_HorizonRadius

Protected Attributes inherited from Metric

- std::vector< int > m_Symmetries {}
- const bool m_rLogScale

6.32.1 Constructor & Destructor Documentation

6.32.1.1 KerrSchildMetric() [1/2]

```
KerrSchildMetric::KerrSchildMetric ()  [delete]
```

6.32.1.2 KerrSchildMetric() [2/2]

```
KerrSchildMetric::KerrSchildMetric (
            real aParam,
            bool rLogScale = false)
```

KerrSchildMetric functions.

6.32.2 Member Function Documentation

6.32.2.1 getFullDescriptionStr()

```
std::string KerrSchildMetric::getFullDescriptionStr () const  [final], [virtual]
```

Reimplemented from Metric.

6.32.2.2 getMetric_dd()

```
TwoIndex KerrSchildMetric::getMetric_dd (
            const Point & p) const  [final], [virtual]
```

Implements Metric.

6.32.2.3 getMetric_uu()

```
TwoIndex KerrSchildMetric::getMetric_uu (
            const Point & p) const  [final], [virtual]
```

Implements Metric.

6.32.3 Member Data Documentation

6.32.3.1 m_aParam

```
const real KerrSchildMetric::m_aParam  [private]
```

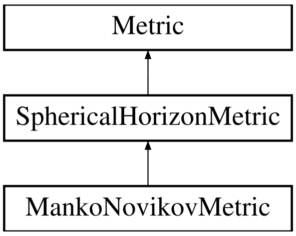
The documentation for this class was generated from the following files:

- FOORT/src/Metric.h
- FOORT/src/Metric.cpp

6.33 MankoNovikovMetric Class Reference

```
#include <Metric.h>
```

Inheritance diagram for MankoNovikovMetric:



Public Member Functions

- MankoNovikovMetric ()=delete
- MankoNovikovMetric (real aParam, real alpha3Param, bool rLogScale=false)

  MankoNovikovMetric functions (implementation by Seppe Staelens)
- TwoIndex getMetric_dd (const Point &p) const final
- TwoIndex getMetric_uu (const Point &p) const final
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from SphericalHorizonMetric

- SphericalHorizonMetric ()=delete
- SphericalHorizonMetric (real HorizonRadius, bool rLogScale)

  SphericalHorizonMetric functions.
- real getHorizonRadius () const

Public Member Functions inherited from Metric

- virtual ∼Metric ()=default
- Metric (bool rlogscale=false)

  Metric (abstract base class) functions.
- virtual ThreeIndex getChristoffel_udd (const Point &p) const
- virtual FourIndex getRiemann_uddd (const Point &p) const
- virtual real getKretschmann (const Point &p) const
- bool getrLogScale () const

Private Attributes

- const real m_aParam
- const real m_alpha3Param
- const real m_alphaParam
- const real m_kParam

Additional Inherited Members

Protected Attributes inherited from SphericalHorizonMetric

- const real m_HorizonRadius

Protected Attributes inherited from Metric

- std::vector< int > m_Symmetries {}
- const bool m_rLogScale

6.33.1 Constructor & Destructor Documentation

6.33.1.1 MankoNovikovMetric() [1/2]

```
MankoNovikovMetric::MankoNovikovMetric ()  [delete]
```

6.33.1.2 MankoNovikovMetric() [2/2]

```
MankoNovikovMetric::MankoNovikovMetric (
             real aParam,
             real alpha3Param,
             bool rLogScale = false)
```

MankoNovikovMetric functions (implementation by Seppe Staelens)

6.33.2 Member Function Documentation

6.33.2.1 getFullDescriptionStr()

```
std::string MankoNovikovMetric::getFullDescriptionStr () const  [final], [virtual]
```

Reimplemented from Metric.

6.33.2.2 getMetric_dd()

```
TwoIndex MankoNovikovMetric::getMetric_dd (
             const Point & p) const  [final], [virtual]
```

Implements Metric.

6.33.2.3 getMetric_uu()

```
TwoIndex MankoNovikovMetric::getMetric_uu (
             const Point & p) const  [final], [virtual]
```

Implements Metric.

6.33.3 Member Data Documentation

6.33.3.1 m_alpha3Param

```
const real MankoNovikovMetric::m_alpha3Param  [private]
```

6.33.3.2 m_alphaParam

```
const real MankoNovikovMetric::m_alphaParam  [private]
```

6.33.3.3 m_aParam

```
const real MankoNovikovMetric::m_aParam  [private]
```

6.33.3.4 m_kParam

```
const real MankoNovikovMetric::m_kParam  [private]
```

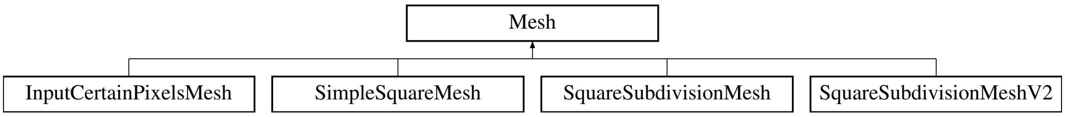
The documentation for this class was generated from the following files:

- FOORT/src/Metric.h
- FOORT/src/Metric.cpp

6.34 Mesh Class Reference

```
#include <Mesh.h>
```

Inheritance diagram for Mesh:



Public Member Functions

- Mesh (DiagBitflag valdiag)
- virtual ∼Mesh ()=default
- virtual largecounter getCurNrGeodesics () const =0
- virtual void getNewInitConds (largecounter index, ScreenPoint &newunitpoint, ScreenIndex &newscreenindex) const =0
- virtual void GeodesicFinished (largecounter index, std::vector< real > finalValues)=0
- virtual void EndCurrentLoop ()=0
- virtual bool IsFinished () const =0
- virtual std::string getFullDescriptionStr () const

  Mesh (abstract base class) functions.

Protected Attributes

- const std::unique_ptr< const Diagnostic > m_DistanceDiagnostic

6.34.1 Constructor & Destructor Documentation

6.34.1.1 Mesh()

```
Mesh::Mesh (
            DiagBitflag valdiag)  [inline]
```

6.34.1.2 ∼Mesh()

```
virtual Mesh::∼Mesh ()  [virtual], [default]
```

6.34.2 Member Function Documentation

6.34.2.1 EndCurrentLoop()

```
virtual void Mesh::EndCurrentLoop ()  [pure virtual]
```

Implemented in InputCertainPixelsMesh, SimpleSquareMesh, SquareSubdivisionMesh, and SquareSubdivisionMeshV2.

6.34.2.2 GeodesicFinished()

```
virtual void Mesh::GeodesicFinished (
            largecounter index,
            std::vector< real > finalValues)  [pure virtual]
```

Implemented in InputCertainPixelsMesh, SimpleSquareMesh, SquareSubdivisionMesh, and SquareSubdivisionMeshV2.

6.34.2.3 getCurNrGeodesics()

```
virtual largecounter Mesh::getCurNrGeodesics () const  [pure virtual]
```

Implemented in InputCertainPixelsMesh, SimpleSquareMesh, SquareSubdivisionMesh, and SquareSubdivisionMeshV2.

6.34.2.4 getFullDescriptionStr()

```
std::string Mesh::getFullDescriptionStr () const  [virtual]
```

Mesh (abstract base class) functions.

Reimplemented in InputCertainPixelsMesh, SimpleSquareMesh, SquareSubdivisionMesh, and SquareSubdivisionMeshV2.

6.34.2.5 getNewInitConds()

```
virtual void Mesh::getNewInitConds (
             largecounter index,
             ScreenPoint & newunitpoint,
             ScreenIndex & newscreenindex) const  [pure virtual]
```

Implemented in InputCertainPixelsMesh, SimpleSquareMesh, SquareSubdivisionMesh, and SquareSubdivisionMeshV2.

6.34.2.6 IsFinished()

```
virtual bool Mesh::IsFinished () const  [pure virtual]
```

Implemented in InputCertainPixelsMesh, SimpleSquareMesh, SquareSubdivisionMesh, and SquareSubdivisionMeshV2.

6.34.3 Member Data Documentation

6.34.3.1 m_DistanceDiagnostic

```
const std::unique_ptr<const Diagnostic> Mesh::m_DistanceDiagnostic  [protected]
```

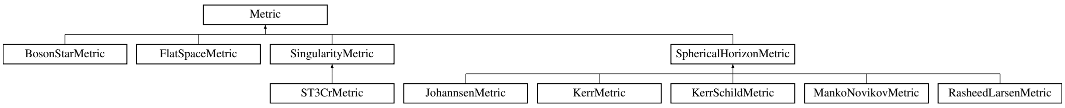
The documentation for this class was generated from the following files:

- FOORT/src/Mesh.h
- FOORT/src/Mesh.cpp

6.35 Metric Class Reference

```
#include <Metric.h>
```

Inheritance diagram for Metric:



Public Member Functions

- virtual ~Metric ()=default
- Metric (bool rlogscale=false)

  Metric (abstract base class) functions.
- virtual TwoIndex getMetric_dd (const Point &p) const =0
- virtual TwoIndex getMetric_uu (const Point &p) const =0
- virtual ThreeIndex getChristoffel_udd (const Point &p) const
- virtual FourIndex getRiemann_uddd (const Point &p) const
- virtual real getKretschmann (const Point &p) const
- virtual std::string getFullDescriptionStr () const
- bool getrLogScale () const

Protected Attributes

- std::vector< int > m_Symmetries {}
- const bool m_rLogScale

6.35.1 Constructor & Destructor Documentation

6.35.1.1 ∼Metric()

```
virtual Metric::~Metric ()  [virtual], [default]
```

6.35.1.2 Metric()

```
Metric::Metric (
             bool rlogscale = false)
```

Metric (abstract base class) functions.

6.35.2 Member Function Documentation

6.35.2.1 getChristoffel_udd()

```
ThreeIndex Metric::getChristoffel_udd (
             const Point & p) const  [virtual]
```

6.35.2.2 getFullDescriptionStr()

```
std::string Metric::getFullDescriptionStr () const  [virtual]
```

Reimplemented in BosonStarMetric, FlatSpaceMetric, JohannsenMetric, KerrMetric, KerrSchildMetric, MankoNovikovMetric, RasheedLarsenMetric, and ST3CrMetric.

6.35.2.3 getKretschmann()

```
real Metric::getKretschmann (
             const Point & p) const  [virtual]
```

6.35.2.4 getMetric_dd()

```
virtual TwoIndex Metric::getMetric_dd (
             const Point & p) const  [pure virtual]
```

Implemented in BosonStarMetric, FlatSpaceMetric, JohannsenMetric, KerrMetric, KerrSchildMetric, MankoNovikovMetric, RasheedLarsenMetric, and ST3CrMetric.

6.35.2.5 getMetric_uu()

```
virtual TwoIndex Metric::getMetric_uu (
             const Point & p) const  [pure virtual]
```

Implemented in BosonStarMetric, FlatSpaceMetric, JohannsenMetric, KerrMetric, KerrSchildMetric, ManKoNovikovMetric, RasheedLarsenMetric, and ST3CrMetric.

6.35.2.6 getRiemann_uddd()

```
FourIndex Metric::getRiemann_uddd (
             const Point & p) const  [virtual]
```

6.35.2.7 getrLogScale()

```
bool Metric::getrLogScale () const
```

6.35.3 Member Data Documentation

6.35.3.1 m_rLogScale

```
const bool Metric::m_rLogScale  [protected]
```

6.35.3.2 m_Symmetries

```
std::vector<int> Metric::m_Symmetries {}  [protected]
```

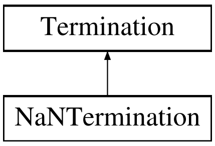
The documentation for this class was generated from the following files:

- FOORT/src/Metric.h
- FOORT/src/Metric.cpp

6.36 NaNTermination Class Reference

```
#include <Terminations.h>
```

Inheritance diagram for NaNTermination:



Public Member Functions

- NaNTermination (Geodesic ∗const theGeodesic)
- Term CheckTermination () final

  NaNTermination functions.
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from Termination

- Termination ()=delete
- Termination (Geodesic ∗const theGeodesic)
- virtual void Reset ()

  Termination (abstract base class) functions.
- virtual ∼Termination ()=default

Static Public Attributes

- static std::unique_ptr< NaNTermOptions > TermOptions

Additional Inherited Members

Protected Member Functions inherited from Termination

- bool DecideUpdate (largecounter UpdateNSteps)

Protected Attributes inherited from Termination

- Geodesic ∗const m_OwnerGeodesic
- largecounter m_StepsSinceUpdated {}

6.36.1 Constructor & Destructor Documentation

6.36.1.1 NaNTermination()

```
NaNTermination::NaNTermination (
             Geodesic *const theGeodesic)  [inline]
```

6.36.2 Member Function Documentation

6.36.2.1 CheckTermination()

```
Term NaNTermination::CheckTermination ()  [final], [virtual]
```

NaNTermination functions.

Implements Termination.

6.36.2.2 getFullDescriptionStr()

```
std::string NaNTermination::getFullDescriptionStr () const  [final], [virtual]
```

Implements Termination.

6.36.3 Member Data Documentation

6.36.3.1 TermOptions

```
std::unique_ptr< NaNTermOptions > NaNTermination::TermOptions  [static]
```

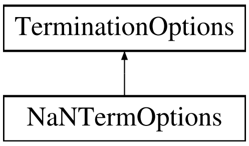
The documentation for this class was generated from the following files:

- FOORT/src/Terminations.h
- FOORT/src/Config.cpp
- FOORT/src/Terminations.cpp

6.37 NaNTermOptions Struct Reference

```
#include <Terminations.h>
```

Inheritance diagram for NaNTermOptions:



Public Member Functions

- NaNTermOptions (bool consoleoutputon, largecounter Nsteps)

Public Member Functions inherited from TerminationOptions

- TerminationOptions (largecounter Nsteps)
- virtual ∼TerminationOptions ()=default

Public Attributes

- const bool OutputToConsole

Public Attributes inherited from TerminationOptions

- const largecounter UpdateEveryNSteps

6.37.1 Constructor & Destructor Documentation

6.37.1.1 NaNTermOptions()

```
NaNTermOptions::NaNTermOptions (
             bool consoleoutputon,
             largecounter Nsteps)  [inline]
```

6.37.2 Member Data Documentation

6.37.2.1 OutputToConsole

```
const bool NaNTermOptions::OutputToConsole
```

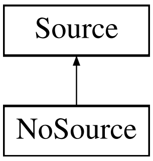
The documentation for this struct was generated from the following file:

- FOORT/src/Terminations.h

6.38 NoSource Class Reference

```
#include <Geodesic.h>
```

Inheritance diagram for NoSource:



Public Member Functions

- NoSource (const Metric ∗const theMetric)
- OneIndex getSource (Point pos, OneIndex vel) const final
- std::string getFullDescriptionStr () const final

  Source (and descendant classes) functions.

Public Member Functions inherited from Source

- Source (const Metric ∗const theMetric)
- virtual ∼Source ()=default

Additional Inherited Members

Protected Attributes inherited from Source

- const Metric ∗const m_theMetric

6.38.1 Constructor & Destructor Documentation

6.38.1.1 NoSource()

```
NoSource::NoSource (
             const Metric *const theMetric)  [inline]
```

6.38.2 Member Function Documentation

6.38.2.1 getFullDescriptionStr()

```
std::string NoSource::getFullDescriptionStr () const  [final], [virtual]
```

Source (and descendant classes) functions.

Reimplemented from Source.

6.38.2.2 getSource()

```
OneIndex NoSource::getSource (
             Point pos,
             OneIndex vel) const  [final], [virtual]
```

Implements Source.

The documentation for this class was generated from the following files:

- FOORT/src/Geodesic.h
- FOORT/src/Geodesic.cpp

6.39 SquareSubdivisionMesh::PixelInfo Struct Reference

Public Member Functions

- PixelInfo (ScreenIndex ind, int subdiv)

Public Attributes

- ScreenIndex Index {}
- int SubdivideLevel {}
- real Weight {-1}
- std::vector< real > DiagValue {}
- largecounter LowerNbrIndex {0}
- largecounter RightNbrIndex {0}

6.39.1 Constructor & Destructor Documentation

6.39.1.1 PixelInfo()

```
SquareSubdivisionMesh::PixelInfo::PixelInfo (
            ScreenIndex ind,
            int subdiv)  [inline]
```

6.39.2 Member Data Documentation

6.39.2.1 DiagValue

```
std::vector<real> SquareSubdivisionMesh::PixelInfo::DiagValue {}
```

6.39.2.2 Index

```
ScreenIndex SquareSubdivisionMesh::PixelInfo::Index {}
```

6.39.2.3 LowerNbrIndex

```
largecounter SquareSubdivisionMesh::PixelInfo::LowerNbrIndex {0}
```

6.39.2.4 RightNbrIndex

```
largecounter SquareSubdivisionMesh::PixelInfo::RightNbrIndex {0}
```

6.39.2.5 SubdivideLevel

```
int SquareSubdivisionMesh::PixelInfo::SubdivideLevel {}
```

6.39.2.6 Weight

```
real SquareSubdivisionMesh::PixelInfo::Weight {-1}
```

The documentation for this struct was generated from the following file:

- FOORT/src/Mesh.h

6.40 SquareSubdivisionMeshV2::PixelInfo Struct Reference

Public Member Functions

- PixelInfo (ScreenIndex ind, int subdiv)

Public Attributes

- const ScreenIndex Index {}
- int SubdivideLevel {}
- real Weight {-1}
- std::vector< real > DiagValue {}
- PixelInfo ∗ LeftNbr {nullptr}
- PixelInfo ∗ RightNbr {nullptr}
- PixelInfo ∗ UpNbr {nullptr}
- PixelInfo ∗ DownNbr {nullptr}
- PixelInfo ∗ SEdiagNbr {nullptr}

6.40.1 Constructor & Destructor Documentation

6.40.1.1 PixelInfo()

```
SquareSubdivisionMeshV2::PixelInfo::PixelInfo (
             ScreenIndex ind,
             int subdiv)  [inline]
```

6.40.2 Member Data Documentation

6.40.2.1 DiagValue

```
std::vector<real> SquareSubdivisionMeshV2::PixelInfo::DiagValue {}
```

6.40.2.2 DownNbr

```
PixelInfo* SquareSubdivisionMeshV2::PixelInfo::DownNbr {nullptr}
```

6.40.2.3 Index

```
const ScreenIndex SquareSubdivisionMeshV2::PixelInfo::Index {}
```

6.40.2.4 LeftNbr

```
PixelInfo* SquareSubdivisionMeshV2::PixelInfo::LeftNbr {nullptr}
```

6.40.2.5 RightNbr

```
PixelInfo* SquareSubdivisionMeshV2::PixelInfo::RightNbr {nullptr}
```

6.40.2.6 SEdiagNbr

```
PixelInfo* SquareSubdivisionMeshV2::PixelInfo::SEdiagNbr {nullptr}
```

6.40.2.7 SubdivideLevel

```
int SquareSubdivisionMeshV2::PixelInfo::SubdivideLevel {}
```

6.40.2.8 UpNbr

```
PixelInfo* SquareSubdivisionMeshV2::PixelInfo::UpNbr {nullptr}
```

6.40.2.9 Weight

```
real SquareSubdivisionMeshV2::PixelInfo::Weight {-1}
```

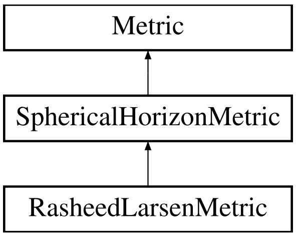
The documentation for this struct was generated from the following file:

- FOORT/src/Mesh.h

6.41 RasheedLarsenMetric Class Reference

```
#include <Metric.h>
```

Inheritance diagram for RasheedLarsenMetric:



Public Member Functions

- RasheedLarsenMetric ()=delete
- RasheedLarsenMetric (real mParam, real aParam, real pParam, real qParam, bool rLogScale=false)

  RasheedLarsenMetric functions.
- TwoIndex getMetric_dd (const Point &p) const final
- TwoIndex getMetric_uu (const Point &p) const final
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from SphericalHorizonMetric

- SphericalHorizonMetric ()=delete
- SphericalHorizonMetric (real HorizonRadius, bool rLogScale)

  SphericalHorizonMetric functions.
- real getHorizonRadius () const

Public Member Functions inherited from Metric

- virtual ~Metric ()=default
- Metric (bool rlogscale=false)

  Metric (abstract base class) functions.
- virtual ThreeIndex getChristoffel_udd (const Point &p) const
- virtual FourIndex getRiemann_uddd (const Point &p) const
- virtual real getKretschmann (const Point &p) const
- bool getrLogScale () const

Private Attributes

- const real m_aParam
- const real m_mParam
- const real m_pParam
- const real m_qParam

Additional Inherited Members

Protected Attributes inherited from SphericalHorizonMetric

- const real m_HorizonRadius

Protected Attributes inherited from Metric

- std::vector< int > m_Symmetries {}
- const bool m_rLogScale

6.41.1 Constructor & Destructor Documentation

6.41.1.1 RasheedLarsenMetric() [1/2]

```
RasheedLarsenMetric::RasheedLarsenMetric ()  [delete]
```

6.41.1.2 RasheedLarsenMetric() [2/2]

```
RasheedLarsenMetric::RasheedLarsenMetric (
              real mParam,
              real aParam,
              real pParam,
              real qParam,
              bool rLogScale = false)
```

RasheedLarsenMetric functions.

6.41.2 Member Function Documentation

6.41.2.1 getFullDescriptionStr()

```
std::string RasheedLarsenMetric::getFullDescriptionStr () const  [final], [virtual]
```

Reimplemented from Metric.

6.41.2.2 getMetric_dd()

```
TwoIndex RasheedLarsenMetric::getMetric_dd (
             const Point & p) const  [final], [virtual]
```

Implements Metric.

6.41.2.3 getMetric_uu()

```
TwoIndex RasheedLarsenMetric::getMetric_uu (
             const Point & p) const  [final], [virtual]
```

Implements Metric.

6.41.3 Member Data Documentation

6.41.3.1 m_aParam

```
const real RasheedLarsenMetric::m_aParam  [private]
```

6.41.3.2 m_mParam

```
const real RasheedLarsenMetric::m_mParam  [private]
```

6.41.3.3 m_pParam

```
const real RasheedLarsenMetric::m_pParam  [private]
```

6.41.3.4 m_qParam

```
const real RasheedLarsenMetric::m_qParam  [private]
```

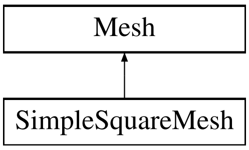
The documentation for this class was generated from the following files:

- FOORT/src/Metric.h
- FOORT/src/Metric.cpp

6.42 SimpleSquareMesh Class Reference

```
#include <Mesh.h>
```

Inheritance diagram for SimpleSquareMesh:



Public Member Functions

- SimpleSquareMesh ()=delete
- SimpleSquareMesh (largecounter totalPixels, DiagBitflag valdiag)
- largecounter getCurNrGeodesics () const final
- void getNewInitConds (largecounter index, ScreenPoint &newunitpoint, ScreenIndex &newscreenindex) const final
- void GeodesicFinished (largecounter index, std::vector< real > finalValues) final
- void EndCurrentLoop () final
- bool IsFinished () const final

  SimpleSquareMesh functions.
- std::string getFullDescriptionStr () const final

  Mesh (abstract base class) functions.

Public Member Functions inherited from Mesh

- Mesh (DiagBitflag valdiag)
- virtual ∼Mesh ()=default

Private Attributes

- const largecounter m_TotalPixels
- const pixelcoord m_RowColumnSize
- bool m_Finished {false}

Additional Inherited Members

Protected Attributes inherited from Mesh

- const std::unique_ptr< const Diagnostic > m_DistanceDiagnostic

6.42.1 Constructor & Destructor Documentation

6.42.1.1 SimpleSquareMesh() [1/2]

```
SimpleSquareMesh::SimpleSquareMesh ()  [delete]
```

6.42.1.2 SimpleSquareMesh() [2/2]

```
SimpleSquareMesh::SimpleSquareMesh (
            largecounter totalPixels,
            DiagBitflag valdiag)  [inline]
```

6.42.2 Member Function Documentation

6.42.2.1 EndCurrentLoop()

```
void SimpleSquareMesh::EndCurrentLoop ()  [final], [virtual]
```

Implements Mesh.

6.42.2.2 GeodesicFinished()

```
void SimpleSquareMesh::GeodesicFinished (
            largecounter index,
            std::vector< real > finalValues)  [final], [virtual]
```

Implements Mesh.

6.42.2.3 getCurNrGeodesics()

```
largecounter SimpleSquareMesh::getCurNrGeodesics () const  [final], [virtual]
```

Implements Mesh.

6.42.2.4 getFullDescriptionStr()

```
std::string SimpleSquareMesh::getFullDescriptionStr () const  [final], [virtual]
```

Mesh (abstract base class) functions.

Reimplemented from Mesh.

6.42.2.5 getNewInitConds()

```
void SimpleSquareMesh::getNewInitConds (
            largecounter index,
            ScreenPoint & newunitpoint,
            ScreenIndex & newscreenindex) const  [final], [virtual]
```

Implements Mesh.

6.42.2.6 IsFinished()

```
bool SimpleSquareMesh::IsFinished () const  [final], [virtual]
```

SimpleSquareMesh functions.

Implements Mesh.

6.42.3 Member Data Documentation

6.42.3.1 m_Finished

```
bool SimpleSquareMesh::m_Finished {false}  [private]
```

6.42.3.2 m_RowColumnSize

```
const pixelcoord SimpleSquareMesh::m_RowColumnSize  [private]
```

6.42.3.3 m_TotalPixels

```
const largecounter SimpleSquareMesh::m_TotalPixels  [private]
```

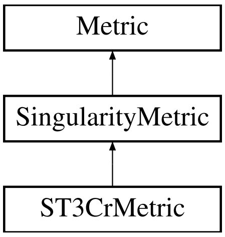
The documentation for this class was generated from the following files:

- FOORT/src/Mesh.h
- FOORT/src/Mesh.cpp

6.43 SingularityMetric Class Reference

```
#include <Metric.h>
```

Inheritance diagram for SingularityMetric:



Public Member Functions

- SingularityMetric (std::vector< Singularity > thesings, bool rLogScale)

  SingularityMetric functions.
- std::vector< Singularity > getSingularities () const

Public Member Functions inherited from Metric

- virtual ∼Metric ()=default
- Metric (bool rlogscale=false)

    Metric (abstract base class) functions.
- virtual TwoIndex getMetric_dd (const Point &p) const =0
- virtual TwoIndex getMetric_uu (const Point &p) const =0
- virtual ThreeIndex getChristoffel_udd (const Point &p) const
- virtual FourIndex getRiemann_uddd (const Point &p) const
- virtual real getKretschmann (const Point &p) const
- virtual std::string getFullDescriptionStr () const
- bool getrLogScale () const

Protected Attributes

- const std::vector< Singularity > m_AllSingularities

Protected Attributes inherited from Metric

- std::vector< int > m_Symmetries {}
- const bool m_rLogScale

6.43.1 Constructor & Destructor Documentation

6.43.1.1 SingularityMetric()

```
SingularityMetric::SingularityMetric (
            std::vector< Singularity > thesings,
            bool rLogScale)
```

SingularityMetric functions.

6.43.2 Member Function Documentation

6.43.2.1 getSingularities()

```
std::vector< Singularity > SingularityMetric::getSingularities () const
```

6.43.3 Member Data Documentation

6.43.3.1 m_AllSingularities

```
const std::vector<Singularity> SingularityMetric::m_AllSingularities  [protected]
```

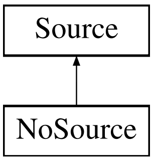
The documentation for this class was generated from the following files:

- FOORT/src/Metric.h
- FOORT/src/Metric.cpp

6.44 Source Class Reference

```
#include <Geodesic.h>
```

Inheritance diagram for Source:



Public Member Functions

- Source (const Metric ∗const theMetric)
- virtual ∼Source ()=default
- virtual OneIndex getSource (Point pos, OneIndex vel) const =0
- virtual std::string getFullDescriptionStr () const

  Source (and descendant classes) functions.

Protected Attributes

- const Metric ∗const m_theMetric

6.44.1 Constructor & Destructor Documentation

6.44.1.1 Source()

```
Source::Source (
             const Metric ∗const theMetric)  [inline]
```

6.44.1.2 ∼Source()

```
virtual Source::∼Source ()  [virtual], [default]
```

6.44.2 Member Function Documentation

6.44.2.1 getFullDescriptionStr()

```
std::string Source::getFullDescriptionStr () const  [virtual]
```

Source (and descendant classes) functions.

Reimplemented in NoSource.

6.44.2.2 getSource()

```
virtual OneIndex Source::getSource (
            Point pos,
            OneIndex vel) const  [pure virtual]
```

Implemented in NoSource.

6.44.3 Member Data Documentation

6.44.3.1 m_theMetric

```
const Metric* const Source::m_theMetric  [protected]
```

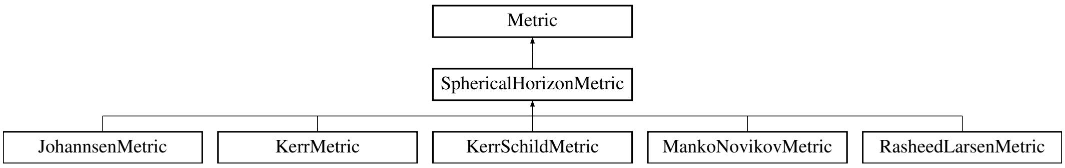
The documentation for this class was generated from the following files:

- FOORT/src/Geodesic.h
- FOORT/src/Geodesic.cpp

6.45 SphericalHorizonMetric Class Reference

```
#include <Metric.h>
```

Inheritance diagram for SphericalHorizonMetric:



Public Member Functions

- SphericalHorizonMetric ()=delete
- SphericalHorizonMetric (real HorizonRadius, bool rLogScale)

  SphericalHorizonMetric functions.
- real getHorizonRadius () const

Public Member Functions inherited from Metric

- virtual ∼Metric ()=default
- Metric (bool rlogscale=false)

  Metric (abstract base class) functions.
- virtual TwoIndex getMetric_dd (const Point &p) const =0
- virtual TwoIndex getMetric_uu (const Point &p) const =0
- virtual ThreeIndex getChristoffel_udd (const Point &p) const
- virtual FourIndex getRiemann_uddd (const Point &p) const
- virtual real getKretschmann (const Point &p) const
- virtual std::string getFullDescriptionStr () const
- bool getrLogScale () const

Protected Attributes

- const real m_HorizonRadius

Protected Attributes inherited from Metric

- std::vector< int > m_Symmetries {}
- const bool m_rLogScale

6.45.1 Constructor & Destructor Documentation

6.45.1.1 SphericalHorizonMetric() [1/2]

```
SphericalHorizonMetric::SphericalHorizonMetric ()  [delete]
```

6.45.1.2 SphericalHorizonMetric() [2/2]

```
SphericalHorizonMetric::SphericalHorizonMetric (
             real HorizonRadius,
             bool rLogScale)
```

SphericalHorizonMetric functions.

6.45.2 Member Function Documentation

6.45.2.1 getHorizonRadius()

```
real SphericalHorizonMetric::getHorizonRadius () const
```

6.45.3 Member Data Documentation

6.45.3.1 m_HorizonRadius

```
const real SphericalHorizonMetric::m_HorizonRadius  [protected]
```

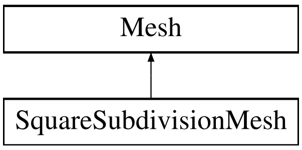
The documentation for this class was generated from the following files:

- FOORT/src/Metric.h
- FOORT/src/Metric.cpp

6.46 SquareSubdivisionMesh Class Reference

```
#include <Mesh.h>
```

Inheritance diagram for SquareSubdivisionMesh:



Classes

- struct PixelInfo

Public Member Functions

- SquareSubdivisionMesh ()=delete
- SquareSubdivisionMesh (largecounter maxPixels, largecounter initialPixels, int maxSubdivide, largecounter iterationPixels, bool initialSubToFinal, DiagBitflag valdiag)
- largecounter getCurNrGeodesics () const final

  SquareSubdivisionMesh functions.
- void getNewInitConds (largecounter index, ScreenPoint &newunitpoint, ScreenIndex &newscreenindex) const final
- void GeodesicFinished (largecounter index, std::vector< real > finalValues) final
- void EndCurrentLoop () final
- bool IsFinished () const final
- std::string getFullDescriptionStr () const final

  Mesh (abstract base class) functions.

Public Member Functions inherited from Mesh

- Mesh (DiagBitflag valdiag)
- virtual ∼Mesh ()=default

Private Member Functions

- void InitializeFirstGrid ()
- void UpdateAllNeighbors ()
- void UpdateAllWeights ()
- void SubdivideAndQueue (largecounter ind)
- pixelcoord ExpInt (int base, int exp)

Private Attributes

- const largecounter m_InitialPixels
- const int m_MaxSubdivide
- const pixelcoord m_RowColumnSize
- const largecounter m_IterationPixels
- const largecounter m_MaxPixels
- const bool m_InitialSubDividideToFinal
- const bool m_InfinitePixels
- largecounter m_PixelsLeft
- std::vector< PixelInfo > m_CurrentPixelQueue {}
- std::vector< bool > m_CurrentPixelQueueDone {}
- std::vector< PixelInfo > m_AllPixels {}

Additional Inherited Members

Protected Attributes inherited from Mesh

- const std::unique_ptr< const Diagnostic > m_DistanceDiagnostic

6.46.1 Constructor & Destructor Documentation

6.46.1.1 SquareSubdivisionMesh() [1/2]

```
SquareSubdivisionMesh::SquareSubdivisionMesh ()  [delete]
```

6.46.1.2 SquareSubdivisionMesh() [2/2]

```
SquareSubdivisionMesh::SquareSubdivisionMesh (
             largecounter maxPixels,
             largecounter initialPixels,
             int maxSubdivide,
             largecounter iterationPixels,
             bool initialSubToFinal,
             DiagBitflag valdiag)  [inline]
```

6.46.2 Member Function Documentation

6.46.2.1 EndCurrentLoop()

```
void SquareSubdivisionMesh::EndCurrentLoop ()  [final], [virtual]
```

Implements Mesh.

6.46.2.2 ExpInt()

```
pixelcoord SquareSubdivisionMesh::ExpInt (
             int base,
             int exp)  [private]
```

6.46.2.3 GeodesicFinished()

```
void SquareSubdivisionMesh::GeodesicFinished (
            largecounter index,
            std::vector< real > finalValues)  [final], [virtual]
```

Implements Mesh.

6.46.2.4 getCurNrGeodesics()

```
largecounter SquareSubdivisionMesh::getCurNrGeodesics () const  [final], [virtual]
```

SquareSubdivisionMesh functions.

Implements Mesh.

6.46.2.5 getFullDescriptionStr()

```
std::string SquareSubdivisionMesh::getFullDescriptionStr () const  [final], [virtual]
```

Mesh (abstract base class) functions.

Reimplemented from Mesh.

6.46.2.6 getNewInitConds()

```
void SquareSubdivisionMesh::getNewInitConds (
            largecounter index,
            ScreenPoint & newunitpoint,
            ScreenIndex & newscreenindex) const  [final], [virtual]
```

Implements Mesh.

6.46.2.7 InitializeFirstGrid()

```
void SquareSubdivisionMesh::InitializeFirstGrid ()  [private]
```

6.46.2.8 IsFinished()

```
bool SquareSubdivisionMesh::IsFinished () const  [final], [virtual]
```

Implements Mesh.

6.46.2.9 SubdivideAndQueue()

```
void SquareSubdivisionMesh::SubdivideAndQueue (
            largecounter ind)  [private]
```

6.46.2.10 UpdateAllNeighbors()

```
void SquareSubdivisionMesh::UpdateAllNeighbors ()  [private]
```

6.46.2.11 UpdateAllWeights()

```
void SquareSubdivisionMesh::UpdateAllWeights ()  [private]
```

6.46.3 Member Data Documentation

6.46.3.1 m_AllPixels

```
std::vector<PixelInfo> SquareSubdivisionMesh::m_AllPixels {}  [private]
```

6.46.3.2 m_CurrentPixelQueue

```
std::vector<PixelInfo> SquareSubdivisionMesh::m_CurrentPixelQueue {}  [private]
```

6.46.3.3 m_CurrentPixelQueueDone

```
std::vector<bool> SquareSubdivisionMesh::m_CurrentPixelQueueDone {}  [private]
```

6.46.3.4 m_InfinitePixels

```
const bool SquareSubdivisionMesh::m_InfinitePixels  [private]
```

6.46.3.5 m_InitialPixels

```
const largecounter SquareSubdivisionMesh::m_InitialPixels  [private]
```

6.46.3.6 m_InitialSubDivideToFinal

```
const bool SquareSubdivisionMesh::m_InitialSubDivideToFinal  [private]
```

6.46.3.7 m_IterationPixels

```
const largecounter SquareSubdivisionMesh::m_IterationPixels  [private]
```

6.46.3.8 m_MaxPixels

```
const largecounter SquareSubdivisionMesh::m_MaxPixels  [private]
```

6.46.3.9 m_MaxSubdivide

```
const int SquareSubdivisionMesh::m_MaxSubdivide  [private]
```

6.46.3.10 m_PixelsLeft

```
largecounter SquareSubdivisionMesh::m_PixelsLeft  [private]
```

6.46.3.11 m_RowColumnSize

```
const pixelcoord SquareSubdivisionMesh::m_RowColumnSize  [private]
```

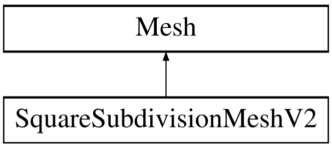
The documentation for this class was generated from the following files:

- FOORT/src/Mesh.h
- FOORT/src/Mesh.cpp

6.47 SquareSubdivisionMeshV2 Class Reference

```
#include <Mesh.h>
```

Inheritance diagram for SquareSubdivisionMeshV2:



Classes

- struct PixelInfo

Public Member Functions

- SquareSubdivisionMeshV2 ()=delete
- SquareSubdivisionMeshV2 (largecounter maxPixels, largecounter initialPixels, int maxSubdivide, largecounter iterationPixels, bool initialSubToFinal, DiagBitflag valdiag)
- largecounter getCurNrGeodesics () const final

  SquareSubdivisionMeshV2 functions.
- void getNewInitConds (largecounter index, ScreenPoint &newunitpoint, ScreenIndex &newscreenindex) const final
- void GeodesicFinished (largecounter index, std::vector< real > finalValues) final
- void EndCurrentLoop () final
- bool IsFinished () const final
- std::string getFullDescriptionStr () const final

  Mesh (abstract base class) functions.

Public Member Functions inherited from Mesh

- Mesh (DiagBitflag valdiag)
- virtual ∼Mesh ()=default

Private Member Functions

- void InitializeFirstGrid ()
- void UpdateAllWeights ()
- PixelInfo ∗ GetUp (PixelInfo ∗p, int subdiv) const
- PixelInfo ∗ GetDown (PixelInfo ∗p, int subdiv) const
- PixelInfo ∗ GetRight (PixelInfo ∗p, int subdiv) const
- PixelInfo ∗ GetLeft (PixelInfo ∗p, int subdiv) const
- void SubdivideAndQueue (largecounter ind)
- pixelcoord ExpInt (int base, int exp) const

Private Attributes

- const largecounter m_InitialPixels
- const int m_MaxSubdivide
- const pixelcoord m_RowColumnSize
- const largecounter m_IterationPixels
- const largecounter m_MaxPixels
- const bool m_InitialSubDividideToFinal
- const bool m_InfinitePixels
- largecounter m_PixelsLeft
- largecounter m_PixelsIntegrated {0}
- std::forward_list< std::unique_ptr< PixelInfo > > m_AllPixels {}
- std::vector< PixelInfo ∗ > m_ActivePixels {}
- std::vector< PixelInfo ∗ > m_CurrentPixelQueue {}
- std::vector< bool > m_CurrentPixelQueueDone {}
- std::vector< PixelInfo ∗ > m_CurrentPixelUpdating {}

Additional Inherited Members

Protected Attributes inherited from Mesh

- const std::unique_ptr< const Diagnostic > m_DistanceDiagnostic

6.47.1 Constructor & Destructor Documentation

6.47.1.1 SquareSubdivisionMeshV2() [1/2]

```
SquareSubdivisionMeshV2::SquareSubdivisionMeshV2 ()  [delete]
```

6.47.1.2 SquareSubdivisionMeshV2() [2/2]

```
SquareSubdivisionMeshV2::SquareSubdivisionMeshV2 (
             largecounter maxPixels,
             largecounter initialPixels,
             int maxSubdivide,
             largecounter iterationPixels,
             bool initialSubToFinal,
             DiagBitflag valdiag)  [inline]
```

6.47.2 Member Function Documentation

6.47.2.1 EndCurrentLoop()

```
void SquareSubdivisionMeshV2::EndCurrentLoop ()  [final], [virtual]
```

Implements Mesh.

6.47.2.2 ExpInt()

```
pixelcoord SquareSubdivisionMeshV2::ExpInt (
             int base,
             int exp) const  [private]
```

6.47.2.3 GeodesicFinished()

```
void SquareSubdivisionMeshV2::GeodesicFinished (
             largecounter index,
             std::vector< real > finalValues)  [final], [virtual]
```

Implements Mesh.

6.47.2.4 getCurNrGeodesics()

```
largecounter SquareSubdivisionMeshV2::getCurNrGeodesics () const  [final], [virtual]
```

SquareSubdivisionMeshV2 functions.

Implements Mesh.

6.47.2.5 GetDown()

```
SquareSubdivisionMeshV2::PixelInfo * SquareSubdivisionMeshV2::GetDown (
             PixelInfo * p,
             int subdiv) const  [private]
```

6.47.2.6 getFullDescriptionStr()

```
std::string SquareSubdivisionMeshV2::getFullDescriptionStr () const  [final], [virtual]
```

Mesh (abstract base class) functions.

Reimplemented from Mesh.

6.47.2.7 GetLeft()

```
SquareSubdivisionMeshV2::PixelInfo * SquareSubdivisionMeshV2::GetLeft (
            PixelInfo * p,
            int subdiv) const  [private]
```

6.47.2.8 getNewInitConds()

```
void SquareSubdivisionMeshV2::getNewInitConds (
            largecounter index,
            ScreenPoint & newunitpoint,
            ScreenIndex & newscreenindex) const  [final], [virtual]
```

Implements Mesh.

6.47.2.9 GetRight()

```
SquareSubdivisionMeshV2::PixelInfo * SquareSubdivisionMeshV2::GetRight (
            PixelInfo * p,
            int subdiv) const  [private]
```

6.47.2.10 GetUp()

```
SquareSubdivisionMeshV2::PixelInfo * SquareSubdivisionMeshV2::GetUp (
            PixelInfo * p,
            int subdiv) const  [private]
```

6.47.2.11 InitializeFirstGrid()

```
void SquareSubdivisionMeshV2::InitializeFirstGrid ()  [private]
```

6.47.2.12 IsFinished()

```
bool SquareSubdivisionMeshV2::IsFinished () const  [final], [virtual]
```

Implements Mesh.

6.47.2.13 SubdivideAndQueue()

```
void SquareSubdivisionMeshV2::SubdivideAndQueue (
             largecounter ind)   [private]
```

6.47.2.14 UpdateAllWeights()

```
void SquareSubdivisionMeshV2::UpdateAllWeights ()  [private]
```

6.47.3 Member Data Documentation

6.47.3.1 m_ActivePixels

```
std::vector<PixelInfo *> SquareSubdivisionMeshV2::m_ActivePixels {}  [private]
```

6.47.3.2 m_AllPixels

```
std::forward_list<std::unique_ptr<PixelInfo> > SquareSubdivisionMeshV2::m_AllPixels {}  [private]
```

6.47.3.3 m_CurrentPixelQueue

```
std::vector<PixelInfo *> SquareSubdivisionMeshV2::m_CurrentPixelQueue {}  [private]
```

6.47.3.4 m_CurrentPixelQueueDone

```
std::vector<bool> SquareSubdivisionMeshV2::m_CurrentPixelQueueDone {}  [private]
```

6.47.3.5 m_CurrentPixelUpdating

```
std::vector<PixelInfo *> SquareSubdivisionMeshV2::m_CurrentPixelUpdating {}  [private]
```

6.47.3.6 m_InfinitePixels

```
const bool SquareSubdivisionMeshV2::m_InfinitePixels  [private]
```

6.47.3.7 m_InitialPixels

```
const largecounter SquareSubdivisionMeshV2::m_InitialPixels  [private]
```

6.47.3.8 m_InitialSubDivideToFinal

```
const bool SquareSubdivisionMeshV2::m_InitialSubDivideToFinal  [private]
```

6.47.3.9 m_IterationPixels

```
const largecounter SquareSubdivisionMeshV2::m_IterationPixels [private]
```

6.47.3.10 m_MaxPixels

```
const largecounter SquareSubdivisionMeshV2::m_MaxPixels [private]
```

6.47.3.11 m_MaxSubdivide

```
const int SquareSubdivisionMeshV2::m_MaxSubdivide [private]
```

6.47.3.12 m_PixelsIntegrated

```
largecounter SquareSubdivisionMeshV2::m_PixelsIntegrated {0} [private]
```

6.47.3.13 m_PixelsLeft

```
largecounter SquareSubdivisionMeshV2::m_PixelsLeft [private]
```

6.47.3.14 m_RowColumnSize

```
const pixelcoord SquareSubdivisionMeshV2::m_RowColumnSize [private]
```

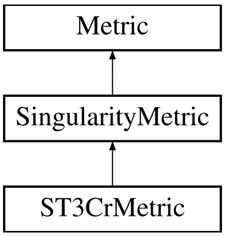
The documentation for this class was generated from the following files:

- FOORT/src/Mesh.h
- FOORT/src/Mesh.cpp

6.48 ST3CrMetric Class Reference

```
#include <Metric.h>
```

Inheritance diagram for ST3CrMetric:



Public Member Functions

- ST3CrMetric (real P, real q0, real lambda, bool rlogscale=false)

  ST3CrMetric functions.
- TwoIndex getMetric_dd (const Point &p) const final
- TwoIndex getMetric_uu (const Point &p) const final
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from SingularityMetric

- SingularityMetric (std::vector< Singularity > thesings, bool rLogScale)

  SingularityMetric functions.
- std::vector< Singularity > getSingularities () const

Public Member Functions inherited from Metric

- virtual ∼Metric ()=default
- Metric (bool rlogscale=false)

  Metric (abstract base class) functions.
- virtual ThreeIndex getChristoffel_udd (const Point &p) const
- virtual FourIndex getRiemann_uddd (const Point &p) const
- virtual real getKretschmann (const Point &p) const
- bool getrLogScale () const

Private Member Functions

- real get_omega (real r, real theta, real l) const
- real f_phi (real phi, real r, real theta, real l, real R) const
- real f_om_phi (real phi, real r, real theta, real l, real R) const

Private Attributes

- const real m_P
- const real m_q0
- const real m_lambda

Additional Inherited Members

Protected Attributes inherited from SingularityMetric

- const std::vector< Singularity > m_AllSingularities

Protected Attributes inherited from Metric

- std::vector< int > m_Symmetries {}
- const bool m_rLogScale

6.48.1 Constructor & Destructor Documentation

6.48.1.1 ST3CrMetric()

```
ST3CrMetric::ST3CrMetric (
            real P,
            real q0,
            real lambda,
            bool rlogscale = false)
```

ST3CrMetric functions.

6.48.2 Member Function Documentation

6.48.2.1 f_om_phi()

```
real ST3CrMetric::f_om_phi (
            real phi,
            real r,
            real theta,
            real l,
            real R) const  [private]
```

6.48.2.2 f_phi()

```
real ST3CrMetric::f_phi (
            real phi,
            real r,
            real theta,
            real l,
            real R) const  [private]
```

6.48.2.3 get_omega()

```
real ST3CrMetric::get_omega (
            real r,
            real theta,
            real l) const  [private]
```

6.48.2.4 getFullDescriptionStr()

```
std::string ST3CrMetric::getFullDescriptionStr () const  [final], [virtual]
```

Reimplemented from Metric.

6.48.2.5 getMetric_dd()

```
TwoIndex ST3CrMetric::getMetric_dd (
            const Point & p) const  [final], [virtual]
```

Implements Metric.

6.48.2.6 getMetric_uu()

```
TwoIndex ST3CrMetric::getMetric_uu (
             const Point & p) const  [final], [virtual]
```

Implements Metric.

6.48.3 Member Data Documentation

6.48.3.1 m_lambda

```
const real ST3CrMetric::m_lambda  [private]
```

6.48.3.2 m_P

```
const real ST3CrMetric::m_P  [private]
```

6.48.3.3 m_q0

```
const real ST3CrMetric::m_q0  [private]
```

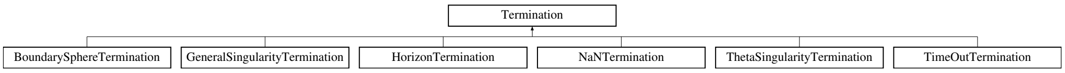
The documentation for this class was generated from the following files:

- FOORT/src/Metric.h
- FOORT/src/Metric.cpp

6.49 Termination Class Reference

```
#include <Terminations.h>
```

Inheritance diagram for Termination:


Termination
BoundarySphereTermination
GeneralSingularityTermination
HorizonTermination
NaNTermination
ThetaSingularityTermination
TimeOutTermination

Public Member Functions

- Termination ()=delete
- Termination (Geodesic ∗const theGeodesic)
- virtual void Reset ()

  Termination (abstract base class) functions.
- virtual ∼Termination ()=default
- virtual Term CheckTermination ()=0
- virtual std::string getFullDescriptionStr () const =0

Protected Member Functions

- bool DecideUpdate (largecounter UpdateNSteps)

Protected Attributes

- Geodesic ∗const m_OwnerGeodesic
- largecounter m_StepsSinceUpdated {}

6.49.1 Constructor & Destructor Documentation

6.49.1.1 Termination() [1/2]

```
Termination::Termination ()  [delete]
```

6.49.1.2 Termination() [2/2]

```
Termination::Termination (
            Geodesic *const theGeodesic)  [inline]
```

6.49.1.3 ∼Termination()

```
virtual Termination::~Termination ()  [virtual], [default]
```

6.49.2 Member Function Documentation

6.49.2.1 CheckTermination()

```
virtual Term Termination::CheckTermination ()  [pure virtual]
```

Implemented in BoundarySphereTermination, GeneralSingularityTermination, HorizonTermination, NaNTermination, ThetaSingularityTermination, and TimeOutTermination.

6.49.2.2 DecideUpdate()

```
bool Termination::DecideUpdate (
            largecounter UpdateNSteps)  [protected]
```

6.49.2.3 getFullDescriptionStr()

```
virtual std::string Termination::getFullDescriptionStr () const  [pure virtual]
```

Implemented in BoundarySphereTermination, GeneralSingularityTermination, HorizonTermination, NaNTermination, ThetaSingularityTermination, and TimeOutTermination.

6.49.2.4 Reset()

```
void Termination::Reset ()  [virtual]
```

Termination (abstract base class) functions.

Reimplemented in TimeOutTermination.

6.49.3 Member Data Documentation

6.49.3.1 m_OwnerGeodesic

```
Geodesic* const Termination::m_OwnerGeodesic  [protected]
```

6.49.3.2 m_StepsSinceUpdated

```
largecounter Termination::m_StepsSinceUpdated {}  [protected]
```

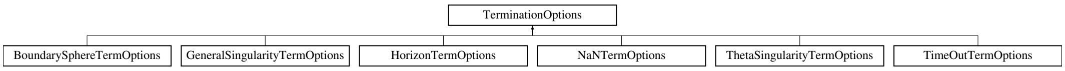
The documentation for this class was generated from the following files:

- FOORT/src/Terminations.h
- FOORT/src/Terminations.cpp

6.50 TerminationOptions Struct Reference

```
#include <Terminations.h>
```

Inheritance diagram for TerminationOptions:


TerminationOptions
BoundarySphereTermOptions
GeneralSingularityTermOptions
HorizonTermOptions
NaNTermOptions
ThetaSingularityTermOptions
TimeOutTermOptions

Public Member Functions

- TerminationOptions (largecounter Nsteps)
- virtual ∼TerminationOptions ()=default

Public Attributes

- const largecounter UpdateEveryNSteps

6.50.1 Constructor & Destructor Documentation

6.50.1.1 TerminationOptions()

```
TerminationOptions::TerminationOptions (
             largecounter Nsteps)  [inline]
```

6.50.1.2 ∼TerminationOptions()

```
virtual TerminationOptions::∼TerminationOptions ()  [virtual], [default]
```

6.50.2 Member Data Documentation

6.50.2.1 UpdateEveryNSteps

```
const largecounter TerminationOptions::UpdateEveryNSteps
```

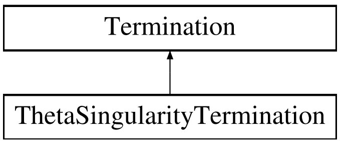
The documentation for this struct was generated from the following file:

- FOORT/src/Terminations.h

6.51 ThetaSingularityTermination Class Reference

```
#include <Terminations.h>
```

Inheritance diagram for ThetaSingularityTermination:



Public Member Functions

- ThetaSingularityTermination (Geodesic ∗const theGeodesic)
- Term CheckTermination () final

  ThetaSingularityTermination functions.
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from Termination

- Termination ()=delete
- Termination (Geodesic ∗const theGeodesic)
- virtual void Reset ()

  Termination (abstract base class) functions.
- virtual ∼Termination ()=default

Static Public Attributes

- static std::unique_ptr< ThetaSingularityTermOptions > TermOptions

Additional Inherited Members

Protected Member Functions inherited from Termination

- bool DecideUpdate (largecounter UpdateNSteps)

Protected Attributes inherited from Termination

- Geodesic ∗const m_OwnerGeodesic
- largecounter m_StepsSinceUpdated {}

6.51.1 Constructor & Destructor Documentation

6.51.1.1 ThetaSingularityTermination()

```
ThetaSingularityTermination::ThetaSingularityTermination (
            Geodesic ∗const theGeodesic)  [inline]
```

6.51.2 Member Function Documentation

6.51.2.1 CheckTermination()

```
Term ThetaSingularityTermination::CheckTermination ()  [final], [virtual]
```

ThetaSingularityTermination functions.

Implements Termination.

6.51.2.2 getFullDescriptionStr()

```
std::string ThetaSingularityTermination::getFullDescriptionStr () const  [final], [virtual]
```

Implements Termination.

6.51.3 Member Data Documentation

6.51.3.1 TermOptions

```
std::unique_ptr< ThetaSingularityTermOptions > ThetaSingularityTermination::TermOptions  [static]
```

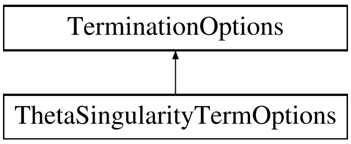
The documentation for this class was generated from the following files:

- FOORT/src/Terminations.h
- FOORT/src/Config.cpp
- FOORT/src/Terminations.cpp

6.52 ThetaSingularityTermOptions Struct Reference

```
#include <Terminations.h>
```

Inheritance diagram for ThetaSingularityTermOptions:



Public Member Functions

- ThetaSingularityTermOptions (real epsilon, largecounter Nsteps)

Public Member Functions inherited from TerminationOptions

- TerminationOptions (largecounter Nsteps)
- virtual ∼TerminationOptions ()=default

Public Attributes

- const real ThetaSingEpsilon

Public Attributes inherited from TerminationOptions

- const largecounter UpdateEveryNSteps

6.52.1 Constructor & Destructor Documentation

6.52.1.1 ThetaSingularityTermOptions()

```
ThetaSingularityTermOptions::ThetaSingularityTermOptions (
            real epsilon,
            largecounter Nsteps)  [inline]
```

6.52.2 Member Data Documentation

6.52.2.1 ThetaSingEpsilon

```
const real ThetaSingularityTermOptions::ThetaSingEpsilon
```

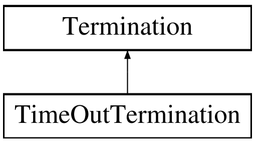
The documentation for this struct was generated from the following file:

- FOORT/src/Terminations.h

6.53 TimeOutTermination Class Reference

```
#include <Terminations.h>
```

Inheritance diagram for TimeOutTermination:



Public Member Functions

- TimeOutTermination (Geodesic ∗const theGeodesic)
- void Reset () final

  TimeOutTermination functions.
- Term CheckTermination () final
- std::string getFullDescriptionStr () const final

Public Member Functions inherited from Termination

- Termination ()=delete
- Termination (Geodesic ∗const theGeodesic)
- virtual ∼Termination ()=default

Static Public Attributes

- static std::unique_ptr< TimeOutTermOptions > TermOptions

Private Attributes

- largecounter m_CurNrSteps {0}

Additional Inherited Members

Protected Member Functions inherited from Termination

- bool DecideUpdate (largecounter UpdateNSteps)

Protected Attributes inherited from Termination

- Geodesic ∗const m_OwnerGeodesic
- largecounter m_StepsSinceUpdated {}

6.53.1 Constructor & Destructor Documentation

6.53.1.1 TimeOutTermination()

```
TimeOutTermination::TimeOutTermination (
            Geodesic *const theGeodesic)  [inline]
```

6.53.2 Member Function Documentation

6.53.2.1 CheckTermination()

```
Term TimeOutTermination::CheckTermination ()  [final], [virtual]
```

Implements Termination.

6.53.2.2 getFullDescriptionStr()

```
std::string TimeOutTermination::getFullDescriptionStr () const  [final], [virtual]
```

Implements Termination.

6.53.2.3 Reset()

```
void TimeOutTermination::Reset ()  [final], [virtual]
```

TimeOutTermination functions.

Reimplemented from Termination.

6.53.3 Member Data Documentation

6.53.3.1 m_CurNrSteps

```
largecounter TimeOutTermination::m_CurNrSteps {0}  [private]
```

6.53.3.2 TermOptions

```
std::unique_ptr< TimeOutTermOptions > TimeOutTermination::TermOptions  [static]
```

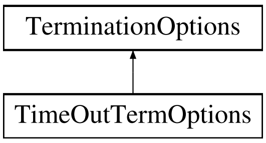
The documentation for this class was generated from the following files:

- FOORT/src/Terminations.h
- FOORT/src/Config.cpp
- FOORT/src/Terminations.cpp

6.54 TimeOutTermOptions Struct Reference

```
#include <Terminations.h>
```

Inheritance diagram for TimeOutTermOptions:



Public Member Functions

- TimeOutTermOptions (largecounter MaxStepsAllowed, largecounter Nsteps)

Public Member Functions inherited from TerminationOptions

- TerminationOptions (largecounter Nsteps)
- virtual ∼TerminationOptions ()=default

Public Attributes

- const largecounter MaxSteps

Public Attributes inherited from TerminationOptions

- const largecounter UpdateEveryNSteps

6.54.1 Constructor & Destructor Documentation

6.54.1.1 TimeOutTermOptions()

```
TimeOutTermOptions::TimeOutTermOptions (
            largecounter MaxStepsAllowed,
            largecounter Nsteps)  [inline]
```

6.54.2 Member Data Documentation

6.54.2.1 MaxSteps

```
const largecounter TimeOutTermOptions::MaxSteps
```

The documentation for this struct was generated from the following file:

- FOORT/src/Terminations.h

6.55 Utilities::Timer Class Reference

```
#include <Utilities.h>
```

Public Member Functions

- void reset ()

  Utilities::Timer functions.
- double elapsed () const

Private Types

- using Clock = std::chrono::steady_clock
- using Second = std::chrono::duration<double, std::ratio<1>>

Private Attributes

- std::chrono::time_point< Clock > m_beg {Clock::now()}

6.55.1 Member Typedef Documentation

6.55.1.1 Clock

```
using Utilities::Timer::Clock = std::chrono::steady_clock  [private]
```

6.55.1.2 Second

```
using Utilities::Timer::Second = std::chrono::duration<double, std::ratio<1>>  [private]
```

6.55.2 Member Function Documentation

6.55.2.1 elapsed()

```
double Utilities::Timer::elapsed () const
```

6.55.2.2 reset()

```
void Utilities::Timer::reset ()
```

Utilities::Timer functions.

6.55.3 Member Data Documentation

6.55.3.1 m_beg

```
std::chrono::time_point<Clock> Utilities::Timer::m_beg {Clock::now()}  [private]
```

The documentation for this class was generated from the following files:

- FOORT/src/Utilities.h
- FOORT/src/Utilities.cpp

6.56 UpdateFrequency Struct Reference

```
#include <Diagnostics.h>
```

Public Attributes

- largecounter UpdateNSteps {0}
- bool UpdateStart {false}
- bool UpdateFinish {false}

6.56.1 Member Data Documentation

6.56.1.1 UpdateFinish

```
bool UpdateFrequency::UpdateFinish {false}
```

6.56.1.2 UpdateNSteps

```
largecounter UpdateFrequency::UpdateNSteps {0}
```

6.56.1.3 UpdateStart

```
bool UpdateFrequency::UpdateStart {false}
```

The documentation for this struct was generated from the following file:

- FOORT/src/Diagnostics.h

6.57 ViewScreen Class Reference

```
#include <ViewScreen.h>
```

Public Member Functions

- ViewScreen ()=delete
- ViewScreen (Point pos, OneIndex dir, ScreenPoint screensize, ScreenPoint screencenter, std::unique_ptr< Mesh > theMesh, const Metric ∗const theMetric, GeodesicType thegeodtype=GeodesicType::Null)
- void SetNewInitialConditions (largecounter index, Point &pos, OneIndex &vel, ScreenIndex &scrIndex) const
- bool IsFinished () const
- largecounter getCurNrGeodesics () const
- void EndCurrentLoop ()
- void GeodesicFinished (largecounter index, std::vector< real > finalValues)
- std::string getFullDescriptionStr () const

Private Member Functions

- void ConstructVielbein ()

  ViewScreen functions.

Private Attributes

- TwoIndex m_Metric_dd {}
- TwoIndex m_Vielbein {}
- const Point m_Pos
- const OneIndex m_Direction
- const ScreenPoint m_ScreenSize
- const ScreenPoint m_ScreenCenter
- const bool m_rLogScale
- const Metric ∗const m_theMetric
- const GeodesicType m_GeodType {GeodesicType::Null}
- const std::unique_ptr< Mesh > m_theMesh

6.57.1 Constructor & Destructor Documentation

6.57.1.1 ViewScreen() [1/2]

```
ViewScreen::ViewScreen ()  [delete]
```

6.57.1.2 ViewScreen() [2/2]

```
ViewScreen::ViewScreen (
            Point pos,
            OneIndex dir,
            ScreenPoint screensize,
            ScreenPoint screencenter,
            std::unique_ptr< Mesh > theMesh,
            const Metric ∗const theMetric,
            GeodesicType thegeodtype = GeodesicType::Null)  [inline]
```

6.57.2 Member Function Documentation

6.57.2.1 ConstructVielbein()

```
void ViewScreen::ConstructVielbein ()  [private]
```

ViewScreen functions.

6.57.2.2 EndCurrentLoop()

```
void ViewScreen::EndCurrentLoop ()
```

6.57.2.3 GeodesicFinished()

```
void ViewScreen::GeodesicFinished (
             largecounter index,
             std::vector< real > finalValues)
```

6.57.2.4 getCurNrGeodesics()

```
largecounter ViewScreen::getCurNrGeodesics () const
```

6.57.2.5 getFullDescriptionStr()

```
std::string ViewScreen::getFullDescriptionStr () const
```

6.57.2.6 IsFinished()

```
bool ViewScreen::IsFinished () const
```

6.57.2.7 SetNewInitialConditions()

```
void ViewScreen::SetNewInitialConditions (
             largecounter index,
             Point & pos,
             OneIndex & vel,
             ScreenIndex & scrIndex) const
```

6.57.3 Member Data Documentation

6.57.3.1 m_Direction

```
const OneIndex ViewScreen::m_Direction  [private]
```

6.57.3.2 m_GeodType

```
const GeodesicType ViewScreen::m_GeodType {GeodesicType::Null}  [private]
```

6.57.3.3 m_Metric_dd

```
TwoIndex ViewScreen::m_Metric_dd {}  [private]
```

6.57.3.4 m_Pos

```
const Point ViewScreen::m_Pos  [private]
```

6.57.3.5 m_rLogScale

```
const bool ViewScreen::m_rLogScale  [private]
```

6.57.3.6 m_ScreenCenter

```
const ScreenPoint ViewScreen::m_ScreenCenter  [private]
```

6.57.3.7 m_ScreenSize

```
const ScreenPoint ViewScreen::m_ScreenSize  [private]
```

6.57.3.8 m_theMesh

```
const std::unique_ptr<Mesh> ViewScreen::m_theMesh  [private]
```

6.57.3.9 m_theMetric

```
const Metric* const ViewScreen::m_theMetric  [private]
```

6.57.3.10 m_Vielbein

```
TwoIndex ViewScreen::m_Vielbein {}  [private]
```

The documentation for this class was generated from the following files:

- FOORT/src/ViewScreen.h
- FOORT/src/ViewScreen.cpp

Chapter 7

File Documentation

7.1 FOORT/src/Config.cpp File Reference

```
#include "Config.h"
#include "Utilities.h"
#include <algorithm>
#include <cctype>
#include <utility>
```

7.2 FOORT/src/Config.h File Reference

```
#include "Geometry.h"
#include "Metric.h"
#include "Diagnostics.h"
#include "Terminations.h"
#include "ViewScreen.h"
#include "Geodesic.h"
#include "Integrators.h"
#include <memory>
#include <string>
#include <exception>
#include "ConfigReader.h"
```

Namespaces

- namespace Config

Macros

- #define CONFIGURATION_MODE

Typedefs

- using Config::ConfigCollection = ConfigReader::ConfigCollection
- using Config::SettingError = std::invalid_argument

Functions

- void Config::InitializeScreenOutput (const ConfigCollection &theCfg)
- std::unique_ptr< Metric > Config::GetMetric (const ConfigCollection &theCfg)

  Config::GetMetric(): Use configuration to create the correct Metric with specified parameters.
- std::unique_ptr< Source > Config::GetSource (const ConfigCollection &theCfg, const Metric ∗const the↩ Metric)

  Config::GetSource(): Use configuration to create the correct Source with specified parameters.
- void Config::InitializeDiagnostics (const ConfigCollection &theCfg, DiagBitflag &alldiags, DiagBitflag &valdiag, const Metric ∗const theMetric)

  Config::InitializeDiagnostics(): Use configuration to set the Diagnostics bitflag appropriately; initialize all DiagnosticOptions for all Diagnostics that are turned on; and set bitflag for diagnostic to be used for coarseness evaluating in Mesh.
- void Config::InitializeTerminations (const ConfigCollection &theCfg, TermBitflag &allterms, const Metric ∗const theMetric)

  Config::InitializeTerminations(): Use configuration to set the Termination bitflag appropriately; and initialize all TerminationOptions for all Terminations that are turned on.
- std::unique_ptr< ViewScreen > Config::GetViewScreen (const ConfigCollection &theCfg, DiagBitflag valdiag, const Metric ∗const theMetric)

  Config::GetViewScreen(): Use configuration to create the ViewScreen object; with options set according to the configuration.
- std::unique_ptr< Mesh > Config::GetMesh (const ConfigCollection &theCfg, DiagBitflag valdiag)

  Config::GetMesh(): Use configuration to create the Mesh object; with options set according to the configuration. Config::GetViewScreen() calls this when creating the ViewScreen object.
- GeodesicIntegratorFunc Config::GetGeodesicIntegrator (const ConfigCollection &theCfg)

  Config::GetGeodesicIntegrator(): Returns a pointer to the integrator function to be used as specified in the configuration file.
- std::unique_ptr< GeodesicOutputHandler > Config::GetOutputHandler (const ConfigCollection &theCfg, DiagBitflag alldiags, DiagBitflag valdiag, std::string FirstLineInfo)

  Config::GetOutputHandler(): Creates the GeodesicOutputHandler object with options specified according to the configuration file, for handling of geodesic outputs.

Variables

- constexpr auto Config::Output_Important_Default = OutputLevel::Level_0_WARNING
- constexpr auto Config::Output_Other_Default = OutputLevel::Level_1_PROC

7.2.1 Macro Definition Documentation

7.2.1.1 CONFIGURATION_MODE

```
#define CONFIGURATION_MODE
```

7.3 Config.h

Go to the documentation of this file.

```
#ifndef _FOORT_CONFIG_H
#define _FOORT_CONFIG_H


// COMMENT ONLY THIS LINE OUT TO BE IN PRECOMPILED OPTIONS MODE
#define CONFIGURATION_MODE

// We need essentially all of the possible different objects as configuration functions initialize all of them
#include "Geometry.h"
#include "Metric.h"
#include "Diagnostics.h"
#include "Terminations.h"
#include "ViewScreen.h"
#include "Geodesic.h"
#include "Integrators.h"

// The entire configuration namespace and its functions are only defined in CONFIGURATION MODE!
#ifdef CONFIGURATION_MODE

#include <memory> // std::unique_ptr
#include <string> // std::string

#include <exception> // needed to define our own configuration error

#include "ConfigReader.h"

// Namespace for all configuration functions that initialize objects based on configuration file
namespace Config
{
    // Output level for important missing information that will default
    // (e.g. no metric selected, no diagnostics selected, ...)
    constexpr auto Output_Important_Default = OutputLevel::Level_0_WARNING;
    // Output level for less important information that will default
    // (e.g. Kerr metric a parameter not specified)
    constexpr auto Output_Other_Default = OutputLevel::Level_1_PROC;

    // Total configuration object, as loaded from configuration file
    using ConfigCollection = ConfigReader::ConfigCollection;

    // An exception to throw whenever an important setting is not found
    // Always should be caught and then reverted to default settings
    using SettingError = std::invalid_argument;


    // Use configuration to initialize the screen output level
    void InitializeScreenOutput(const ConfigCollection &theCfg);

    // Use configuration to create the correct Metric with specified parameters
    std::unique_ptr<Metric> GetMetric(const ConfigCollection &theCfg);

    // Use configuration to create the correct Source with specified parameters
    std::unique_ptr<Source> GetSource(const ConfigCollection &theCfg, const Metric *const theMetric);

    // Use configuration to set the Diagnostics bitflag appropriately;
    // initialize all DiagnosticOptions for all Diagnostics that are turned on;
    // and set bitflag for diagnostic to be used for coarseness evaluating in Mesh
    void InitializeDiagnostics(const ConfigCollection &theCfg, DiagBitflag &alldiags, DiagBitflag &valdiag, const Metric *const theMetric);

    // Use configuration to set the Terminations bitflag appropriately;
    // initialize all TerminationOptions for all Terminations that are turned on;
    void InitializeTerminations(const ConfigCollection &theCfg, TermBitflag &allterms, const Metric *const theMetric);

    // Use configuration to create ViewScreen appropriately
    std::unique_ptr<ViewScreen> GetViewScreen(const ConfigCollection &theCfg, DiagBitflag valdiag, const Metric *const theMetric);
    // GetViewScreen calls GetMesh to create the correct Mesh
    std::unique_ptr<Mesh> GetMesh(const ConfigCollection &theCfg, DiagBitflag valdiag);

    // Use configuration to return a pointer to the correct integration function to use
    GeodesicIntegratorFunc GetGeodesicIntegrator(const ConfigCollection &theCfg);

    // Use configuration to initialize the output handler
    std::unique_ptr<GeodesicOutputHandler> GetOutputHandler(const ConfigCollection &theCfg,
                                                            DiagBitflag alldiags, DiagBitflag valdiag, std::string FirstLineInfo);

} // end namespace Config

#endif // CONFIGURATION_MODE
```

```
00088
00089 #endif
```

7.4 FOORT/src/ConfigReader.cpp File Reference

```
#include "ConfigReader.h"
#include <ios>
#include <type_traits>
```

7.5 FOORT/src/ConfigReader.h File Reference

```
#include <string_view>
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
#include <variant>
#include <stdexcept>
#include <limits>
#include <memory>
```

Classes

- class ConfigReader::ConfigReaderException
- class ConfigReader::ConfigCollection
- struct ConfigReader::ConfigCollection::ConfigSetting

Namespaces

- namespace ConfigReader

Variables

- const long long ConfigReader::MaxStreamSize {std::numeric_limits<std::streamsize>::max()}

7.6 ConfigReader.h

Go to the documentation of this file.

```
#ifndef _CONFIGREADER_H
#define _CONFIGREADER_H

#include <string_view> // std::string_view
#include <string>      // std::string
#include <iostream>    // needed for file and console output
#include <fstream>     // needed for file input
#include <vector>      // needed to create vectors of Settings
#include <variant>     // needed for std::variant
#include <stdexcept>   // needed for exceptions
#include <limits>      // for std::numeric_limits
#include <memory>      // for std::unique_ptr

// Namespace to put all ConfigReader objects
namespace ConfigReader
{
    // Shorthand for the maximum number of characters we can ignore in a stream
    const long long MaxStreamSize{std::numeric_limits<std::streamsize>::max()};

    // Exception that is thrown if anything bad happens when reading in configuration file
    class ConfigReaderException : public std::runtime_error
    {
    public:
        ConfigReaderException(const std::string &error, std::vector<int> settingtrace = {})
            : std::runtime_error{error.c_str()}, m_settingtrace{settingtrace}
        {
        }
        // no need to override what() since we can just use std::runtime_error::what()

        std::vector<int> trace() const
        {
            return m_settingtrace;
        };

    private:
        std::vector<int> m_settingtrace;
    };

    // A collection of configuration settings
    class ConfigCollection
    {
    public:
        // Returns true if the collection contains a setting with the given name
        bool Exists(std::string_view SettingName) const;
        // Returns total number of settings in this collection
        int NrSettings() const;

        // Returns true if the setting with the given name (resp. index) is a collection of subsettings
        // (Returns false if this setting name does not exist, or index is out of bounds)
        bool IsCollection(std::string_view SettingName) const;
        bool IsCollection(int SettingIndex) const;
        // If CollectionName (resp. CollectionIndex) is the name (resp. index)
        // of a setting that is a subcollection, then this returns
        // a const reference to this subcollection; otherwise throws ConfigReaderException
        const ConfigCollection &operator[](std::string_view CollectionName) const;
        const ConfigCollection &operator[](int CollectionIndex) const;

        // Generic functions that look up the value of a given setting
        // and put it in the output variable
        // Returns true if successful, false if unsuccesful
        // (either due to type mismatch of output and setting value, or if setting doesn’t exist)
        // Templated functions implemented below in .h file!
        template <class OutputType>
        bool LookupValue(std::string_view SettingName, OutputType &theOutput) const;
        template <class OutputType>
        bool LookupValue(int SettingIndex, OutputType &theOutput) const;

        // Specific functions for looking up signed integral types
        // These will first see if the setting value is of the given type, then
        // they will try smaller types and then recast to the larger type
        // (so e.g. for long long, first long long is tried, then long, then int)
        bool LookupValueInteger(std::string_view SettingName, int &theOutput) const;
        bool LookupValueInteger(std::string_view SettingName, long &theOutput) const;
        bool LookupValueInteger(std::string_view SettingName, long long &theOutput) const;
        // Specific functions for looking up unsigned integral types
        // These will first test if the setting value is of the signed case, if it is and it is >=0 then it always fits in
        // the unsigned version; then it will test larger (signed) types and see if the result still fits in the unsigned version
        bool LookupValueInteger(std::string_view SettingName, unsigned int &theOutput) const;
        bool LookupValueInteger(std::string_view SettingName, unsigned long &theOutput) const;
        bool LookupValueInteger(std::string_view SettingName, unsigned long long &theOutput) const;
        // Templated functions implemented below in .h file!
        template <class OutputType>
        bool LookupValueInteger(int SettingIndex, OutputType &theOutput) const;

        void DisplayCollection(std::ostream &OutputStream, int Indent = 0) const;

        // this overwrites the current collection with the contents of the config file
        // Will throw ConfigReaderException if parse/syntax error in configuration file
        // Returns true if successful
        bool ReadFile(const std::string &FileName);

    private:
        // A configuration setting value can be an integral type, bool, double, string, or a (sub)collection of settings
        // Note: when reading in a configuration file, an integral value will always be stored in the smallest possible type!
        using ConfigSettingValue = std::variant<bool,
                                                int, long, long long,
                                                double,
                                                std::string,
                                                std::unique_ptr<ConfigCollection»;
        // A setting is a name and setting value
        struct ConfigSetting
        {
            std::string SettingName;
            ConfigSettingValue SettingValue;
        };
        // The vector of all settings in this collection
        std::vector<ConfigSetting> m_Settings{};

        // Helper function that returns the index of the setting with the given name
        // returns -1 if setting not found
        int GetSettingIndex(std::string_view SettingName) const;

        // Helper functions for DisplayCollection
        // DisplaySetting displays one setting; if it’s a subcollection then it calls
        // DisplayCollection on the subcollection
        void DisplaySetting(std::ostream &OutputStream, int SettingIndex, int Indent) const;
        // Output given number of tabs
        void DisplayTabs(std::ostream &OutputStream, int NrTabs) const;

        // Helper functions for ReadFile()
        // These all take an ifstream at a given location and will read in one specific object
        //
        // Read in entire collection (including subcollections)
        // Pre: stream is positioned right after { (or at beginning of file)
        // Post: stream is positioned right after } (or at EOF)
        void ReadCollection(std::ifstream &InputFile);
        // Read in a setting name
        // Pre: next non-whitespace character in stream is beginning of name
        // Post: stream has just read in all characters of name (but no more);
        // returns "" if there is no more setting to read in the current collection, in this case
        // ’}’ has been read in (for subcollections)
        void ReadSettingName(std::ifstream &InputFile, std::string &theName);
        // Read in one specific character only (not ’/’!)
        // Pre: next non-whitespace character in stream is this character
        // Post: stream is just after character
        void ReadSettingSpecificChar(std::ifstream &InputFile, char theChar) const;
        // Read in setting value (could be subcollection)
        // Pre: next non-whitespace character in stream is beginning of value
        // (can be ’{’ for subcollection, digit for number, ’"’ for string, or ’t’/’f’ for boolean)
        // Post: Value has entirely been read in (i.e. next character should be ’;’)
        void ReadSettingValue(std::ifstream &InputFile, ConfigSettingValue &theValue);
    };

} // end namespace declarations


template <class OutputType>
bool ConfigReader::ConfigCollection::LookupValue(int SettingIndex, OutputType &theOutput) const
{
    // Get the setting value, if it is indeed of this type
    const OutputType *OutputPointer{std::get_if<OutputType>(&m_Settings[SettingIndex].SettingValue)};
    if (OutputPointer)
    {
        // Success, the setting is indeed of this type; set the output accordingly
        theOutput = *OutputPointer;
        return true;
    }
    return false;
}

template <class OutputType>
bool ConfigReader::ConfigCollection::LookupValue(std::string_view SettingName, OutputType &theOutput) const
{
    // Look up setting index and then defer to index-based implementation, if setting exists
    int SettingIndex{GetSettingIndex(SettingName)};
    if (SettingIndex >= 0)
    {
        return LookupValue(SettingIndex, theOutput);
    }
    return false;
}

// This simply defers to the setting-name-based implementation of the function with given output type
template <class OutputType>
bool ConfigReader::ConfigCollection::LookupValueInteger(int SettingIndex, OutputType &theOutput) const
{
    return LookupValueInteger(m_Settings[SettingIndex].SettingName, theOutput);
}

#endif
```

7.7 FOORT/src/Diagnostics.cpp File Reference

```
#include "Diagnostics.h"
#include "DiagnosticsEmission.h"
#include "Geodesic.h"
#include "InputOutput.h"
#include "Integrators.h"
#include <algorithm>
#include <cmath>
```

Functions

- DiagnosticUniqueVector CreateDiagnosticVector (DiagBitflag diagflags, DiagBitflag valdiag, Geodesic ∗const theGeodesic)

  Diagnostic helper function.

7.7.1 Function Documentation

7.7.1.1 CreateDiagnosticVector()

```
DiagnosticUniqueVector CreateDiagnosticVector (
            DiagBitflag diagflags,
            DiagBitflag valdiag,
            Geodesic ∗const theGeodesic)
```

Diagnostic helper function.

7.8 FOORT/src/Diagnostics.h File Reference

```
#include "Geometry.h"
#include "Metric.h"
#include "InputOutput.h"
#include "DiagnosticsEmission.h"
#include <cstdint>
#include <string>
#include <memory>
#include <vector>
```

Classes

- struct UpdateFrequency
- class Diagnostic
- class FourColorScreenDiagnostic
- class GeodesicPositionDiagnostic
- class EquatorialPassesDiagnostic
- class ClosestRadiusDiagnostic
- class EquatorialEmissionDiagnostic
- struct DiagnosticOptions
- struct GeodesicPositionOptions
- struct EquatorialPassesOptions
- struct ClosestRadiusOptions
- struct EquatorialEmissionOptions

Typedefs

- using DiagBitflag = std::uint16_t
- using DiagnosticUniqueVector = std::vector<std::unique_ptr<Diagnostic>>

Functions

- DiagnosticUniqueVector CreateDiagnosticVector (DiagBitflag diagflags, DiagBitflag valdiag, Geodesic *const theGeodesic)

  Diagnostic helper function.

Variables

- constexpr DiagBitflag Diag_None {0b0000'0000'0000'0000}
- constexpr DiagBitflag Diag_GeodesicPosition {0b0000'0000'0000'0001}
- constexpr DiagBitflag Diag_FourColorScreen {0b0000'0000'0000'0010}
- constexpr DiagBitflag Diag_EquatorialPasses {0b0000'0000'0000'0100}
- constexpr DiagBitflag Diag_ClosestRadius {0b0000'0000'0000'1000}
- constexpr DiagBitflag Diag_EquatorialEmission {0b0000'0000'0001'0000}

7.8.1 Typedef Documentation

7.8.1.1 DiagBitflag

```
using DiagBitflag = std::uint16_t
```

7.8.1.2 DiagnosticUniqueVector

```
using DiagnosticUniqueVector = std::vector<std::unique_ptr<Diagnostic>>
```

7.8.2 Function Documentation

7.8.2.1 CreateDiagnosticVector()

```
DiagnosticUniqueVector CreateDiagnosticVector (
            DiagBitflag diagflags,
            DiagBitflag valdiag,
            Geodesic *const theGeodesic)
```

Diagnostic helper function.

7.8.3 Variable Documentation

7.8.3.1 Diag_ClosestRadius

```
DiagBitflag Diag_ClosestRadius {0b0000'0000'0000'1000}  [constexpr]
```

7.8.3.2 Diag_EquatorialEmission

```
DiagBitflag Diag_EquatorialEmission {0b0000'0000'0001'0000}  [constexpr]
```

7.8.3.3 Diag_EquatorialPasses

```
DiagBitflag Diag_EquatorialPasses {0b0000'0000'0000'0100}  [constexpr]
```

7.8.3.4 Diag_FourColorScreen

```
DiagBitflag Diag_FourColorScreen {0b0000'0000'0000'0010}  [constexpr]
```

7.8.3.5 Diag_GeodesicPosition

```
DiagBitflag Diag_GeodesicPosition {0b0000'0000'0000'0001}  [constexpr]
```

7.8.3.6 Diag_None

```
DiagBitflag Diag_None {0b0000'0000'0000'0000}  [constexpr]
```

7.9 Diagnostics.h

Go to the documentation of this file.

```
#ifndef _FOORT_DIAGNOSTICS_H
#define _FOORT_DIAGNOSTICS_H



#include "Geometry.h"  // for tensors
#include "Metric.h"      // for metric
#include "InputOutput.h" // for ScreenOutput

#include "DiagnosticsEmission.h" // Emission models

#include <cstdint> // for std::uint16_t
#include <string>  // for strings
#include <memory>  // for std::unique_ptr
#include <vector>  // for std::vector

// Forward declaration of Geodesic class needed here, since Diagnostics are passed a pointer to their owner Geodesic
// (note "Geodesic.h" is NOT included to avoid header loop, and we do not need Geodesic member functions here!)
class Geodesic;



// Diagnostic bitflags
// Used for constructing vector of Diagnostics
// Note that this means every diagnostic is either "on" or "off";
// it is not possible to have a Diagnostic "on" more than once
// Note: why not std::bitset<size>? Because in this way we can use expressions with DiagBitflag in conditional expressions,
// e.g. if ( mydiagbitflag & Diag_FourColorScreen)
using DiagBitflag = std::uint16_t;

// Define a bitflag per existing diagnostic
constexpr DiagBitflag Diag_None{0b0000’0000’0000’0000};
constexpr DiagBitflag Diag_GeodesicPosition{0b0000’0000’0000’0001};
constexpr DiagBitflag Diag_FourColorScreen{0b0000’0000’0000’0010};
constexpr DiagBitflag Diag_EquatorialPasses{0b0000’0000’0000’0100};
constexpr DiagBitflag Diag_ClosestRadius{0b0000’0000’0000’1000};
constexpr DiagBitflag Diag_EquatorialEmission{0b0000’0000’0001’0000};

// Add a DiagBitflag for your new diagnostic. Make sure you use a bitflag that has not been used before!
// Sample code:
/*
constexpr DiagBitflag Diag_MyDiag               { 0b0000’0000’0000’1000 };
*/

// This carries the information for a Diagnostic to update itself: if UpdateNsteps > 0, then every so many steps.
// If UpdateNSteps == 0, then it only updates at the start and/or finish of integration if the appropriate bool is set.
struct UpdateFrequency
{
    largecounter UpdateNSteps{0};
    bool UpdateStart{false};
    bool UpdateFinish{false};
};



// Abstract base class for all diagnostics
class Diagnostic
{
public:
    // Constructor must initialize the pointer to its owner Geodesic
    Diagnostic() = delete;
    Diagnostic(Geodesic *const theGeodesic) : m_OwnerGeodesic{theGeodesic}
    {
    }

    // Resets Diagnostic object. This is called when the owner Geodesic is reset in order to start integrating
    // a new geodesic.
    // The base class implementation only resets m_StepsSinceUpdated
    // Descendants can override this if they need to reset additional internal variables
    virtual void Reset();

    // virtual destructor to ensure correct destruction of descendants
    virtual ~Diagnostic() = default;

    // This is the heart of the Diagnostic. It calls the helper function DecideUpdate(), and if this
    // returns true, will update its internal status based on the current (new) state of its owner geodesic.
    virtual void UpdateData() = 0;
    // These functions are for use at the end of integration of a geodesic.
    // getFullData() returns all the data stored in the Diagnostic as a string (for output to file)
    virtual std::string getFullDataStr() const = 0;
    // getFinalDataVal() associates a value to the geodesic corresponding to the final value of this diagnostic
    // (used for determining coarseness of nearby geodesics)
    virtual std::vector<real> getFinalDataVal() const = 0;

    // Function used to determine distance between two values obtained from getFinalDataVal()
    // (for determining coarseness of nearby geodesics)
    // This should return a number >=0
    virtual real FinalDataValDistance(const std::vector<real> &val1, const std::vector<real> &val2) const = 0;

    // Getters for descriptions
    // This returns the name (only) of the Diagnostic, as a string without spaces that will be appended
    // to an output file (e.g. prefix_DiagName.ext). Must be implemented!
    virtual std::string getNameStr() const = 0;
    // This returns the full description of the Diagnostic. Default implementation
    // returns GetDiagNameStr()
    virtual std::string getFullDescriptionStr() const;

protected:
    // The geodesic that owns the Diagnostic (a const pointer to the Geodesic)
    Geodesic *const m_OwnerGeodesic;

    // Helper function to decide if the Diagnostic should indeed update its status, based on
    // its UpdateFrequency struct information (which will come from the Diagnostics's DiagnosticOptions)
    bool DecideUpdate(const UpdateFrequency &myUpdateFrequency);

    // The diagnostic is itself in charge of keeping track of how many steps it has been since it has been updated
    // The Diagnostic's DiagnosticOptions struct tells it how many steps it needs to wait between updates
    largecounter m_StepsSinceUpdated{};
};

// Owner vector of derived Diagnostics classes
using DiagnosticUniqueVector = std::vector<std::unique_ptr<Diagnostic»;

// Helper to create a new vector of Diagnostic options, based on the bitflag
// The first diagnostic is the value diagnostic
DiagnosticUniqueVector CreateDiagnosticVector(DiagBitflag diagflags, DiagBitflag valdiag, Geodesic *const theGeodesic);


// The four color screen: associates one of four colors based on the quadrant that the geodesic finishes in
// Will ONLY return a color if the geodesic indeed finishes because it passes through the boundary sphere!
class FourColorScreenDiagnostic final : public Diagnostic
{
public:
    // Basic constructor only passes on Geodesic pointer to base class constructor
    FourColorScreenDiagnostic(Geodesic *const theGeodesic) : Diagnostic(theGeodesic) {}

    // Reset needs to be overridden in order to also reset m_quadrant
    void Reset() final;

    // Sets the quadrant to the appropriate value, IF the boundary sphere has been reached
    void UpdateData() override;

    // Both of these output functions simply returns the quadrant number associated with the geodesic's end position
    std::string getFullDataStr() const final;
    std::vector<real> getFinalDataVal() const final;

    // Discrete metric for distance: returns 0 if the quadrants are the same, 1 if they are not
    real FinalDataValDistance(const std::vector<real> &val1, const std::vector<real> &val2) const final;

    // Description string getters
    std::string getNameStr() const final;
    std::string getFullDescriptionStr() const final;

    // FourColorScreen does not need any (static) options!

private:
    // Note initialization to 0; this means the default value returned will be 0
    // (e.g. if Term::BoundarySphere is not reached)
    int m_quadrant{0};
};

// Forward declaration needed before Diagnostic
```

```
struct GeodesicPositionOptions;
// Geodesic position tracker
class GeodesicPositionDiagnostic final : public Diagnostic
{
public:
    // Basic constructor only passes on Geodesic pointer to base class constructor
    GeodesicPositionDiagnostic(Geodesic *const theGeodesic) : Diagnostic(theGeodesic) {}

    // Override Reset to also clear m_AllSavedPoints
    void Reset() final;

    // Stores the current position of the geodesic
    void UpdateData() final;

    // This returns as many stored positions as is specified in the options struct
    std::string getFullDataStr() const final;
    // This returns the final (theta,phi) value of the geodesic
    std::vector<real> getFinalDataVal() const final;

    // This determines the angular distance between two geodesic
    // (based on their final angles on the boundary sphere (theta, phi))
    real FinalDataValDistance(const std::vector<real> &val1, const std::vector<real> &val2) const final;

    // Description string getters
    std::string getNameStr() const final;
    std::string getFullDescriptionStr() const final;

    // The options: specifies the update frequency but also how many points we want to output in the end
    static std::unique_ptr<GeodesicPositionOptions> DiagOptions;

private:
    // Keeps track of the points that are saved
    std::vector<Point> m_AllSavedPoints{};
};

// Forward declaration needed before Diagnostic
struct EquatorialPassesOptions;
// Diagnostic for counting number of passes through equatorial plane
class EquatorialPassesDiagnostic : public Diagnostic
{
public:
    // Basic constructor only passes on Geodesic pointer to base class constructor
    EquatorialPassesDiagnostic(Geodesic *const theGeodesic) : Diagnostic(theGeodesic) {}

    // Override Reset to also reset m_EquatPasses and m_PrevTheta
    void Reset() override;

    // Checks to see if we have a new cross over the equatorial plane
    void UpdateData() override;

    // Returns the number of passes over the equatorial plane
    std::string getFullDataStr() const override;
    std::vector<real> getFinalDataVal() const override;

    // Simple absolute value of difference of passes
    real FinalDataValDistance(const std::vector<real> &val1, const std::vector<real> &val2) const override;

    // Description string getters
    std::string getNameStr() const override;
    std::string getFullDescriptionStr() const override;

    // Needs the extra option of a threshold
    static std::unique_ptr<EquatorialPassesOptions> DiagOptions;

protected:
    // Keeps track of how many passes have been made
    int m_EquatPasses{0};

private:
    // Keeps track of the previous theta angle, so that we can compare with current theta angle
    real m_PrevTheta{-1};
};

// Forward declaration needed before Diagnostic
struct ClosestRadiusOptions;
// Diagnostic keeps track of the closes radius that the geodesic passes through
// Note: always keeps track of true radius, not log radius
class ClosestRadiusDiagnostic final : public Diagnostic
{
public:
    // Basic constructor only passes on Geodesic pointer to base class constructor
    ClosestRadiusDiagnostic(Geodesic *const theGeodesic) : Diagnostic(theGeodesic) {}

    // re-initialize closest radius
    void Reset() final;

    // Check to see if we have travelled closer
    void UpdateData() final;

    // Return closest radius travelled
    std::string getFullDataStr() const final;
    std::vector<real> getFinalDataVal() const final;

    // Returns the absolute value of the difference between closest radii
    real FinalDataValDistance(const std::vector<real> &val1, const std::vector<real> &val2) const final;

    // Description string getters
    std::string getNameStr() const final;
    std::string getFullDescriptionStr() const final;

    // Option struct to keep track of RLogScale of Metric
    static std::unique_ptr<ClosestRadiusOptions> DiagOptions;

private:
    // Keeps track of closest radius travelled
    real m_ClosestRadius{-1};
};

// Forward declaration needed before Diagnostic
struct EquatorialEmissionOptions;
// Diagnostic that calculates brightness intensity for the geodesic, based on
// a specified equatorial disc emission model
// (intensity profile and fluid velocity profile, both specified in the options struct)
// Note that EquatorialEmissionDiagnostic inherits from EquatorialPassesDiagnostic,
// since it needs to keep track of when it passes through the equatorial plane.
// As a result it also keeps track of the number of equatorial passes
class EquatorialEmissionDiagnostic final : public EquatorialPassesDiagnostic
{
public:
    // Basic constructor only passes on Geodesic pointer to parent class constructor
    EquatorialEmissionDiagnostic(Geodesic *const theGeodesic) : EquatorialPassesDiagnostic(theGeodesic) {}

    // Reset intensity back to 0.0, and call parent class Reset()
    void Reset() final;

    // Use parent class UpdateData() to decide whether the update the intensity
    // using the specified emission model
    void UpdateData() final;

    // Returns total intensity and total number of equatorial passes (i.e. normal output from EquatorialPassesDiagnostic)
    std::string getFullDataStr() const final;
    std::vector<real> getFinalDataVal() const final;

    // Distance is difference in intensities multiplied by a factor magnifying difference in equatorial passes
    real FinalDataValDistance(const std::vector<real> &val1, const std::vector<real> &val2) const final;

    // Description string getters
    std::string getNameStr() const final;
    std::string getFullDescriptionStr() const final;

    // Option struct to keep track emission model specifics
    static std::unique_ptr<EquatorialEmissionOptions> DiagOptions;

private:
    // Brightness intensity of the geodesic
    real m_Intensity{0.0};
};

// Declare your Diagnostic class here, inheriting from Diagnostic.
// Sample code:
/*
// Forward declaration needed before Diagnostic (if using base class DiagnosticOptions, this is strictly speaking not necessary)
struct DiagnosticOptions;
// class definition
class MyDiagnostic final : public Diagnostic // good practice to make the class final unless descendant classes are possible
{
public:
    // constructor must at least take and pass along the const pointer to the owner Geodesic
    MyDiagnostic(Geodesic* const theGeodesic) : Diagnostic(theGeodesic) {}

    // If you have MyDiagnostic-specific member variables, then they probably need to be reset at the beginning
    // of integration for each new geodesic; in that case, you need to override Reset() to do so.
    // Note: make sure to call the base class implementation Diagnostic::Reset()
    // from within your implementation of MyDiagnostic::Reset(), so that the base class internal variable is also reset!
    void Reset() final;

    // This is the heart of the Diagnostic: here the Diagnostic updates its internal state according to
    // the owner Geodesic’s current state
    void UpdateData() final;

    // This should return the string that is to be outputted to the file as the final output of this Diagnostic
    // for its owner Geodesic
    std::string getFullDataStr() const final;
    // This should return a vector of real numbers that indicates the final "value" that should be associated to
    // the owner Geodesic --- this is then used in FinalDataValDistance() to find "distances" between geodesics.
    std::vector<real> getFinalDataVal() const final;

    // This should return a (positive) distance of two values returned by getFinalDataVal(), indicated the
    // "distance" of two geodesics (this is used for Mesh refinement)
    real FinalDataValDistance(const std::vector<real>& val1, const std::vector<real>& val2) const final;

    // Must implement getNameStr() (and recommended also to implement getFullDescriptionStr)
    // getNameStr() is a simple, short string (without spaces) that will be appended to the file name
    // where this Diagnostic’s output is written. getFullDescriptionStr() is a descriptive string that should list
    // all relevant options set; this is outputted to e.g. the screen at runtime.
    std::string getNameStr() const final;
    std::string getFullDescriptionStr() const final;

    // Diagnostic options: basic DiagnosticOptions only contains UpdateFrequency information,
    // if needed, can use a descendant class with more options (see e.g. GeodesicPositionDiagnostic)
    static std::unique_ptr<DiagnosticOptions> DiagOptions;

private:
    // will probably want some private member variable(s) to keep track of whatever geodesic property is desired
};
*/


// Base class for DiagnosticOptions. Other Diagnostics can inherit from here if they require more options.
struct DiagnosticOptions
{
public:
    // Basic constructor only sets the number of steps between updates
    DiagnosticOptions(UpdateFrequency thefrequency) : theUpdateFrequency{thefrequency}
    {
    }

    virtual ~DiagnosticOptions() = default;

    const UpdateFrequency theUpdateFrequency;
};

// GeodesicPositionDiagnostic needs to keep track of number of steps to output
struct GeodesicPositionOptions : public DiagnosticOptions
{
public:
    GeodesicPositionOptions(largecounter outputsteps, UpdateFrequency thefrequency) : OutputNrSteps{outputsteps},
        DiagnosticOptions(thefrequency)
    {
    }

    const largecounter OutputNrSteps;
};

// EquatorialPassesDiagnostic needs to keep track of the threshold
struct EquatorialPassesOptions : public DiagnosticOptions
{
public:
    EquatorialPassesOptions(real ththreshold, UpdateFrequency thefrequency) : Threshold{ththreshold},
        DiagnosticOptions(thefrequency)
    {
    }

    const real Threshold;
};

```

```
// ClosestRadiusOptions needs to keep track of whether we are using r or log(r) radial coordinate
struct ClosestRadiusOptions : public DiagnosticOptions
{
public:
    ClosestRadiusOptions(bool rlog, UpdateFrequency thefrequency) : RLogScale{rlog},
                                                                     DiagnosticOptions(thefrequency)
    {
    }

    const bool RLogScale;
};

// Forward declarations of abstract EmissionModel and FluidVelocityModel classes.
// See Diagnostics_Emission.h and .cpp for declarations and definitions of these and their descendant classes!
struct EmissionModel;
struct FluidVelocityModel;
// EquatorialEmissionDiagnostic needs to keep track of all tuneable parameters of the emission
// Note that this inherits from EquatorialPassesOptions
struct EquatorialEmissionOptions : public EquatorialPassesOptions
{
public:
    EquatorialEmissionOptions(real thefudgefactor, int equatupper,
                              std::unique_ptr<EmissionModel> theemission, std::unique_ptr<FluidVelocityModel> thefluidmodel,
                              bool rlog, int theredshiftpower, real ththreshold, UpdateFrequency thefrequency) : RedShiftPower{theredshiftpower}, RLogScale{rlog}, GeometricFudgeFactor{thefudgefactor},
        EquatPassUpperBound{equatupper},
        TheEmissionModel{std::move(theemission)}, TheFluidVelocityModel{std::move(thefluidmodel)},
        EquatorialPassesOptions(ththreshold, thefrequency) // constructor for parent class
    {
    }

    // Geometric fudge factor for n>0 passes
    const real GeometricFudgeFactor;
    // Upper bound allowed for contribution to intensity
    const int EquatPassUpperBound;

    // Logarithmic radius scale used or not
    const bool RLogScale;

    // Power of redshift in intensity contribution (should be 3 or 4)
    const int RedShiftPower;

    // The emission model used (see Diagnostics_Emission.h and .cpp for specific models)
    const std::unique_ptr<EmissionModel> TheEmissionModel;
    // The fluid velocity model used (see Diagnostics_Emission.h and .cpp for specific models)
    const std::unique_ptr<FluidVelocityModel> TheFluidVelocityModel;
};

// if necessary, define your new DiagnosticOptions class here
// Sample code:
/*
struct MyDiagnosticOptions : public DiagnosticOptions
{
public:
    // Constructor should pass along UpdateFrequency information to base class
    MyDiagnosticOptions(UpdateFrequency thefrequency) : DiagnosticOptions(thefrequency) //, (...) other initializations
    {}

    // other member variables here - make them const!
    // ...
};
*/

#endif
```

7.10 FOORT/src/DiagnosticsEmission.cpp File Reference

```
#include "DiagnosticsEmission.h"
#include "Integrators.h"
#include <cmath>
```

7.11 FOORT/src/DiagnosticsEmission.h File Reference

```
#include "Geometry.h"
#include "Metric.h"
#include "InputOutput.h"
#include <string>
#include <cmath>
```

Classes

- struct EmissionModel
- struct GLMJohnsonSUEmission
- struct FluidVelocityModel
- struct GeneralCircularRadialFluid

7.12 DiagnosticsEmission.h

Go to the documentation of this file.

```
#ifndef _FOORT_DIAGNOSTICS_EMISSION_H
#define _FOORT_DIAGNOSTICS_EMISSION_H

#include "Geometry.h"  // Tensor objects
#include "Metric.h"      // for Metric functions
#include "InputOutput.h" // for ScreenOutput

#include <string> // for strings
#include <cmath>  // for fmax, fmin

// Here we declare the emission and fluid velocity models used for equatorial disc emission

// Emission model abstract base class
struct EmissionModel
{
public:
    // Virtual destructor to ensure correct destruction
    virtual ~EmissionModel() = default;

    // Function which returns the emitted brightness intensity at Point p
    // Note: always pass the true radius (not log(r)) to EmissionModel!
    virtual real GetEmission(const Point &p) const = 0;

    // Description string getter
    virtual std::string getFullDescriptionStr() const;
};

// The Johnson SU emission model used in GLM
struct GLMJohnsonSUEmission final : public EmissionModel
{
public:
    // Constructor
    GLMJohnsonSUEmission(real mu, real gamma, real sigma) : m_mu{mu}, m_gamma{gamma}, m_sigma{sigma}
    {
    }

    // Emitted brightness (only depends on radius)
    real GetEmission(const Point &p) const final;

    // Description string getter
    std::string getFullDescriptionStr() const final;

private:
    // mu, gamma, sigma are the three parameters of the model
    const real m_mu;
    const real m_gamma;
    const real m_sigma;
};


// Fluid velocity abstract base class
struct FluidVelocityModel
```

```
{
public:
    // Constructor is passed Metric pointer
    FluidVelocityModel(const Metric *const theMetric) : m_theMetric{theMetric} {}

    // Virtual destructor to ensure correct destruction
    virtual ~FluidVelocityModel() = default;

    // Get the local four-velocity (with index down!) of the fluid at Point p
    virtual OneIndex GetFourVelocityd(const Point &p) const = 0;

    // Description string getter
    virtual std::string getFullDescriptionStr() const;

protected:
    // Metric pointer, used for e.g. calculating geodesic orbits
    const Metric *const m_theMetric;
};

// This fluid velocity model has three tuneable parameters and represents fluid travelling at a mix of
// (sub)Keplerian circular orbits and radially infalling orbits in the equatorial plane
struct GeneralCircularRadialFluid final : public FluidVelocityModel
{
    // Constructor with three parameters and Metric pointer (which is passed to base class constructor)
    GeneralCircularRadialFluid(real subKeplerParam, real betar, real betaphi, const Metric *const theMetric) : m_subKeplerParam{fmin(fmax(subKeplerParam, 0.0), 1.0)}, m_betaR{fmin(fmax(betar, 0.0), 1.0)},
        m_betaPhi{fmin(fmax(betaphi, 0.0), 1.0)}, FluidVelocityModel(theMetric)
    {
        // Do some checks on three params, which must lie between 0.0 and 1.0 (note that they are adjusted as such in
        // initializer above)
        if (subKeplerParam < 0.0)
            ScreenOutput("Sub-Keplerian parameter must be between 0 and 1; adjusting to 0", OutputLevel::Level_0_WARNING);
        if (subKeplerParam > 1.0)
            ScreenOutput("Sub-Keplerian parameter must be between 0 and 1; adjusting to 1", OutputLevel::Level_0_WARNING);

        if (betar < 0.0)
            ScreenOutput("beta_r parameter must be between 0 and 1; adjusting to 0", OutputLevel::Level_0_WARNING);
        if (betar > 1.0)
            ScreenOutput("beta_r parameter must be between 0 and 1; adjusting to 1", OutputLevel::Level_0_WARNING);

        if (betaphi < 0.0)
            ScreenOutput("beta_phi parameter must be between 0 and 1; adjusting to 0", OutputLevel::Level_0_WARNING);
        if (betaphi > 1.0)
            ScreenOutput("beta_phi parameter must be between 0 and 1; adjusting to 1", OutputLevel::Level_0_WARNING);

        // Find the (equatorial) ISCO for this Metric
        FindISCO();
    }

    // Get the local four-velocity of the fluid according to this model
    // Note: will always calculate with Point p exactly on the equator theta=pi/2, despite what p[2] may be passed
    OneIndex GetFourVelocityd(const Point &p) const final;

    // Description string getter
    std::string getFullDescriptionStr() const final;

private:
    // Three parameters determining the flow
    const real m_subKeplerParam;
    const real m_betaR;
    const real m_betaPhi;

    // Helper function to get circular (sub)Keplerian velocity outside the ISCO
    OneIndex GetCircularVelocityd(const Point &p, bool subKeplerianOn = true) const;
    // Helper function to get circular and infalling (sub)Keplerian velocity inside ISCO
    // (according to prescription of Cunningham)
    OneIndex GetInsideISCOCircularVelocityd(const Point &p) const;

    // Helper function to get radial infalling velocity
    OneIndex GetRadialVelocityd(const Point &p) const;

    // Helper function to find the ISCO (called in Constructor)
    void FindISCO();
    // ISCO radius and momentum (index down) components
    bool m_ISCOexists{false};
    real m_ISCOr{-1.0};
    real m_ISCOpt{};
    real m_ISCOpphi{};

    // Helper function which returns \partial_r(g^{ab}\Gamma^r_{bc}g^{cd}),
    // whose sign (after contracted with p_a p_d of the corresponding circular orbit)
    // tells us if the circular orbit considered is stable or not
    TwoIndex GetChristrRaisedDer(real r) const;
};

#endif
```

7.13 FOORT/src/Geodesic.cpp File Reference

```
#include "Geodesic.h"
#include "InputOutput.h"
```

7.14 FOORT/src/Geodesic.h File Reference

```
#include "Geometry.h"
#include "Metric.h"
#include "Diagnostics.h"
#include "Terminations.h"
#include "Integrators.h"
#include <string>
#include <vector>
```

Classes

- class Source
- class NoSource
- class Geodesic

7.15 Geodesic.h

Go to the documentation of this file.

```
#ifndef _FOORT_GEODESIC_H
#define _FOORT_GEODESIC_H


#include "Geometry.h"   // for tensor objects
#include "Metric.h"       // for the metric
#include "Diagnostics.h"  // Geodesics own Diagnostics
#include "Terminations.h" // Geodesics own Terminations
#include "Integrators.h"  // Geodesics use an GeodesicIntegratorFunc to integrate itself

#include <string> // for strings
#include <vector> // for std::vector


// Abstract base class
class Source
{
public:
    // Constructor initializes Metric
    Source(const Metric *const theMetric) : m_theMetric{theMetric} {}

    // Virtual destructor to ensure correct descendant destruction
    virtual ~Source() = default;

    // Get the source for the current geodesic position and velocity
    virtual OneIndex getSource(Point pos, OneIndex vel) const = 0;

    // Full description string (space allowed), to be outputted to file
    virtual std::string getFullDescriptionStr() const;

protected:
    // A const pointer to a const metric
    const Metric *const m_theMetric;
};

// NoSource: there is no source, i.e. the geodesic is indeed a geodesic (and feels no force)
class NoSource final : public Source
{
public:
    // Simple constructor passes on the Metric pointer to the base constructor
    NoSource(const Metric *const theMetric) : Source(theMetric) {}

    // Returns zero source
    OneIndex getSource(Point pos, OneIndex vel) const final;

    // Description string getter
    std::string getFullDescriptionStr() const final;
};


// Geodesic class: an instance of this class is created for each Geodesic that is integrated.
// The Geodesic is in charge of integrating itself until termination, updating its Diagnostics accordingly,
// and (after termination) returning the appropriate output.
class Geodesic
{
public:
    // Default constructor not allowed
    Geodesic() = delete;
    // Copy constructor or copy assignment not allowed
    Geodesic(const Geodesic &) = delete;
    Geodesic &operator=(const Geodesic &) = delete;

    // Constructor which creates the Geodesic object
    // Takes the following arguments which initialize the private member variables that remain the same over all geodesics
    // - Metric (pointer)
    // - Source (pointer)
    // - Diagnostic bitflag (& value Diagnostic bitflag) (used to create a vector of new instances of Diagnostics)
    // - Termination bitflag (used to create a vector of new instances of Terminations)
    // - Geodesic integrator function to use for integrating geodesic equation
    Geodesic(const Metric *const theMetric, const Source *const theSource,
             DiagBitflag diagbit, DiagBitflag valdiagbit,
             TermBitflag termbit, GeodesicIntegratorFunc theIntegrator) : m_theMetric{theMetric}, m_theSource{theSource},
        m_AllDiagnostics{CreateDiagnosticVector(diagbit, valdiagbit, this)},
        m_AllTerminations{CreateTerminationVector(termbit, this)},
        m_theIntegrator{theIntegrator}
    {
    }

    // This initializes/resets the geodesic with a given ScreenIndex, initial position, and initial velocity
    // Also resets all Diagnostics and Terminations, resets the TermCondition to Term::Continue,
    // and puts the Geodesic back to lambda = 0.0.
    void Reset(ScreenIndex scrindex, Point initpos, OneIndex initvel);

    // This makes the Geodesic integrate itself one step; then the Geodesic loops through all Terminations and Diagnostics to update
    Term Update();

    // Getters for properties of its internal state
    Term getTermCondition() const;      // Current termination condition (Term::Continue if not done integrating)
    Point getCurrentPos() const;       // Current position
    OneIndex getCurrentVel() const;     // Current velocity
    real getCurrentLambda() const;      // Current value of affine parameter
    ScreenIndex getScreenIndex() const; // screen index

    // Output getters, to be called after the Geodesic terminates
    // This gets the complete output that should be written to the output files;
    // there is one string more than the count of Diagnostics: one string per Diagnostic,
    // PLUS the first string is the screen index.
    std::vector<std::string> getAllOutputStr() const;
    // This returns the "value" (from the Diagnostic that was set to the value Diagnostic) that is associated
    // to the Geodesic. Will be used to determine "distance" between Geodesics which is used in Mesh refinement.
    std::vector<real> getDiagnosticFinalValue() const;

private:
    // These variables define its internal state
    Term m_TermCond{Term::Uninitialized}; // As long as this is Term::Continue, not done integrating yet
    Point m_CurrentPos{};                // Current position
    OneIndex m_CurrentVel{};              // Current proper velocity
    real m_curLambda{0.0};                // Current value of affine parameter (starts at 0.0)

    // The Geodesic keeps track of what index it has been assigned;
    // it outputs this information in its final output string
    ScreenIndex m_ScreenIndex{};

    // These are const pointers (or const vectors of pointers) that contain all the information the Geodesic needs
    const Metric *const m_theMetric; // Metric is needed to evaluate the geodesic equation
    const Source *const m_theSource; // Source for the rhs of the geodesic equation
    // An instance of each Diagnostic and Termination is created for the Geodesic (in its constructor);
    // so the Geodesic is the owner of these objects.
    const DiagnosticUniqueVector m_AllDiagnostics;
    const TerminationUniqueVector m_AllTerminations;
    const GeodesicIntegratorFunc m_theIntegrator; // This is the function that will integrate the geodesic equation one step
};

#endif
```

7.16 FOORT/src/Geometry.h File Reference

```
#include <limits>
#include <string>
#include <array>
#include <utility>
#include <vector>
```

Macros

- #define LARGECOUNTER_MAX std::numeric_limits<largecounter>::max()
- #define PIXEL_MAX std::numeric_limits<pixelcoord>::max()

Typedefs

- using real = double
- using largecounter = unsigned long
- using pixelcoord = largecounter
- using Point = std::array<real, dimension>

  TENSOR DEFINITIONS.
- using ScreenPoint = std::array<real, dimension - 2>
- using ScreenIndex = std::array<pixelcoord, dimension - 2>
- using OneIndex = Point
- using TwoIndex = std::array<OneIndex, dimension>
- using ThreeIndex = std::array<TwoIndex, dimension>
- using FourIndex = std::array<ThreeIndex, dimension>
- using SingularityCoord = std::pair<int, real>
- using Singularity = std::vector<SingularityCoord>

Functions

- template<size_t TensorDim>
  std::string toString (const std::array< largecounter, TensorDim > &theTensor)

  PRINTING TENSORS TO STRING.
- template<size_t TensorDim>
  std::string toString (const std::array< real, TensorDim > &theTensor)
- template<typename Tensor , size_t TensorDim>
  std::string toString (const std::array< Tensor, TensorDim > &theTensor)
- template<typename t , size_t TensorDim>
  std::array< t, TensorDim > operator+ (const std::array< t, TensorDim > &a1, const std::array< t, TensorDim > &a2)

  TENSOR ARITHMETIC: addition/subtraction of tensors, scalar multiplication/division.
- template<typename t , size_t TensorDim>
  std::array< t, TensorDim > operator- (const std::array< t, TensorDim > &a1, const std::array< t, TensorDim > &a2)
- template<typename t , size_t TensorDim>
  std::array< t, TensorDim > operator* (const std::array< t, TensorDim > &t1, real lambda)
- template<typename t , size_t TensorDim>
  std::array< t, TensorDim > operator* (real lambda, const std::array< t, TensorDim > &t1)
- template<typename t , size_t TensorDim>
  std::array< t, TensorDim > operator/ (const std::array< t, TensorDim > &t1, real lambda)

Variables

- constexpr real pi {3.1415926535}

  CONSTANTS.
- constexpr int dimension {4}

7.16.1 Macro Definition Documentation

7.16.1.1 LARGECOUNTER_MAX

```
#define LARGECOUNTER_MAX std::numeric_limits<largecounter>::max()
```

7.16.1.2 PIXEL_MAX

```
#define PIXEL_MAX std::numeric_limits<pixelcoord>::max()
```

7.16.2 Typedef Documentation

7.16.2.1 FourIndex

```
using FourIndex = std::array<ThreeIndex, dimension>
```

7.16.2.2 largecounter

```
using largecounter = unsigned long
```

7.16.2.3 OneIndex

```
using OneIndex = Point
```

7.16.2.4 pixelcoord

```
using pixelcoord = largecounter
```

7.16.2.5 Point

```
using Point = std::array<real, dimension>
```

TENSOR DEFINITIONS.

7.16.2.6 real

```
using real = double
```

7.16.2.7 ScreenIndex

```
using ScreenIndex = std::array<pixelcoord, dimension - 2>
```

7.16.2.8 ScreenPoint

```
using ScreenPoint = std::array<real, dimension - 2>
```

7.16.2.9 Singularity

```
using Singularity = std::vector<SingularityCoord>
```

7.16.2.10 SingularityCoord

```
using SingularityCoord = std::pair<int, real>
```

7.16.2.11 ThreeIndex

```
using ThreeIndex = std::array<TwoIndex, dimension>
```

7.16.2.12 TwoIndex

```
using TwoIndex = std::array<OneIndex, dimension>
```

7.16.3 Function Documentation

7.16.3.1 operator*() [1/2]

```
template<typename t , size_t TensorDim>
std::array< t, TensorDim > operator* (
            const std::array< t, TensorDim > & t1,
            real lambda)
```

7.16.3.2 operator*() [2/2]

```
template<typename t , size_t TensorDim>
std::array< t, TensorDim > operator* (
            real lambda,
            const std::array< t, TensorDim > & t1)
```

7.16.3.3 operator+()

```
template<typename t , size_t TensorDim>
std::array< t, TensorDim > operator+ (
            const std::array< t, TensorDim > & a1,
            const std::array< t, TensorDim > & a2)
```

TENSOR ARITHMETIC: addition/subtraction of tensors, scalar multiplication/division.

7.16.3.4 operator-()

```
template<typename t , size_t TensorDim>
std::array< t, TensorDim > operator- (
            const std::array< t, TensorDim > & a1,
            const std::array< t, TensorDim > & a2)
```

7.16.3.5 operator/()

```
template<typename t , size_t TensorDim>
std::array< t, TensorDim > operator/ (
            const std::array< t, TensorDim > & t1,
            real lambda)
```

7.16.3.6 toString() [1/3]

```
template<size_t TensorDim>
std::string toString (
            const std::array< largecounter, TensorDim > & theTensor)
```

PRINTING TENSORS TO STRING.

7.16.3.7 toString() [2/3]

```
template<size_t TensorDim>
std::string toString (
            const std::array< real, TensorDim > & theTensor)
```

7.16.3.8 toString() [3/3]

```
template<typename Tensor , size_t TensorDim>
std::string toString (
            const std::array< Tensor, TensorDim > & theTensor)
```

7.16.4 Variable Documentation

7.16.4.1 dimension

```
int dimension {4}  [inline], [constexpr]
```

7.16.4.2 pi

```
real pi {3.1415926535}  [inline], [constexpr]
```

CONSTANTS.

7.17 Geometry.h

Go to the documentation of this file.

```
#ifndef _FOORT_GEOMETRY_H
#define _FOORT_GEOMETRY_H


#include <limits>  // for std::numeric_limits
#include <string>  // needed for toString(...) to convert tensors to strings
#include <array>   // needed to define tensors as fixed-size arrays of real or pixelcoord
#include <utility> // needed for std::pair
#include <vector>  // for std::vector

// A real number.
// (Could be changed to use arbitrary precision in the future.)
using real = double;

// Note: An unsigned long is guaranteed to be able to hold at least 4 294 967 295 (4.10^10).
// An unsigned int is only guaranteed to be able to hold 65 535, although
// many modern-day implementations will actually make the int 32-bit and so much larger
//
// This type is used to count geodesics integrated
using largecounter = unsigned long;
// A pixel coordinate: always >=0 and integer; we use the largecounter type
using pixelcoord = largecounter;

// Macro definition of maximum value that can be held in this large counter
#ifndef LARGECOUNTER_MAX
#define LARGECOUNTER_MAX std::numeric_limits<largecounter>::max()
#endif
#ifndef PIXEL_MAX
#define PIXEL_MAX std::numeric_limits<pixelcoord>::max()
#endif


inline constexpr real pi{3.1415926535};
```

```

// The spacetime dimension
inline constexpr int dimension{4};


// A point in spacetime
// Note that coordinates are always assumed to be (t, r, theta, phi)
using Point = std::array<real, dimension>;

// A point on the ViewScreen; this does not have a time or radial extent
using ScreenPoint = std::array<real, dimension - 2>;

// An index on the ViewScreen (row, column)
using ScreenIndex = std::array<pixelcoord, dimension - 2>;

// Object with one index has the same structure as a Point
using OneIndex = Point;
// Object with two indices is an array of OneIndex objects
using TwoIndex = std::array<OneIndex, dimension>;
// Object with three indices is an array of TwoIndex objects
using ThreeIndex = std::array<TwoIndex, dimension>;
// Object with four indices is an array of ThreeIndex objects
using FourIndex = std::array<ThreeIndex, dimension>;

// Definition used to define singularity of arbitrary codimension
// SingularityCoord: pair of (coordinate number, coordinate value)
using SingularityCoord = std::pair<int, real>;
// Singularity: a number of SingularityCoords together that define a arbitrary codimension surface/line/point
using Singularity = std::vector<SingularityCoord>;


// Base case for single index tensor of unsigned integers (ScreenIndex).
// We do not want toString(ScreenIndex) to convert its entries to reals and use the
// implementation for a single index tensor of reals, because we do not want decimal points in our
// string for the ints!
template <size_t TensorDim>
std::string toString(const std::array<largecounter, TensorDim> &theTensor)
{
    std::string theStr{"("}; // no spaces for the innermost brackets
    for (int i = 0; i < TensorDim - 1; ++i)
    {
        // Here we use the std::to_string to convert the real to string
        theStr += std::to_string(theTensor[i]);
        theStr += ", ";
    }
    theStr += std::to_string(theTensor[TensorDim - 1]);
    theStr += ")"; // no spaces for the innermost brackets

    return theStr;
}

// Base case for single index tensor of reals (Point, OneIndex, ScreenPoint)
template <size_t TensorDim>
std::string toString(const std::array<real, TensorDim> &theTensor)
{
    std::string theStr{"("}; // no spaces for the innermost brackets

    for (int i = 0; i < TensorDim - 1; ++i)
    {
        // Here we use the std::to_string to convert the real to string
        theStr += std::to_string(theTensor[i]);
        theStr += ", ";
    }
    theStr += std::to_string(theTensor[TensorDim - 1]);
    theStr += ")"; // no spaces for the innermost brackets

    return theStr;
}

// General printing function for a tensor (TwoIndex, ThreeIndex, FourInedex);
// recursively calls the lower rank tensor to print itself
template <typename Tensor, size_t TensorDim>
std::string toString(const std::array<Tensor, TensorDim> &theTensor)
{
    std::string theStr{"( "}; // All but the innermost brackets have an extra space padding the bracket

    for (int i = 0; i < TensorDim - 1; ++i)
    {
        theStr += toString(theTensor[i]);
        theStr += ", ";
    }
    theStr += toString(theTensor[TensorDim - 1]); // the last element doesn’t have a comma after it

    theStr += " )"; // All but the innermost brackets have an extra space padding the bracket

    return theStr;
}


// Function to recursively call + on the lower rank tensor (OR the underlying reals/ints, if the tensor is rank 1)
template <typename t, size_t TensorDim>
std::array<t, TensorDim> operator+(const std::array<t, TensorDim> &a1, const std::array<t, TensorDim> &a2)
{
    std::array<t, TensorDim> temp{a1};
    for (int i = 0; i < TensorDim; ++i)
        temp[i] = temp[i] + a2[i];

    return temp;
}

// Function to recursively call - on the lower rank tensor (OR the underlying reals/ints, if the tensor is rank 1)
template <typename t, size_t TensorDim>
std::array<t, TensorDim> operator-(const std::array<t, TensorDim> &a1, const std::array<t, TensorDim> &a2)
{
    std::array<t, TensorDim> temp{a1};
    for (int i = 0; i < TensorDim; ++i)
        temp[i] = temp[i] - a2[i];

    return temp;
}

// Function to recursively scalar multiply the lower-rank tensors (OR the underlying reals/ints for the rank-1 tensor)
template <typename t, size_t TensorDim>
std::array<t, TensorDim> operator*(const std::array<t, TensorDim> &t1, real lambda)
{
    std::array<t, TensorDim> temp{t1};
    for (int i = 0; i < TensorDim; ++i)
        temp[i] = static_cast<t>(temp[i] * lambda); // static_cast necessary if t is integral type!
    return temp;
}

// For multiplication with a scalar on the left, call the multiplication on the right defined above
template <typename t, size_t TensorDim>
std::array<t, TensorDim> operator*(real lambda, const std::array<t, TensorDim> &t1)
{
    return t1 * lambda;
}

// For division with a scalar, call the multiplication defined above
template <typename t, size_t TensorDim>
std::array<t, TensorDim> operator/(const std::array<t, TensorDim> &t1, real lambda)
{
    return t1 * (1 / lambda);
}

#endif
```

7.18 FOORT/src/Header.h File Reference

7.19 Header.h

Go to the documentation of this file.

```
#pragma once
```

7.20 FOORT/src/InputOutput.cpp File Reference

```
#include "InputOutput.h"
#include <algorithm>
#include <filesystem>
```

Functions

- void SetOutputLevel (OutputLevel theLvl)
- void SetLoopMessageFrequency (largecounter thefreq)
- largecounter GetLoopMessageFrequency ()
- void ScreenOutput (std::string_view theOutput, OutputLevel lvl, bool newLine)

7.20.1 Function Documentation

7.20.1.1 GetLoopMessageFrequency()

```
largecounter GetLoopMessageFrequency ()
```

7.20.1.2 ScreenOutput()

```
void ScreenOutput (
            std::string_view theOutput,
            OutputLevel lvl,
            bool newLine)
```

7.20.1.3 SetLoopMessageFrequency()

```
void SetLoopMessageFrequency (
            largecounter thefreq)
```

7.20.1.4 SetOutputLevel()

```
void SetOutputLevel (
            OutputLevel theLvl)
```

7.21 FOORT/src/InputOutput.h File Reference

```
#include "Geometry.h"
#include <string_view>
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
```

Classes

- class GeodesicOutputHandler

Enumerations

- enum class OutputLevel {
  Level_0_WARNING = 0 , Level_1_PROC = 1 , Level_2_SUBPROC = 2 , Level_3_ALLDETAIL = 3 ,
  Level_4_DEBUG = 4 , MaxLevel }

Functions

- void SetOutputLevel (OutputLevel theLvl)
- void ScreenOutput (std::string_view theOutput, OutputLevel lvl=OutputLevel::Level_3_ALLDETAIL, bool newLine=true)
- void SetLoopMessageFrequency (largecounter thefreq)
- largecounter GetLoopMessageFrequency ()

7.21.1 Enumeration Type Documentation

7.21.1.1 OutputLevel

```
enum class OutputLevel  [strong]
```

Enumerator

Level_0_WARNING	
Level_1_PROC	
Level_2_SUBPROC	
Level_3_ALLDETAIL	
Level_4_DEBUG	
MaxLevel	

7.21.2 Function Documentation

7.21.2.1 GetLoopMessageFrequency()

```
largecounter GetLoopMessageFrequency ()
```

7.21.2.2 ScreenOutput()

```
void ScreenOutput (
             std::string_view theOutput,
             OutputLevel lvl = OutputLevel::Level_3_ALLDETAIL,
             bool newLine = true)
```

7.21.2.3 SetLoopMessageFrequency()

```
void SetLoopMessageFrequency (
             largecounter thefreq)
```

7.21.2.4 SetOutputLevel()

```
void SetOutputLevel (
            OutputLevel theLvl)
```

7.22 InputOutput.h

Go to the documentation of this file.

```
#ifndef _FOORT_INPUTOUTPUT_H
#define _FOORT_INPUTOUTPUT_H


#include "Geometry.h" // for basic tensor objects

#include <string_view> // std::string_view
#include <iostream>    // needed for file and console output
#include <fstream>     // needed for file ouput
#include <string>      // std::string used in various places
#include <vector>      // needed to create vectors of strings


// Determines at what priority level the output is generated to the console.
enum class OutputLevel
{
    Level_0_WARNING = 0,   // Only warnings are outputted
    Level_1_PROC = 1,      // Coarsest level output; only the major procedures produce output
    Level_2_SUBPROC = 2,   // Subprocedures can also produce output
    Level_3_ALLDETAIL = 3, // Finest level output; all details are shown
    Level_4_DEBUG = 4,    // Finest level output AND debug messages as well

    MaxLevel // (unused)
};

// Set the output level used; note: the output level itself is a static variable in InputOutput.cpp
void SetOutputLevel(OutputLevel theLvl);

// Outputs line to screen console), contingent on it being allowed by the set outputlevel
void ScreenOutput(std::string_view theOutput, OutputLevel lvl = OutputLevel::Level_3_ALLDETAIL, bool newLine = true);

// Set and Get for the loop message frequency (messages indicating progress during each integration loop)
void SetLoopMessageFrequency(largecounter thefreq);
largecounter GetLoopMessageFrequency();

// GeodesicOutputHandler declaration

// GeodesicOutputHandler handles all of the output to file.
// It gets passed all of the output strings for every Geodesic, it then
// stores this data until it eventually writes all data to the appropriate files
class GeodesicOutputHandler
{
public:
    // No default constructor possible
    GeodesicOutputHandler() = delete;
    // Constructor must pass the following strings that are used to construct the file names of the output file:
    // FilePrefix, TimeStamp, FileExtension, and a vector of strings DiagNames (the names of each of the Diagnostics that will
    // be outputting). It must also specify how many outputs to cache before outputting to a file, and
    // how many geodesics are allowed per file created
    GeodesicOutputHandler(std::string FilePrefix, std::string TimeStamp, std::string FileExtension,
                          std::vector<std::string> DiagNames,
                          largecounter nroutputstocache = LARGECOUNTER_MAX - 1, // note -1,
                                                                                // since we will actually cache one more then this number before outputting everything
                          largecounter geodperfile = LARGECOUNTER_MAX,
                          std::string firstlineinfo = "");

    // This tells the OutputHandler to prepare for this many geodesic outputs to arrive;
    // the internal state needs to be prepared such that they can come in without providing a data race
    void PrepareForOutput(largecounter nrOutputToCome);

    // A new vector of output strings from a (single) Geodesic;
    // the length of the vector should be m_DiagNames.size()+1, since the first entry
    // is the screen index
    // NOTE: this procedure needs to be thread-safe!
    void NewGeodesicOutput(largecounter index, std::vector<std::string> theOutput);

    // Calling this indicates that there is no further output to be expected;
    // this means we will write all remaining cached output to file
    void OutputFinished();

    // Returns full description string of output handler
    std::string getFullDescriptionStr() const;

private:
    // Helper function: write everything that is cached to file now (clear the cache)
    void WriteCachedOutputToFile();

    // Helper function: open the file with name filename for the first time, preparing it to write
    // This will effectively clear this file of any pre-existing content.
    void OpenForFirstTime(const std::string &filename);

    // Helper function: return the full file name for the n-th output file
    // for the Diagnostic diagnr (this is an entry in m_DiagNames)
    std::string GetFileName(int diagnr, unsigned short filenr) const;

    // The const strings that are used to construct the output file names
    const std::string m_FilePrefix;
    const std::string m_TimeStamp;
    const std::string m_FileExtension;
    const std::vector<std::string> m_DiagNames;

    // const variables that control whether we write a descriptive first line in
    // every output file or not, and what that first line is
    const bool m_PrintFirstLineInfo;
    const std::string m_FirstLineInfoString;

    // consts setting the maximum number of outputs that can be cached before writing output to file,
    // and the max number of geodesics to store in a file
    const largecounter m_nrOutputsToCache{};
    const largecounter m_nrGeodesicsPerFile{};

    // If this is false, then we are writing to file(s). At any time, if a file I/O error occurs,
    // the output handler switches to outputting everything to the console
    bool m_WriteToConsole{false};

    // How many outputs are already cached in m_nrOutputsToCache before the current iteration of
output
    largecounter m_PrevCached{0};

    // The number of geodesics already written to the current file
    // (once this hits m_nrGeodesicsPerFile, this file is full)
    largecounter m_CurrentGeodesicsInFile{0};

    // The current counter of completely full files (this is kept track of
    // so that it knows what the next file to write output to is)
    // (We had better not have more than 60k files!)
    unsigned short m_CurrentFullFiles{0};

    // Cached data that has not been written to a file yet
    // (once this hits a size of > m_nrOutputsToCache,
    // this must be written to file(s))
    std::vector<std::vector<std::string» m_AllCachedData{};
};

#endif
```

7.23 FOORT/src/Integrators.cpp File Reference

```
#include "Integrators.h"
#include "Geodesic.h"
#include <algorithm>
#include <cmath>
#include <sstream>
#include <iostream>
```

7.24 FOORT/src/Integrators.h File Reference

```
#include "Geometry.h"
#include "Metric.h"
```

```
#include <string>
```

Namespaces

- namespace Integrators

Typedefs

- using GeodesicIntegratorFunc = void (∗)(Point, OneIndex, Point &, OneIndex &, real &, const Metric ∗, const Source ∗)

Functions

- std::string Integrators::GetFullIntegratorDescription ()
- real Integrators::GetAdaptiveStep (Point curpos, OneIndex curvel)
- void Integrators::IntegrateGeodesicStep_RK4 (Point curpos, OneIndex curvel, Point &nextpos, OneIndex &nextvel, real &stepsize, const Metric ∗theMetric, const Source ∗theSource)
- void Integrators::IntegrateGeodesicStep_Verlet (Point curpos, OneIndex curvel, Point &nextpos, OneIndex &nextvel, real &stepsize, const Metric ∗theMetric, const Source ∗theSource)

Variables

- constexpr real Integrators::delta_nodiv0 = 1e-20
- real Integrators::Derivative_hval {1e-7}
- std::string Integrators::IntegratorDescription {"RK4"}
- real Integrators::epsilon {0.03}
- real Integrators::SmallestPossibleStepsize {1e-12}
- real Integrators::VerletVelocityTolerance {0.001}

7.24.1 Typedef Documentation

7.24.1.1 GeodesicIntegratorFunc

```
using GeodesicIntegratorFunc = void (∗)(Point, OneIndex, Point &, OneIndex &, real &, const
Metric ∗, const Source ∗)
```

7.25 Integrators.h

Go to the documentation of this file.

```
#ifndef _FOORT_INTEGRATORS_H
#define _FOORT_INTEGRATORS_H


#include "Geometry.h" // for basic tensor objects
#include "Metric.h"   // for the metric

#include <string> // std::string

// Forward declaration needed of Source
// (Source is declared in Geodesic.h, but we want to avoid a header loop!)
class Source;

// This is the structure of an function that integrates the geodesic equation one step
// It takes as arguments:
// - current position, current velocity
// - references to: next position, next velocity, affine parameter step size (which the functions sets)
// - pointers to: the Metric, the Source that are used to evaluate the geodesic equation
using GeodesicIntegratorFunc = void (*)(Point, OneIndex, Point &, OneIndex &, real &, const Metric *, const Source *);

// Namespace for integrator constants and functions
namespace Integrators
{
    // This is used to avoid dividing by zero
    constexpr real delta_nodiv0 = 1e-20;

    // The amount of any coordinate that we shift to calculate derivatives (using central difference)
    inline real Derivative_hval{1e-7};

    // The name of the integrator selected
    inline std::string IntegratorDescription{"RK4"};

    // Full descriptive string of integrator and all integrator options
    std::string GetFullIntegratorDescription();

    // This is the base step size to be taken (the integrator will adapt this if necessary)
    inline real epsilon{0.03};

    // The affine parameter must always go forward by at least this amount
    inline real SmallestPossibleStepsize{1e-12};

    // Function to get  (adaptive) step size
    real GetAdaptiveStep(Point curpos, OneIndex curvel);

    // This is a GeodesicIntegratorFunc
    // Using the Runge-Kutta-4 algorithm to integrate the geodesic equation
    void IntegrateGeodesicStep_RK4(Point curpos, OneIndex curvel,
                                   Point &nextpos, OneIndex &nextvel, real &stepsize, const Metric *theMetric, const Source *theSource);

    inline real VerletVelocityTolerance{0.001};

    // This is a GeodesicIntegratorFunc
    // Using the velocity Verlet algorithm to integrate the geodesic equation
    void IntegrateGeodesicStep_Verlet(Point curpos, OneIndex curvel,
                                      Point &nextpos, OneIndex &nextvel, real &stepsize, const Metric *theMetric, const Source *theSource);
}

#endif
```

7.26 FOORT/src/Main.cpp File Reference

```
#include "Geometry.h"
#include "Metric.h"
#include "Diagnostics.h"
#include "Terminations.h"
#include "Geodesic.h"
#include "ViewScreen.h"
#include "Integrators.h"
#include "InputOutput.h"
```

```
#include "Utilities.h"
#include <omp.h>
#include <iostream>
#include "Config.h"
```

Functions

- void LoadPrecompiledOptions (std::unique_ptr< Metric > &theM, std::unique_ptr< Source > &theS, DiagBitflag &AllDiags, DiagBitflag &ValDiag, TermBitflag &AllTerms, std::unique_ptr< ViewScreen > &the←View, GeodesicIntegratorFunc &theIntegrator, std::unique_ptr< GeodesicOutputHandler > &theOutput←Handler)
- int main (int argc, char ∗argv[ ])

7.26.1 Function Documentation

7.26.1.1 LoadPrecompiledOptions()

```
void LoadPrecompiledOptions (
            std::unique_ptr< Metric > & theM,
            std::unique_ptr< Source > & theS,
            DiagBitflag & AllDiags,
            DiagBitflag & ValDiag,
            TermBitflag & AllTerms,
            std::unique_ptr< ViewScreen > & theView,
            GeodesicIntegratorFunc & theIntegrator,
            std::unique_ptr< GeodesicOutputHandler > & theOutputHandler)
```

7.26.1.2 main()

```
int main (
            int argc,
            char * argv[ ])
```

7.27 FOORT/src/Mesh.cpp File Reference

```
#include "Mesh.h"
#include "Utilities.h"
#include <algorithm>
#include <limits>
#include <iostream>
```

7.28 FOORT/src/Mesh.h File Reference

```
#include "Geometry.h"
#include "Diagnostics.h"
#include "InputOutput.h"
#include <cmath>
#include <utility>
#include <forward_list>
#include <memory>
#include <vector>
#include <array>
#include <string>
```

Classes

- class Mesh
- class SimpleSquareMesh
- class InputCertainPixelsMesh
- class SquareSubdivisionMesh
- struct SquareSubdivisionMesh::PixelInfo
- class SquareSubdivisionMeshV2
- struct SquareSubdivisionMeshV2::PixelInfo

7.29 Mesh.h

Go to the documentation of this file.

```
#ifndef _FOORT_MESH_H
#define _FOORT_MESH_H


#include "Geometry.h"  // needed for basic tensor objects
#include "Diagnostics.h" // needed for Diagnostic "value" and "distance" functions
#include "InputOutput.h" // needed for ScreenOutput

#include <cmath>        // needed for sqrt (only on Linux)
#include <utility>      // std::move
#include <forward_list> // std::forward_list
#include <memory>       // std::unique_ptr
#include <vector>       // std::vector
#include <array>        // std::array
#include <string>       // for strings

// Abstract Mesh base class
class Mesh
{
public:
    // Basic constructor constructs Diagnostic that is used for determinines "values" and "distances" between values
    Mesh(DiagBitflag valdiag)
        // Calling CreateDiagnosticVector in this way will create a vector with exactly one element in it,
        // i.e. the Diagnostic we need!
        : m_DistanceDiagnostic{std::move((CreateDiagnosticVector(valdiag, valdiag, nullptr))[0])}
    {
    }

    // virtual destructor to ensure correct destruction of descendants
    virtual ~Mesh() = default;

    // Getter for how many geodesics the Mesh currently wants to integrate in this iteration
    virtual largecounter getCurNrGeodesics() const = 0;

    // This sets a new initial conditions (in the form of a ScreenPoint and ScreenIndex)
    // for a next pixel to be integrated in the current iteration
    virtual void getNewInitConds(largecounter index, ScreenPoint &newunitpoint, ScreenIndex &newscreenindex) const = 0;

    // When a geodesic is finished integrating, it tells the Mesh and passes on its final "value"
    // NOTE: despite being a non-const member function, this must be designed to be thread-safe!
    virtual void GeodesicFinished(largecounter index, std::vector<real> finalValues) = 0;

    // This is called when the current iteration is finished. The Mesh can now evaluate whether to continue or not
    virtual void EndCurrentLoop() = 0;

    // Returns false if the Mesh wants another iteration of pixels to integrate
    virtual bool IsFinished() const = 0;

    // Returns a string description of the Mesh (spaces allowed), describing its options
    virtual std::string getFullDescriptionStr() const;

protected:
    // The Diagnostic (a const pointer to a const Diagnostic object) that is used to calculate
    // distances (using FinalDataValDistance()) between the "values" that are assigned to Geodesics
    const std::unique_ptr<const Diagnostic> m_DistanceDiagnostic;
};

// A simple square mesh that will integrate a square of evenly spaced pixels
class SimpleSquareMesh final : public Mesh
{
public:
    // Default constructor not possible
    SimpleSquareMesh() = delete;
    // Constructor initializes total number of pixels and passes valdiag to base constructor
    // Note that we static_cast the sqrt() to round off the row/column size to an integer number
    SimpleSquareMesh(largecounter totalPixels, DiagBitflag valdiag)
        : m_TotalPixels{static_cast<pixelcoord>(sqrt(totalPixels)) * static_cast<pixelcoord>(sqrt(totalPixels))},
          m_RowColumnSize{static_cast<pixelcoord>(sqrt(totalPixels))},
          Mesh(valdiag)
    {
        if constexpr (dimension != 4)
            ScreenOutput("SimpleSquareMesh only defined in 4D!", OutputLevel::Level_0_WARNING);
    }

    // Declarations of overriding virtual functions

    largecounter getCurNrGeodesics() const final;

    void getNewInitConds(largecounter index, ScreenPoint &newunitpoint, ScreenIndex &newscreenindex) const final;

    void GeodesicFinished(largecounter index, std::vector<real> finalValues) final;

    void EndCurrentLoop() final;

    bool IsFinished() const final;

    // Description string getter
    std::string getFullDescriptionStr() const final;

private:
    // Total amount of pixels in grid (is the square of m_RowColumnSize)
    const largecounter m_TotalPixels;
    // Amount of pixels per row or column (square grid)
    const pixelcoord m_RowColumnSize;
    // Are we done integrating or not?
    bool m_Finished{false};
};

// Mesh which integrates only certain user-inputted pixels
class InputCertainPixelsMesh : public Mesh
{
public:
    // Default constructor not possible
    InputCertainPixelsMesh() = delete;
    // Copy constructor not possible
    InputCertainPixelsMesh(const InputCertainPixelsMesh &) = delete;
    // Constructor given in Mesh.cpp file; constructor asks for input of pixels
    InputCertainPixelsMesh(largecounter totalPixels, DiagBitflag valdiag);

    // Declarations of overriding virtual functions

    largecounter getCurNrGeodesics() const final;

    void getNewInitConds(largecounter index, ScreenPoint &newunitpoint, ScreenIndex &newscreenindex) const final;

    void GeodesicFinished(largecounter index, std::vector<real> finalValues) final;

    void EndCurrentLoop() final;

    bool IsFinished() const final;

    // Description string getter
    std::string getFullDescriptionStr() const final;

private:
    // Total pixel size of the screen (square grid)
    const pixelcoord m_RowColumnSize;

    // How many pixels have been inputted in total, i.e. need integrating
    largecounter m_TotalPixels{0};
    // All pixels' location
    std::vector<ScreenIndex> m_PixelsToIntegrate{};
    // Are we finished integrating?
    bool m_Finished{false};
};

// Adaptive subdivision Mesh: starts with evenly spaced, square Mesh,
// then decides to subdivide certain squares of pixels into smaller squares,
// based on which pixels have a bigger "weight", which is defined as the maximum
// "distance" (using the Diagnostic value distance) between the upper-left
// vertex of the square with the other three vertices of the square.
class SquareSubdivisionMesh : public Mesh
{
public:
    // default constructor not possible
    SquareSubdivisionMesh() = delete;
    // Constructor must be called with arguments:
    // - maxPixels: max. nr of pixels that can be integrated in TOTAL, over all iterations (if 0, then this is infinite,
    // i.e. we keep integrating until all squares are maximally subdivided or have weight 0)
    // - initialPixels: initial number of pixels to integrate (spaced equally over the screen)
    // - maxSubdivide: maximum number of times that we can subdivide squares (note: 1 denotes the
    // initial grid, so 2 would be subdividing the squares once)
    // - iterationPixels: maximum number of pixels to subdivide in each integration iteration
    // (max number of pixels that will be integrates is then 5*iterationPixels)
    // - initialSubToFinal: once we decide to subdivide a square, do we automatically keep subdividing it
    // until we reach maxSubdivision?
    // - valdiag: the "value" and "distance" Diagnostic to use
    SquareSubdivisionMesh(largecounter maxPixels, largecounter initialPixels, int maxSubdivide, largecounter iterationPixels, bool initialSubToFinal,
                                DiagBitflag valdiag)
        : m_InitialPixels{static_cast<pixelcoord>(sqrt(initialPixels)) * static_cast<pixelcoord>(sqrt(initialPixels))},
          m_MaxSubdivide{maxSubdivide},
          m_RowColumnSize{static_cast<pixelcoord>((sqrt(initialPixels) - 1) * ExpInt(2, maxSubdivide - 1) + 1)},
          m_PixelsLeft{maxPixels}, m_MaxPixels{maxPixels}, m_InfinitePixels{maxPixels == 0}, m_IterationPixels{iterationPixels},
          m_InitialSubDividideToFinal{initialSubToFinal}, Mesh(valdiag)
    {
        if constexpr (dimension != 4)
            ScreenOutput("SquareSubdivisionMesh only defined in 4D!", OutputLevel::Level_0_WARNING);

        // DEBUG message for constructor (can delete)
        ScreenOutput("SquareSubdivisionMesh constructed: maxPixels: " + (m_InfinitePixels ? "infinite" : std::to_string(maxPixels)) + "; m_InitialPixels: " + std::to_string(m_InitialPixels) + "; m_RowColumnSize: " + std::to_string(m_RowColumnSize), OutputLevel::Level_4_DEBUG);

        // Initialize the initial square, equally spaced grid
        InitializeFirstGrid();
    }

    // Declarations of overriding virtual functions

    largecounter getCurNrGeodesics() const final;

    void getNewInitConds(largecounter index, ScreenPoint &newunitpoint, ScreenIndex &newscreenindex) const final;

    void GeodesicFinished(largecounter index, std::vector<real> finalValues) final;

    void EndCurrentLoop() final;

    bool IsFinished() const final;

    // Description string getter
    std::string getFullDescriptionStr() const final;

private:
    // How many initial pixels (spread uniformly over the grid) do we integrate?
    const largecounter m_InitialPixels;
    // How many times are we allowed  to subdivide a square? Note: the initial grid is already at 1
    const int m_MaxSubdivide;
    // The total size in pixels of a row or column (square grid)
    const pixelcoord m_RowColumnSize;
    // How many pixels per iteration can we subdivide?
    const largecounter m_IterationPixels;
    // How many pixels can we integrate in total over all iterations?
    const largecounter m_MaxPixels;
    // If we decide to subdivide a square, do we automatically subdivide it further to the max level?
    const bool m_InitialSubDivideToFinal;
    // Are we allowed to integrate as many pixels as we want? (m_MaxPixels == 0)
    const bool m_InfinitePixels;

    // How many pixels are we still allowed to integrate (if !m_InfinitePixels)?
    largecounter m_PixelsLeft;

    // A struct the Mesh uses to keep all information about a given pixel
    struct PixelInfo
    {
        // Constructor with its ScreenIndex and current subdivision level
        PixelInfo(ScreenIndex ind, int subdiv) : Index{ind}, SubdivideLevel{subdiv} {}

        // The pixel's screenindex
        ScreenIndex Index{};

        // The level at which the pixel has been subdivided
        // Note: initial grid pixels are at 1; pixels at 0 are pixels that cannot be subdivided
        // (for example, at the right or lower edges)
        int SubdivideLevel{};

        // Weight of the pixel: if negative, this signifies that it needs to be updated/calculated!
        // The weight is determined as the max of the distance (as calculated by the value Diagnostic)
        // between its values and those of its right, lower, and right-lower neighbors.
        real Weight{-1};

        // The values associated to this pixel (as calculated by the value Diagnostic)
        std::vector<real> DiagValue{};

        // Where its lower and right neighbors are located in m_AllPixels
        // Note: the pixel with index 0 is (0,0) and can never be the lower or right neighbor of any other pixel!
        largecounter LowerNbrIndex{0};
        largecounter RightNbrIndex{0};
    };
    // The current queue of pixels to be integrated
    std::vector<PixelInfo> m_CurrentPixelQueue{};
    // A bool for every pixel in the current queue: gets set to true when the pixel is done integrating and gets its values returned
    std::vector<bool> m_CurrentPixelQueueDone{};
    // All pixels that have been integrated already (so does not include the pixels in the current queue)
    std::vector<PixelInfo> m_AllPixels{};

    // Initializes the first nxn screen in m_CurrentPixelQueue
    void InitializeFirstGrid();

    // Updates the neighbors of all pixels (that should have neighbors and don't yet) in m_AllPixels
    void UpdateAllNeighbors();

    // Updates all weights of the pixels in m_AllPixels with weight < 0 and subdiv > 0 and subdiv < m_MaxSubdivide
    // Assumes all squares have neighbors assigned correctly
    void UpdateAllWeights();

    // This will take the pixel m_AllPixels[ind] and subdivide it,
    // adding up to <=5 pixels to the CurrentPixelQueue
    void SubdivideAndQueue(largecounter ind);

    // Helper function to exponentiate an int to an int
    // Note: the result can be larger than fits in an int, but the base is always 2 and the exp is always
    // a number <=m_MaxSubdivide (which is an int)
    pixelcoord ExpInt(int base, int exp);
};

// Adaptive subdivision Mesh: starts with evenly spaced, square Mesh,
// then decides to subdivide certain squares of pixels into smaller squares,
// based on which pixels have a bigger "weight", which is defined as the maximum
// "distance" (using the Diagnostic value distance) between the upper-left
// vertex of the square with the other three vertices of the square.
// V2: new way of dealing with neighbors and looping over pixels
class SquareSubdivisionMeshV2 : public Mesh
{
public:
    // default constructor not possible
    SquareSubdivisionMeshV2() = delete;
    // Constructor must be called with arguments:
    // - maxPixels: max. nr of pixels that can be integrated in TOTAL, over all iterations (if 0, then this is infinite,
    // i.e. we keep integrating until all squares are maximally subdivided or have weight 0)
    // - initialPixels: initial number of pixels to integrate (spaced equally over the screen)
    // - maxSubdivide: maximum number of times that we can subdivide squares (note: 1 denotes the
    // initial grid, so 2 would be subdividing the squares once)
    // - iterationPixels: maximum number of pixels to subdivide in each integration iteration
    // (max number of pixels that will be integrates is then 5*iterationPixels)
    // - initialSubToFinal: once we decide to subdivide a square, do we automatically keep subdividing it
    // until we reach maxSubdivision?
    // - valdiag: the "value" and "distance" Diagnostic to use
    SquareSubdivisionMeshV2(largecounter maxPixels, largecounter initialPixels, int maxSubdivide, largecounter iterationPixels, bool initialSubToFinal,
                            DiagBitflag valdiag)
        : m_InitialPixels{static_cast<pixelcoord>(sqrt(initialPixels)) * static_cast<pixelcoord>(sqrt(initialPixels))},
          m_MaxSubdivide{maxSubdivide},
          m_RowColumnSize{static_cast<pixelcoord>((sqrt(initialPixels) - 1) * ExpInt(2, maxSubdivide - 1) + 1)},
          m_PixelsLeft{maxPixels}, m_MaxPixels{maxPixels}, m_InfinitePixels{maxPixels == 0}, m_IterationPixels{iterationPixels},
          m_InitialSubDividideToFinal{initialSubToFinal}, Mesh(valdiag)
    {
        if constexpr (dimension != 4)
            ScreenOutput("SquareSubdivisionMeshV2 only defined in 4D!", OutputLevel::Level_0_WARNING);

        // Initialize the initial square, equally spaced grid
        InitializeFirstGrid();
    }

    // Declarations of overriding virtual functions

    largecounter getCurNrGeodesics() const final;

    void getNewInitConds(largecounter index, ScreenPoint &newunitpoint, ScreenIndex &newscreenindex) const final;

    void GeodesicFinished(largecounter index, std::vector<real> finalValues) final;

    void EndCurrentLoop() final;

    bool IsFinished() const final;

    // Description string getter
    std::string getFullDescriptionStr() const final;

private:
    // How many initial pixels (spread uniformly over the grid) do we integrate?
    const largecounter m_InitialPixels;
    // How many times are we allowed  to subdivide a square? Note: the initial grid is already at 1
    const int m_MaxSubdivide;
    // The total size in pixels of a row or column (square grid)
    const pixelcoord m_RowColumnSize;
    // How many pixels per iteration can we subdivide?
    const largecounter m_IterationPixels;
    // How many pixels can we integrate in total over all iterations?
    const largecounter m_MaxPixels;
    // If we decide to subdivide a square, do we automatically subdivide it further to the max level?
    const bool m_InitialSubDividideToFinal;
    // Are we allowed to integrate as many pixels as we want? (m_MaxPixels == 0)
    const bool m_InfinitePixels;

    // How many pixels are we still allowed to integrate (if !m_InfinitePixels)?
    largecounter m_PixelsLeft;

    // How many pixels we have integrated so far
    largecounter m_PixelsIntegrated{0};

    // A struct the Mesh uses to keep all information about a given pixel
    struct PixelInfo
    {
        // Constructor with its ScreenIndex and current subdivision level
        PixelInfo(ScreenIndex ind, int subdiv) : Index{ind}, SubdivideLevel{subdiv} {}

        // The pixel's screenindex: this gets set by the constructor and cannot change anymore
        const ScreenIndex Index{};

        // The level at which the pixel has been subdivided
        // Note: initial grid pixels are at 1; pixels at 0 are pixels that cannot be subdivided
        // (for example, at the right or lower edges)
        int SubdivideLevel{};

        // Weight of the pixel: if negative, this signifies that it needs to be updated/calculated!
        // The weight is determined as the max of the distance (as calculated by the value Diagnostic)
        // between its values and those of its right, lower, and right-lower neighbors.
        real Weight{-1};

        // The values associated to this pixel (as calculated by the value Diagnostic)
        std::vector<real> DiagValue{};

        // Pointers to its neighbors
        PixelInfo *LeftNbr{nullptr};
        PixelInfo *RightNbr{nullptr};
        PixelInfo *UpNbr{nullptr};
        PixelInfo *DownNbr{nullptr};
        PixelInfo *SEdiagNbr{nullptr};
    };

    // Master list of all pixels
    // std::forward_list is more space-efficient than std::list, bidirectional iteration is not needed, no random access supported
    // This list is only used to store the owner pointers (and thus the objects) of all pixels.
    // The other pixel vectors are used to iterate through (and need random access)
    std::forward_list<std::unique_ptr<PixelInfo>> m_AllPixels{};

    // List of active pixels, i.e. those that can be subdivided and have non-zero weight
    std::vector<PixelInfo *> m_ActivePixels{};
    // List of current queue of pixels to be sent to be integrated
    std::vector<PixelInfo *> m_CurrentPixelQueue{};
    // A bool for every pixel in the current queue: gets set to true when the pixel is done integrating and gets its values returned
    std::vector<bool> m_CurrentPixelQueueDone{};
    // List of pixels that are already integrated but need updating weights after current queue is all integrated
    std::vector<PixelInfo *> m_CurrentPixelUpdating{};

    // Initializes the first nxn screen and puts them in m_CurrentPixelQueue
    void InitializeFirstGrid();

    // Updates all weights of the pixels in m_CurrentPixelUpdating;
    // these should have subdiv > 0 and subdiv < m_MaxSubdivide, and all their neigbors assigned correctly
    // All pixels with weight > 0 will be added to m_ActivePixels
    void UpdateAllWeights();

    // Helper functions that return the appropriate neighbor of p, ONLY if this neighbor exists at the subdivision level specified
    // Returns nullptr otherwise; also return nullptr if p==nullptr
    PixelInfo *GetUp(PixelInfo *p, int subdiv) const;
    PixelInfo *GetDown(PixelInfo *p, int subdiv) const;
    PixelInfo *GetRight(PixelInfo *p, int subdiv) const;
    PixelInfo *GetLeft(PixelInfo *p, int subdiv) const;

    // This will take the pixel m_AllPixels[ind] and subdivide it,
    // adding up to <=5 pixels to the CurrentPixelQueue
    void SubdivideAndQueue(largecounter ind);

    // Helper function to exponentiate an int to an int
    // Note: the result can be larger than fits in an int, but the base is always 2 and the exp is always
    // a number <=m_MaxSubdivide (which is an int)
    pixelcoord ExpInt(int base, int exp) const;
};

#endif
```

7.30 FOORT/src/Metric.cpp File Reference

```
#include "Metric.h"
#include "InputOutput.h"
#include "Integrators.h"
#include <cmath>
#include <algorithm>
#include "Spline.h"
#include <sstream>
```

7.31 FOORT/src/Metric.h File Reference

```
#include "Geometry.h"
#include "Spline.h"
```

```
#include <string>
#include <vector>
```

Classes

- class Metric
- class SphericalHorizonMetric
- class KerrMetric
- class FlatSpaceMetric
- class RasheedLarsenMetric
- class JohannsenMetric
- class MankoNovikovMetric
- class KerrSchildMetric
- class SingularityMetric
- class ST3CrMetric
- class BosonStarMetric

7.32 Metric.h

Go to the documentation of this file.

```
#ifndef _FOORT_METRIC_H
#define _FOORT_METRIC_H

#include "Geometry.h" // Needed for basic tensor objects etc.

#include "Spline.h"
#include <string> // for strings
#include <vector> // needed for the (non-fixed size) vector of symmetries in the metric


// The abstract base class for all Metrics.
class Metric
{
public:
    // Virtual destructor to ensure correct destruction of descendants
    virtual ~Metric() = default;

    Metric(bool rlogscale = false);

    // Basic functions that return the metric with indices down or up:
    // pure virtual as they must be defined in the descendant classes.
    //
    // Get the metric at Point p, indices down
    virtual TwoIndex getMetric_dd(const Point &p) const = 0;
    // Get the metric at Point p, indices up
    virtual TwoIndex getMetric_uu(const Point &p) const = 0;

    // The following functions return the Christoffel and other derivative quantities of the metric.
    // They are implemented for this base class, BUT are left as virtual functions to allow for
    // other metrics to implement their own (more efficient)
    // way of calculating them, if so desired.
    //
    // Get the Christoffel symbol, indices up-down-down
    virtual ThreeIndex getChristoffel_udd(const Point &p) const;
    // Get the Riemann tensor, indices up-down-down-down
    virtual FourIndex getRiemann_uddd(const Point &p) const;
    // Get the Kretschmann scalar
    virtual real getKretschmann(const Point &p) const;

    // Function to get the description of the metric
    // (used for outputting to the screen while running and possibly to the output files)
    // There is a base class implementation of this function returning an undescriptive string
    virtual std::string getFullDescriptionStr() const;

    bool getrLogScale() const;

protected:
    // The symmetries (coordinate Killing vectors) of the metric. Should be set by descendant constructor.
    std::vector<int> m_Symmetries{};
    // Are we using a logarithmic r coordinate?
    const bool m_rLogScale;
};

// Abstract base class for a metric that has a spherical horizon (i.e. horizon at constant radius r)
class SphericalHorizonMetric : public Metric
{
public:
    // No default construction allowed, must specify horizon radius
    SphericalHorizonMetric() = delete;
    // Constructor that initializes horizon radius
    SphericalHorizonMetric(real HorizonRadius, bool rLogScale);

    // Getter functions for the two member variables
    real getHorizonRadius() const;

protected:
    // Radius of the horizon
    const real m_HorizonRadius;
};

// The Kerr metric (normalized so that M = 1)
class KerrMetric final : public SphericalHorizonMetric
{
private:
    // Mass-rescaled rotation parameter for Kerr
    // Note that this should be between -1 and 1.
    const real m_aParam;

    // Mass parameter for Kerr. Default is 1.
    const real m_mParam;

public:
    // No default constructor allowed, must specify a
    KerrMetric() = delete;

    // Constructor setting parameter a
    KerrMetric(real aParam, bool rLogScale = false, real mParam = 1.);

    // The override of the basic metric getter functions
    TwoIndex getMetric_dd(const Point &p) const final;
    TwoIndex getMetric_uu(const Point &p) const final;

    // The override of the description string getter
    std::string getFullDescriptionStr() const final;
};

// Flat space (4D)
class FlatSpaceMetric final : public Metric
{
public:
    // Simple (default) constructor is all that is needed
    FlatSpaceMetric(bool rlogscale = false);

    // The override of the basic metric getter functions
    TwoIndex getMetric_dd(const Point &p) const final;
    TwoIndex getMetric_uu(const Point &p) const final;

    // The override of the description string getter
    std::string getFullDescriptionStr() const final;
};

// Rasheed-Larsen black hole
class RasheedLarsenMetric final : public SphericalHorizonMetric
{
private:
    // Rasheed-Larsen is specified by four parameters
    const real m_aParam;
    const real m_mParam;
    const real m_pParam;
    const real m_qParam;

public:
    // No default constructor allowed, must specify parameters
    RasheedLarsenMetric() = delete;

    // Constructor setting parameter a
    RasheedLarsenMetric(real mParam, real aParam, real pParam, real qParam, bool rLogScale = false);

    // The override of the basic metric getter functions
    TwoIndex getMetric_dd(const Point &p) const final;
    TwoIndex getMetric_uu(const Point &p) const final;

    // The override of the description string getter
    std::string getFullDescriptionStr() const final;
};
```

```

// Johanssen black hole metric (implementation by Seppe Staelens)
class JohannsenMetric final : public SphericalHorizonMetric
{
private:
    // Johannsen up to first order in deviation function is specified by five parameters (if M=1)
    const real m_aParam;
    const real m_alpha13Param;
    const real m_alpha22Param;
    const real m_alpha52Param;
    const real m_eps3Param;

public:
    // No default constructor allowed, must specify parameters
    JohannsenMetric() = delete;

    // Constructor setting parameter a
    JohannsenMetric(real aParam, real alpha13Param, real alpha22Param, real alpha52Param, real eps3Param, bool rLogScale = false);

    // The override of the basic metric getter functions
    TwoIndex getMetric_dd(const Point &p) const final;
    TwoIndex getMetric_uu(const Point &p) const final;

    // The override of the description string getter
    std::string getFullDescriptionStr() const final;
};

// Manko-Novikov metric (with angular momentum and M3 parameter turned on) (implementation by Seppe Staelens)
class MankoNovikovMetric final : public SphericalHorizonMetric
{
private:
    // Manko-Novikov metric with only alpha3 as symmetry breaking parameter
    const real m_aParam;
    const real m_alpha3Param;

    // These are convenient derived quantities from a
    const real m_alphaParam;
    const real m_kParam;

public:
    // No default constructor allowed, must specify parameters
    MankoNovikovMetric() = delete;

    // Constructor setting parameter a and alpha3
    MankoNovikovMetric(real aParam, real alpha3Param, bool rLogScale = false);

    // The override of the basic metric getter functions
    TwoIndex getMetric_dd(const Point &p) const final;
    TwoIndex getMetric_uu(const Point &p) const final;

    // The override of the description string getter
    std::string getFullDescriptionStr() const final;
};

// The Kerr metric in Kerr-Schild coordinates (normalized so that M = 1)
class KerrSchildMetric final : public SphericalHorizonMetric
{
private:
    // Rotation parameter for Kerr
    // Note that this should be between -1 and 1 since M=1
    const real m_aParam;

public:
    // No default constructor allowed, must specify a
    KerrSchildMetric() = delete;

    // Constructor setting parameter a
    KerrSchildMetric(real aParam, bool rLogScale = false);

    // The override of the basic metric getter functions
    TwoIndex getMetric_dd(const Point &p) const final;
    TwoIndex getMetric_uu(const Point &p) const final;

    // The override of the description string getter
    std::string getFullDescriptionStr() const final;
};

// Abstract base class for a metric with an arbitrary number of singularities (of arbitrary codimension)
class SingularityMetric : public Metric
{
public:
    // Constructor that initializes singularities
    SingularityMetric(std::vector<Singularity> thesings, bool rLogScale);

    // Getter functions for the two member variables
    std::vector<Singularity> getSingularities() const;

protected:
    // All singularities of the metric
    const std::vector<Singularity> m_AllSingularities;
};

// Ring fuzzball (implementation Lies Van Dael)
class ST3CrMetric final : public SingularityMetric
{
public:
    // Constructor which will be called to initialize all parameters of the metric
    ST3CrMetric(real P, real q0, real lambda, bool rlogscale = false);

    // The basic getter functions
    TwoIndex getMetric_dd(const Point &p) const final;
    TwoIndex getMetric_uu(const Point &p) const final;

    // The description string getter
    std::string getFullDescriptionStr() const final;

private:
    const real m_P;
    const real m_q0;
    const real m_lambda;

    real get_omega(real r, real theta, real l) const;
    real f_phi(real phi, real r, real theta, real l, real R) const;
    real f_om_phi(real phi, real r, real theta, real l, real R) const;
};

// Boson star with solitonic potential (sigma = 0.06, phi_c = 0.044) (implementation by Seppe Staelens)
class BosonStarMetric final : public Metric
{
public:
    // Simple (default) constructor is all that is needed
    BosonStarMetric(bool rLogScale = false);
    // The override of the basic metric getter functions
    TwoIndex getMetric_dd(const Point &p) const final;
    TwoIndex getMetric_uu(const Point &p) const final;
    // The override of the description string getter
    std::string getFullDescriptionStr() const final;

protected:
    // The spline interpolator for Phi
    tk::spline m_PhiSpline;
    // The spline interpolator for m
    tk::spline m_mSpline;
};

// Declare your new Metric class here, publically inheriting from the base class Metric
// (or SphericalHorizonMetric if your Metric has a horizon, or SingularityMetric if your Metric has other, arbitrary singularities)
// Give definitions (implementation) of these functions in Metric.cpp (or other source code file)
// Don't forget to set m_Symmetries appropriately (in the constructor),
// if your metric has any symmetry (e.g. stationarity, axisymmetry)!
// Sample code:
/*
class MyMetric final : public Metric // good practice to make the class final unless descendant classes are possible
{
public:
    // Constructor which will be called to initialize all parameters of the metric
    MyMetric(args...);

    // The basic getter functions
    // These MUST be implemented
    TwoIndex getMetric_dd(const Point& p) const final;
    TwoIndex getMetric_uu(const Point& p) const final;

    // The description string getter
    // This is optional (but recommended) to implement; if not implemented,
    // the base class Metric::getFullDescriptionStr() will be called instead
    std::string getFullDescriptionStr() const final;

private:
    // good practice to have all const params (initialized in the constructor)
    // since the metric cannot change after initialization
    // const params...;

};
*/

#endif
```

7.33 FOORT/src/Spline.h File Reference

```
#include <cstdio>
#include <cassert>
#include <cmath>
#include <vector>
#include <algorithm>
```

Namespaces

- namespace tk
- namespace tk::internal

7.34 Spline.h

Go to the documentation of this file.

```
/*
 * spline.h
 *
 * simple cubic spline interpolation library without external
 * dependencies
 *
 * ---------------------------------------------------------------------
 * Copyright (C) 2011, 2014, 2016, 2021 Tino Kluge (ttk448 at gmail.com)
 *
 *  This program is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU General Public License
 *  as published by the Free Software Foundation; either version 2
 *  of the License, or (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program.  If not, see <http://www.gnu.org/licenses/>.
 * ---------------------------------------------------------------------
 *
 */

#ifndef TK_SPLINE_H
#define TK_SPLINE_H

#include <cstdio>
#include <cassert>
#include <cmath>
#include <vector>
#include <algorithm>
#ifdef HAVE_SSTREAM
#include <sstream>
#include <string>
#endif // HAVE_SSTREAM

// not ideal but disable unused-function warnings
// (we get them because we have implementations in the header file,
// and this is because we want to be able to quickly separate them
// into a cpp file if necessary)
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused-function"

// unnamed namespace only because the implementation is in this
// header file and we don't want to export symbols to the obj files
namespace
{

    namespace tk
    {

        // spline interpolation
        class spline
        {
        public:
            // spline types
            enum spline_type
            {
                linear = 10,            // linear interpolation
                cspline = 30,           // cubic splines (classical C^2)
                cspline_hermite = 31 // cubic hermite splines (local, only C^1)
            };

            // boundary condition type for the spline end-points
            enum bd_type
            {
                first_deriv = 1,
                second_deriv = 2
            };

        protected:
            std::vector<double> m_x, m_y; // x,y coordinates of points
            // interpolation parameters
            // f(x) = a_i + b_i*(x-x_i) + c_i*(x-x_i)^2 + d_i*(x-x_i)^3
            // where a_i = y_i, or else it won't go through grid points
            std::vector<double> m_b, m_c, m_d; // spline coefficients
            double m_c0;                       // for left extrapolation
            spline_type m_type;
            bd_type m_left, m_right;
            double m_left_value, m_right_value;
            bool m_made_monotonic;
            void set_coeffs_from_b();            // calculate c_i, d_i from b_i
            size_t find_closest(double x) const; // closest idx so that m_x[idx]<=x

        public:
            // default constructor: set boundary condition to be zero curvature
            // at both ends, i.e. natural splines
            spline() : m_type(cspline),
                       m_left(second_deriv), m_right(second_deriv),
                       m_left_value(0.0), m_right_value(0.0), m_made_monotonic(false)
            {
                ;
            }
            spline(const std::vector<double> &X, const std::vector<double> &Y,
                   spline_type type = cspline,
                   bool make_monotonic = false,
                   bd_type left = second_deriv, double left_value = 0.0,
                   bd_type right = second_deriv, double right_value = 0.0) : m_type(type),
                                                                             m_left(left), m_right(right),
                                                                             m_left_value(left_value), m_right_value(right_value),
                                                                             m_made_monotonic(false) // false correct here: make_monotonic() sets it
            {
                this->set_points(X, Y, m_type);
                if (make_monotonic)
                {
                    this->make_monotonic();
                }
            }

            // modify boundary conditions: if called it must be before set_points()
            void set_boundary(bd_type left, double left_value,
                              bd_type right, double right_value);

            // set all data points (cubic_spline=false means linear interpolation)
            void set_points(const std::vector<double> &x,
                            const std::vector<double> &y,
                            spline_type type = cspline);

            // adjust coefficients so that the spline becomes piecewise monotonic
            // where possible
            //   this is done by adjusting slopes at grid points by a non-negative
            //   factor and this will break C^2
            //   this can also break boundary conditions if adjustments need to
            //   be made at the boundary points
            // returns false if no adjustments have been made, true otherwise
            bool make_monotonic();

            // evaluates the spline at point x
            double operator()(double x) const;
            double deriv(int order, double x) const;

            // returns the input data points
            std::vector<double> get_x() const { return m_x; }
            std::vector<double> get_y() const { return m_y; }
            double get_x_min() const
            {
                assert(!m_x.empty());
                return m_x.front();
            }
            double get_x_max() const
            {
                assert(!m_x.empty());
                return m_x.back();
            }

#ifdef HAVE_SSTREAM
            // spline info string, i.e. spline type, boundary conditions etc.
            std::string info() const;
#endif // HAVE_SSTREAM
        };

        namespace internal
        {

            // band matrix solver
            class band_matrix
            {
            private:
                std::vector<std::vector<double» m_upper; // upper band
                std::vector<std::vector<double» m_lower; // lower band
            public:
                band_matrix() {};                       // constructor
                band_matrix(int dim, int n_u, int n_l); // constructor
                ~band_matrix() {};                      // destructor
                void resize(int dim, int n_u, int n_l); // init with dim,n_u,n_l
                int dim() const;                        // matrix dimension
                int num_upper() const
                {
                    return (int)m_upper.size() - 1;
                }
                int num_lower() const
                {
                    return (int)m_lower.size() - 1;
                }
                // access operator
                double &operator()(int i, int j);      // write
                double operator()(int i, int j) const; // read
                // we can store an additional diagonal (in m_lower)
                double &saved_diag(int i);
                double saved_diag(int i) const;
                void lu_decompose();
                std::vector<double> r_solve(const std::vector<double> &b) const;
                std::vector<double> l_solve(const std::vector<double> &b) const;
                std::vector<double> lu_solve(const std::vector<double> &b,
                                             bool is_lu_decomposed = false);
            };

        } // namespace internal

        // ---------------------------------------------------------------------
        // implementation part, which could be separated into a cpp file
        // ---------------------------------------------------------------------

        // spline implementation
        // -----------------------

        void spline::set_boundary(spline::bd_type left, double left_value,
                                  spline::bd_type right, double right_value)
        {
            assert(m_x.size() == 0); // set_points() must not have happened yet
            m_left = left;
            m_right = right;
            m_left_value = left_value;
            m_right_value = right_value;
        }

        void spline::set_coeffs_from_b()
        {
            assert(m_x.size() == m_y.size());
            assert(m_x.size() == m_b.size());
            assert(m_x.size() > 2);
            size_t n = m_b.size();
            if (m_c.size() != n)
                m_c.resize(n);
            if (m_d.size() != n)
                m_d.resize(n);

            for (size_t i = 0; i < n - 1; i++)
            {
                const double h = m_x[i + 1] - m_x[i];
                // from continuity and differentiability condition
                m_c[i] = (3.0 * (m_y[i + 1] - m_y[i]) / h - (2.0 * m_b[i] + m_b[i + 1])) / h;
                // from differentiability condition
                m_d[i] = ((m_b[i + 1] - m_b[i]) / (3.0 * h) - 2.0 / 3.0 * m_c[i]) / h;
            }

            // for left extrapolation coefficients
            m_c0 = (m_left == first_deriv) ? 0.0 : m_c[0];
        }

        void spline::set_points(const std::vector<double> &x,
                                const std::vector<double> &y,
                                spline_type type)
        {
            assert(x.size() == y.size());
            assert(x.size() > 2);
            m_type = type;
            m_made_monotonic = false;
            m_x = x;
            m_y = y;
            int n = (int)x.size();
            // check strict monotonicity of input vector x
            for (int i = 0; i < n - 1; i++)
            {
                assert(m_x[i] < m_x[i + 1]);
            }

            if (type == linear)
            {
                // linear interpolation
                m_d.resize(n);
                m_c.resize(n);
                m_b.resize(n);
                for (int i = 0; i < n - 1; i++)
                {
                    m_d[i] = 0.0;
                    m_c[i] = 0.0;
                    m_b[i] = (m_y[i + 1] - m_y[i]) / (m_x[i + 1] - m_x[i]);
                }
                // ignore boundary conditions, set slope equal to the last segment
                m_b[n - 1] = m_b[n - 2];
                m_c[n - 1] = 0.0;
                m_d[n - 1] = 0.0;
            }
            else if (type == cspline)
            {
                // classical cubic splines which are C^2 (twice cont differentiable)
                // this requires solving an equation system

                // setting up the matrix and right hand side of the equation system
                // for the parameters b[]
                internal::band_matrix A(n, 1, 1);
                std::vector<double> rhs(n);
                for (int i = 1; i < n - 1; i++)
                {
                    A(i, i - 1) = 1.0 / 3.0 * (x[i] - x[i - 1]);
                    A(i, i) = 2.0 / 3.0 * (x[i + 1] - x[i - 1]);
                    A(i, i + 1) = 1.0 / 3.0 * (x[i + 1] - x[i]);
                    rhs[i] = (y[i + 1] - y[i]) / (x[i + 1] - x[i]) - (y[i] - y[i - 1]) / (x[i] - x[i - 1]);
                }
                // boundary conditions
                if (m_left == spline::second_deriv)
                {
                    // 2*c[0] = f”
                    A(0, 0) = 2.0;
                    A(0, 1) = 0.0;
                    rhs[0] = m_left_value;
                }
                else if (m_left == spline::first_deriv)
                {
                    // b[0] = f’, needs to be re-expressed in terms of c:
                    // (2c[0]+c[1])(x[1]-x[0]) = 3 ((y[1]-y[0])/(x[1]-x[0]) - f’)
                    A(0, 0) = 2.0 * (x[1] - x[0]);
                    A(0, 1) = 1.0 * (x[1] - x[0]);
                    rhs[0] = 3.0 * ((y[1] - y[0]) / (x[1] - x[0]) - m_left_value);
                }
                else
                {
                    assert(false);
                }
                if (m_right == spline::second_deriv)
                {
                    // 2*c[n-1] = f”
                    A(n - 1, n - 1) = 2.0;
                    A(n - 1, n - 2) = 0.0;
                    rhs[n - 1] = m_right_value;
                }
                else if (m_right == spline::first_deriv)
                {
                    // b[n-1] = f’, needs to be re-expressed in terms of c:
```

```
                    // (c[n-2]+2c[n-1])(x[n-1]-x[n-2])
                    // = 3 (f' - (y[n-1]-y[n-2])/(x[n-1]-x[n-2]))
                    A(n - 1, n - 1) = 2.0 * (x[n - 1] - x[n - 2]);
                    A(n - 1, n - 2) = 1.0 * (x[n - 1] - x[n - 2]);
                    rhs[n - 1] = 3.0 * (m_right_value - (y[n - 1] - y[n - 2]) / (x[n - 1] - x[n - 2]));
                }
                else
                {
                    assert(false);
                }

                // solve the equation system to obtain the parameters c[]
                m_c = A.lu_solve(rhs);

                // calculate parameters b[] and d[] based on c[]
                m_d.resize(n);
                m_b.resize(n);
                for (int i = 0; i < n - 1; i++)
                {
                    m_d[i] = 1.0 / 3.0 * (m_c[i + 1] - m_c[i]) / (x[i + 1] - x[i]);
                    m_b[i] = (y[i + 1] - y[i]) / (x[i + 1] - x[i]) - 1.0 / 3.0 * (2.0 * m_c[i] + m_c[i + 1]) * (x[i + 1] - x[i]);
                }
                // for the right extrapolation coefficients (zero cubic term)
                // f_{n-1}(x) = y_{n-1} + b*(x-x_{n-1}) + c*(x-x_{n-1})^2
                double h = x[n - 1] - x[n - 2];
                // m_c[n-1] is determined by the boundary condition
                m_d[n - 1] = 0.0;
                m_b[n - 1] = 3.0 * m_d[n - 2] * h * h + 2.0 * m_c[n - 2] * h + m_b[n - 2]; // = f'_{n-2}(x_{n-1})
                if (m_right == first_deriv)
                    m_c[n - 1] = 0.0; // force linear extrapolation
            }
            else if (type == cspline_hermite)
            {
                // hermite cubic splines which are C^1 (cont. differentiable)
                // and derivatives are specified on each grid point
                // (here we use 3-point finite differences)
                m_b.resize(n);
                m_c.resize(n);
                m_d.resize(n);
                // set b to match 1st order derivative finite difference
                for (int i = 1; i < n - 1; i++)
                {
                    const double h = m_x[i + 1] - m_x[i];
                    const double hl = m_x[i] - m_x[i - 1];
                    m_b[i] = -h / (hl * (hl + h)) * m_y[i - 1] + (h - hl) / (hl * h) * m_y[i] + hl / (h * (hl + h)) * m_y[i + 1];
                }
                // boundary conditions determine b[0] and b[n-1]
                if (m_left == first_deriv)
                {
                    m_b[0] = m_left_value;
                }
                else if (m_left == second_deriv)
                {
                    const double h = m_x[1] - m_x[0];
                    m_b[0] = 0.5 * (-m_b[1] - 0.5 * m_left_value * h + 3.0 * (m_y[1] - m_y[0]) / h);
                }
                else
                {
                    assert(false);
                }
                if (m_right == first_deriv)
                {
                    m_b[n - 1] = m_right_value;
                    m_c[n - 1] = 0.0;
                }
                else if (m_right == second_deriv)
                {
                    const double h = m_x[n - 1] - m_x[n - 2];
                    m_b[n - 1] = 0.5 * (-m_b[n - 2] + 0.5 * m_right_value * h + 3.0 * (m_y[n - 1] - m_y[n - 2]) / h);
                    m_c[n - 1] = 0.5 * m_right_value;
                }
                else
                {
                    assert(false);
                }
                m_d[n - 1] = 0.0;

                // parameters c and d are determined by continuity and differentiability
                set_coeffs_from_b();
            }
            else
            {
```

```
            assert(false);
        }

        // for left extrapolation coefficients
        m_c0 = (m_left == first_deriv) ? 0.0 : m_c[0];
    }

    bool spline::make_monotonic()
    {
        assert(m_x.size() == m_y.size());
        assert(m_x.size() == m_b.size());
        assert(m_x.size() > 2);
        bool modified = false;
        const int n = (int)m_x.size();
        // make sure: input data monotonic increasing --> b_i>=0
        //            input data monotonic decreasing --> b_i<=0
        for (int i = 0; i < n; i++)
        {
            int im1 = std::max(i - 1, 0);
            int ip1 = std::min(i + 1, n - 1);
            if (((m_y[im1] <= m_y[i]) && (m_y[i] <= m_y[ip1]) && m_b[i] < 0.0) ||
                ((m_y[im1] >= m_y[i]) && (m_y[i] >= m_y[ip1]) && m_b[i] > 0.0))
            {
                modified = true;
                m_b[i] = 0.0;
            }
        }
        // if input data is monotonic (b[i], b[i+1], avg have all the same sign)
        // ensure a sufficient criteria for monotonicity is satisfied:
        //     sqrt(b[i]^2+b[i+1]^2) <= 3 |avg|, with avg=(y[i+1]-y[i])/h,
        for (int i = 0; i < n - 1; i++)
        {
            double h = m_x[i + 1] - m_x[i];
            double avg = (m_y[i + 1] - m_y[i]) / h;
            if (avg == 0.0 && (m_b[i] != 0.0 || m_b[i + 1] != 0.0))
            {
                modified = true;
                m_b[i] = 0.0;
                m_b[i + 1] = 0.0;
            }
            else if ((m_b[i] >= 0.0 && m_b[i + 1] >= 0.0 && avg > 0.0) ||
                     (m_b[i] <= 0.0 && m_b[i + 1] <= 0.0 && avg < 0.0))
            {
                // input data is monotonic
                double r = sqrt(m_b[i] * m_b[i] + m_b[i + 1] * m_b[i + 1]) / std::fabs(avg);
                if (r > 3.0)
                {
                    // sufficient criteria for monotonicity: r<=3
                    // adjust b[i] and b[i+1]
                    modified = true;
                    m_b[i] *= (3.0 / r);
                    m_b[i + 1] *= (3.0 / r);
                }
            }
        }

        if (modified == true)
        {
            set_coeffs_from_b();
            m_made_monotonic = true;
        }

        return modified;
    }

    // return the closest idx so that m_x[idx] <= x (return 0 if x<m_x[0])
    size_t spline::find_closest(double x) const
    {
        std::vector<double>::const_iterator it;
        it = std::upper_bound(m_x.begin(), m_x.end(), x);    // *it > x
        size_t idx = std::max(int(it - m_x.begin()) - 1, 0); // m_x[idx] <= x
        return idx;
    }

    double spline::operator()(double x) const
    {
        // polynomial evaluation using Horner's scheme
        // TODO: consider more numerically accurate algorithms, e.g.:
        //   - Clenshaw
        //   - Even-Odd method by A.C.R. Newbery
        //   - Compensated Horner Scheme
        size_t n = m_x.size();
        size_t idx = find_closest(x);

        double h = x - m_x[idx];
        double interpol;
        if (x < m_x[0])
```

```
            {
                // extrapolation to the left
                interpol = (m_c0 * h + m_b[0]) * h + m_y[0];
            }
            else if (x > m_x[n - 1])
            {
                // extrapolation to the right
                interpol = (m_c[n - 1] * h + m_b[n - 1]) * h + m_y[n - 1];
            }
            else
            {
                // interpolation
                interpol = ((m_d[idx] * h + m_c[idx]) * h + m_b[idx]) * h + m_y[idx];
            }
            return interpol;
        }

        double spline::deriv(int order, double x) const
        {
            assert(order > 0);
            size_t n = m_x.size();
            size_t idx = find_closest(x);

            double h = x - m_x[idx];
            double interpol;
            if (x < m_x[0])
            {
                // extrapolation to the left
                switch (order)
                {
                case 1:
                    interpol = 2.0 * m_c0 * h + m_b[0];
                    break;
                case 2:
                    interpol = 2.0 * m_c0;
                    break;
                default:
                    interpol = 0.0;
                    break;
                }
            }
            else if (x > m_x[n - 1])
            {
                // extrapolation to the right
                switch (order)
                {
                case 1:
                    interpol = 2.0 * m_c[n - 1] * h + m_b[n - 1];
                    break;
                case 2:
                    interpol = 2.0 * m_c[n - 1];
                    break;
                default:
                    interpol = 0.0;
                    break;
                }
            }
            else
            {
                // interpolation
                switch (order)
                {
                case 1:
                    interpol = (3.0 * m_d[idx] * h + 2.0 * m_c[idx]) * h + m_b[idx];
                    break;
                case 2:
                    interpol = 6.0 * m_d[idx] * h + 2.0 * m_c[idx];
                    break;
                case 3:
                    interpol = 6.0 * m_d[idx];
                    break;
                default:
                    interpol = 0.0;
                    break;
                }
            }
            return interpol;
        }

#ifdef HAVE_SSTREAM
        std::string spline::info() const
        {
            std::stringstream ss;
            ss « "type " « m_type « ", left boundary deriv " « m_left « " = ";
            ss « m_left_value « ", right boundary deriv " « m_right « " = ";
            ss « m_right_value « std::endl;
            if (m_made_monotonic)
            {
                ss « "(spline has been adjusted for piece-wise monotonicity)";
            }
            return ss.str();
        }
#endif // HAVE_SSTREAM

        namespace internal
        {

            // band_matrix implementation
            // -------------------------

            band_matrix::band_matrix(int dim, int n_u, int n_l)
            {
                resize(dim, n_u, n_l);
            }
            void band_matrix::resize(int dim, int n_u, int n_l)
            {
                assert(dim > 0);
                assert(n_u >= 0);
                assert(n_l >= 0);
                m_upper.resize(n_u + 1);
                m_lower.resize(n_l + 1);
                for (size_t i = 0; i < m_upper.size(); i++)
                {
                    m_upper[i].resize(dim);
                }
                for (size_t i = 0; i < m_lower.size(); i++)
                {
                    m_lower[i].resize(dim);
                }
            }
            int band_matrix::dim() const
            {
                if (m_upper.size() > 0)
                {
                    return m_upper[0].size();
                }
                else
                {
                    return 0;
                }
            }

            // defines the new operator (), so that we can access the elements
            // by A(i,j), index going from i=0,...,dim()-1
            double &band_matrix::operator()(int i, int j)
            {
                int k = j - i; // what band is the entry
                assert((i >= 0) && (i < dim()) && (j >= 0) && (j < dim()));
                assert((-num_lower() <= k) && (k <= num_upper()));
                // k=0 -> diagonal, k<0 lower left part, k>0 upper right part
                if (k >= 0)
                    return m_upper[k][i];
                else
                    return m_lower[-k][i];
            }
            double band_matrix::operator()(int i, int j) const
            {
                int k = j - i; // what band is the entry
                assert((i >= 0) && (i < dim()) && (j >= 0) && (j < dim()));
                assert((-num_lower() <= k) && (k <= num_upper()));
                // k=0 -> diagonal, k<0 lower left part, k>0 upper right part
                if (k >= 0)
                    return m_upper[k][i];
                else
                    return m_lower[-k][i];
            }
            // second diag (used in LU decomposition), saved in m_lower
            double band_matrix::saved_diag(int i) const
            {
                assert((i >= 0) && (i < dim()));
                return m_lower[0][i];
            }
            double &band_matrix::saved_diag(int i)
            {
                assert((i >= 0) && (i < dim()));
                return m_lower[0][i];
            }

            // LR-Decomposition of a band matrix
            void band_matrix::lu_decompose()
            {
                int i_max, j_max;
                int j_min;
                double x;

                // preconditioning
                // normalize column i so that a_ii=1
                for (int i = 0; i < this->dim(); i++)
                {
                    assert(this->operator()(i, i) != 0.0);
                    this->saved_diag(i) = 1.0 / this->operator()(i, i);
                    j_min = std::max(0, i - this->num_lower());
                    j_max = std::min(this->dim() - 1, i + this->num_upper());
                    for (int j = j_min; j <= j_max; j++)
                    {
                        this->operator()(i, j) *= this->saved_diag(i);
                    }
                    this->operator()(i, i) = 1.0; // prevents rounding errors
                }

                // Gauss LR-Decomposition
                for (int k = 0; k < this->dim(); k++)
                {
                    i_max = std::min(this->dim() - 1, k + this->num_lower()); // num_lower not a mistake!
                    for (int i = k + 1; i <= i_max; i++)
                    {
                        assert(this->operator()(k, k) != 0.0);
                        x = -this->operator()(i, k) / this->operator()(k, k);
                        this->operator()(i, k) = -x; // assembly part of L
                        j_max = std::min(this->dim() - 1, k + this->num_upper());
                        for (int j = k + 1; j <= j_max; j++)
                        {
                            // assembly part of R
                            this->operator()(i, j) = this->operator()(i, j) + x * this->operator()(k, j);
                        }
                    }
                }
            }
            // solves Ly=b
            std::vector<double> band_matrix::l_solve(const std::vector<double> &b) const
            {
                assert(this->dim() == (int)b.size());
                std::vector<double> x(this->dim());
                int j_start;
                double sum;
                for (int i = 0; i < this->dim(); i++)
                {
                    sum = 0;
                    j_start = std::max(0, i - this->num_lower());
                    for (int j = j_start; j < i; j++)
                        sum += this->operator()(i, j) * x[j];
                    x[i] = (b[i] * this->saved_diag(i)) - sum;
                }
                return x;
            }
            // solves Rx=y
            std::vector<double> band_matrix::r_solve(const std::vector<double> &b) const
            {
                assert(this->dim() == (int)b.size());
                std::vector<double> x(this->dim());
                int j_stop;
                double sum;
                for (int i = this->dim() - 1; i >= 0; i--)
                {
                    sum = 0;
                    j_stop = std::min(this->dim() - 1, i + this->num_upper());
                    for (int j = i + 1; j <= j_stop; j++)
                        sum += this->operator()(i, j) * x[j];
                    x[i] = (b[i] - sum) / this->operator()(i, i);
                }
                return x;
            }

            std::vector<double> band_matrix::lu_solve(const std::vector<double> &b,
                                                      bool is_lu_decomposed)
            {
                assert(this->dim() == (int)b.size());
                std::vector<double> x, y;
                if (is_lu_decomposed == false)
                {
                    this->lu_decompose();
                }
                y = this->l_solve(b);
                x = this->r_solve(y);
                return x;
            }

        } // namespace internal

```

```
    } // namespace tk

} // namespace

#pragma GCC diagnostic pop

#endif /* TK_SPLINE_H */
```

7.35 FOORT/src/Terminations.cpp File Reference

```
#include "Terminations.h"
#include "Geodesic.h"
#include "InputOutput.h"
#include <cmath>
```

Functions

- TerminationUniqueVector CreateTerminationVector (TermBitflag termflags, Geodesic ∗const theGeodesic)

  Termination helper function.

7.35.1 Function Documentation

7.35.1.1 CreateTerminationVector()

```
TerminationUniqueVector CreateTerminationVector (
            TermBitflag termflags,
            Geodesic *const theGeodesic)
```

Termination helper function.

7.36 FOORT/src/Terminations.h File Reference

```
#include "Geometry.h"
#include <cstdint>
#include <memory>
#include <vector>
#include <utility>
```

Classes

- class Termination
- class HorizonTermination
- class BoundarySphereTermination
- class TimeOutTermination
- class ThetaSingularityTermination
- class NaNTermination
- class GeneralSingularityTermination
- struct TerminationOptions
- struct HorizonTermOptions
- struct BoundarySphereTermOptions
- struct TimeOutTermOptions
- struct ThetaSingularityTermOptions
- struct NaNTermOptions
- struct GeneralSingularityTermOptions

Typedefs

- using TermBitflag = std::uint16_t
- using TerminationUniqueVector = std::vector<std::unique_ptr<Termination>>

Enumerations

- enum class Term {
  Uninitialized = -1 , Continue = 0 , Horizon , BoundarySphere ,
  TimeOut , ThetaSingularity , NaN , GeneralSingularity ,
  Maxterms }

Functions

- TerminationUniqueVector CreateTerminationVector (TermBitflag termflags, Geodesic ∗const theGeodesic)

  Termination helper function.

Variables

- constexpr TermBitflag Term_None {0b0000'0000'0000'0000}
- constexpr TermBitflag Term_BoundarySphere {0b0000'0000'0000'0001}
- constexpr TermBitflag Term_TimeOut {0b0000'0000'0000'0010}
- constexpr TermBitflag Term_Horizon {0b0000'0000'0000'0100}
- constexpr TermBitflag Term_ThetaSingularity {0b0000'0000'0000'1000}
- constexpr TermBitflag Term_NaN {0b0000'0000'0001'0000}
- constexpr TermBitflag Term_GeneralSingularity {0b0000'0000'0010'0000}

7.36.1 Typedef Documentation

7.36.1.1 TermBitflag

```
using TermBitflag = std::uint16_t
```

7.36.1.2 TerminationUniqueVector

```
using TerminationUniqueVector = std::vector<std::unique_ptr<Termination>>
```

7.36.2 Enumeration Type Documentation

7.36.2.1 Term

```
enum class Term  [strong]
```

Enumerator

Uninitialized	
Continue	
Horizon	
BoundarySphere	
TimeOut	
ThetaSingularity	
NaN	
GeneralSingularity	
Maxterms	

7.36.3 Function Documentation

7.36.3.1 CreateTerminationVector()

```
TerminationUniqueVector CreateTerminationVector (
             TermBitflag termflags,
             Geodesic *const theGeodesic)
```

Termination helper function.

7.36.4 Variable Documentation

7.36.4.1 Term_BoundarySphere

```
TermBitflag Term_BoundarySphere {0b0000'0000'0000'0001}  [constexpr]
```

7.36.4.2 Term_GeneralSingularity

```
TermBitflag Term_GeneralSingularity {0b0000'0000'0010'0000}  [constexpr]
```

7.36.4.3 Term_Horizon

```
TermBitflag Term_Horizon {0b0000'0000'0000'0100}  [constexpr]
```

7.36.4.4 Term_NaN

```
TermBitflag Term_NaN {0b0000'0000'0001'0000}  [constexpr]
```

7.36.4.5 Term_None

```
TermBitflag Term_None {0b0000'0000'0000'0000}  [constexpr]
```

7.36.4.6 Term_ThetaSingularity

```
TermBitflag Term_ThetaSingularity {0b0000'0000'0000'1000}  [constexpr]
```

7.36.4.7 Term_TimeOut

```
TermBitflag Term_TimeOut {0b0000'0000'0000'0010}  [constexpr]
```

7.37 Terminations.h

Go to the documentation of this file.

```
#ifndef _FOORT_TERMINATIONS_H
#define _FOORT_TERMINATIONS_H


#include "Geometry.h" // for basic tensor objects

#include <cstdint> // for std::uint16_t
#include <memory>  // for std::unique_ptr
#include <vector>  // for std::vector
#include <utility> // for std::pair

// Forward declaration of Geodesic class needed here, since Diagnostics are passed a pointer to their owner Geodesic
// (note "Geodesic.h" is NOT included to avoid header loop, and we do not need Geodesic member functions here!)
class Geodesic;

// Used for constructing vector of Terminations
// Note that this means every Termination is either "on" or "off";
// it is not possible to have a Termination "on" more than once
using TermBitflag = std::uint16_t;

// Define a bitflag per existing Termination
constexpr TermBitflag Term_None{0b0000’0000’0000’0000};
constexpr TermBitflag Term_BoundarySphere{0b0000’0000’0000’0001};
constexpr TermBitflag Term_TimeOut{0b0000’0000’0000’0010};
constexpr TermBitflag Term_Horizon{0b0000’0000’0000’0100};
constexpr TermBitflag Term_ThetaSingularity{0b0000’0000’0000’1000};
constexpr TermBitflag Term_NaN{0b0000’0000’0001’0000};
constexpr TermBitflag Term_GeneralSingularity{0b0000’0000’0010’0000};

// Add a TermBitflag for your new Termination. Make sure you use a bitflag that has not been used before!
// Sample code:
/*
constexpr TermBitflag Term_MyTerm               { 0b0000’0000’0000’1000 };
*/


// Possible termination conditions that can be set by Terminations
enum class Term
{
    Uninitialized = -1, // Geodesic has not been properly initialized yet with initial position/velocity
    Continue = 0,       // All is right, continue integrating geodesic
    Horizon,         // STOP, encountered horizon (set by HorizonTermination)
    BoundarySphere,       // STOP, encountered boundary sphere (set by BoundarySphereTermination)
    TimeOut,         // STOP, taken too many steps (set by TimeOutTermination)
    ThetaSingularity,   // STOP, too close to polar coordinate singularity (theta = 0 or theta = pi/2)
    NaN,             // STOP, NaN encountered in geodesic position or velocity
    GeneralSingularity, // STOP, singularity encountered (of any codimension)

    // Add a new Termination condition that your new Termination can set
    // Sample code:
    /*
    MyTermCond,         // STOP, encountered (...)
    */

    Maxterms // Number of termination conditions that exist
};


// Abstract base class for all Terminations
class Termination
{
public:
    // Constructor must initialize the pointer to its owner Geodesic
    Termination() = delete;
    Termination(Geodesic *const theGeodesic) : m_OwnerGeodesic{theGeodesic}
    {
    }

    // Resets Termination object. This is called when the owner Geodesic is reset in order to start integrating
    // a new geodesic.
    // The base class implementation only resets m_StepsSinceUpdated
    // Descendants can override this if they need to reset additional internal variables
    virtual void Reset();

    // virtual destructor to ensure correct destruction of descendants
    virtual ~Termination() = default;

    // Function that is called to determine whether Termination wants to
    // terminate the Geodesic. Returns Term::Continue if no termination wanted,
    // otherwise it returns the appropriate Term condition
    virtual Term CheckTermination() = 0;

    // This returns the full description of the Termination
    virtual std::string getFullDescriptionStr() const = 0;

protected:
    // The geodesic that owns the Termination (a const pointer to the Geodesic)
    Geodesic *const m_OwnerGeodesic;

    // Helper function to decide if the Termination should indeed update its status, based on
    // UpdateNSteps (which is set to 0 if we always update)
    bool DecideUpdate(largecounter UpdateNSteps);

    // The termination is itself in charge of keeping track of how many steps it has been since it has been updated
    // The Termination's TerminationOptions struct tells it how many steps it needs to wait between updates
    largecounter m_StepsSinceUpdated{};
};

// The owner vector of derived Termination classes
using TerminationUniqueVector = std::vector<std::unique_ptr<Termination»;

// Helper to create a new vector of Termination options, based on the bitflag
TerminationUniqueVector CreateTerminationVector(TermBitflag termflags, Geodesic *const theGeodesic);


// Forward declaration needed before Termination
struct HorizonTermOptions;
// Horizon termination: terminate geodesics if they get too close to the horizon (returns Term::Horizon)
class HorizonTermination final : public Termination
{
public:
    // Basic constructor only passes on Geodesic pointer to base class constructor
    HorizonTermination(Geodesic *const theGeodesic) : Termination(theGeodesic) {}

    // Check if we are too close to the horizon
    Term CheckTermination() final;

    // Description string
    std::string getFullDescriptionStr() const final;

    // Options (contains horizon radius, if we are using logarithmic r coordinate, and distance allowed from the horizon)
    static std::unique_ptr<HorizonTermOptions> TermOptions;
};

// Forward declaration needed before Termination
struct BoundarySphereTermOptions;
// The Boundary Sphere: this terminates the geodesic (and returns Term::BoundarySphere) if
// the geodesic reaches outside of the boundary sphere
class BoundarySphereTermination final : public Termination
{
public:
    // Basic constructor only passes on Geodesic pointer to base class constructor
    BoundarySphereTermination(Geodesic *const theGeodesic) : Termination(theGeodesic) {}

    // Check if we have passed the boundary sphere
    Term CheckTermination() final;

    // Description string
    std::string getFullDescriptionStr() const final;

    // The options that the BoundarySphereTermination keeps (contains the radius of the boundary sphere)
    static std::unique_ptr<BoundarySphereTermOptions> TermOptions;
};

// Forward declaration needed before Termination
struct TimeOutTermOptions;
// The Time Out: this terminates the geodesic if too many steps have been
// taken in its integration (and returns Term::TimeOut)
class TimeOutTermination final : public Termination
{
public:
    // Basic constructor only passes on Geodesic pointer to base class constructor
    TimeOutTermination(Geodesic *const theGeodesic) : Termination(theGeodesic) {}

    // This descendant needs to override Reset in order to also reset m_CurNrSteps
    void Reset() final;

    // Check if we have already taken too many steps
    Term CheckTermination() final;

    // Description string
    std::string getFullDescriptionStr() const final;

    // The options that the TimeOutTermination keeps (contains max number of steps allowed)
    static std::unique_ptr<TimeOutTermOptions> TermOptions;

private:
    // Keep track of the number of steps that the geodesic has taken so far
    largecounter m_CurNrSteps{0};
};

// Forward declaration needed before Termination
struct ThetaSingularityTermOptions;
class ThetaSingularityTermination final : public Termination
{
public:
    ThetaSingularityTermination(Geodesic *const theGeodesic) : Termination(theGeodesic) {}

    // No override of Reset() necessary

    // Check the specific termination condition
    Term CheckTermination() final;

    // Description string
    std::string getFullDescriptionStr() const final;

    // The options that the Termination keeps (will probably be a descendant struct instead, which specifies
    // any additional options the Termination needs)
    static std::unique_ptr<ThetaSingularityTermOptions> TermOptions;
};

// Forward declaration needed before Termination
struct NaNTermOptions;
// NaN termination: terminate geodesics if position or velocity contains a nan
class NaNTermination final : public Termination
{
public:
    // Basic constructor only passes on Geodesic pointer to base class constructor
    NaNTermination(Geodesic *const theGeodesic) : Termination(theGeodesic) {}

    // Check if we are too close to the horizon
    Term CheckTermination() final;

    // Description string
    std::string getFullDescriptionStr() const final;

    // Options (contains horizon radius, if we are using logarithmic r coordinate, and distance allowed from the horizon)
    static std::unique_ptr<NaNTermOptions> TermOptions;
};

// Forward declaration needed before Termination
struct GeneralSingularityTermOptions;
// General singularity: terminate geodesic if it comes too close to one of a given number of (arbitrary codimension) singularities
class GeneralSingularityTermination final : public Termination
{
public:
    // Basic constructor only passes on Geodesic pointer to base class constructor
    GeneralSingularityTermination(Geodesic *const theGeodesic) : Termination(theGeodesic) {}

    // Check if we are too close to the horizon
    Term CheckTermination() final;

    // Description string
    std::string getFullDescriptionStr() const final;

    // Options (contains horizon radius, if we are using logarithmic r coordinate, and distance allowed from the horizon)
    static std::unique_ptr<GeneralSingularityTermOptions> TermOptions;

private:
    std::string SingularityToString(int singnr) const;
};

// Declare your Termination class here, inheriting from Termination.
// Sample code:
/*
// Forward declaration needed before Termination
struct TerminationOptions; // possibly will instead need to declare descendant options struct
class MyTermination final : public Termination // good practice to make the class final unless descendant classes are possible
{
public:
    // Constructor must at least pass on Geodesic pointer to base class constructor
    MyTermination(Geodesic* const theGeodesic) : Termination(theGeodesic) {}

    // Do you need to reset any internal variables specific to MyTermination? If so, override Reset() (This is not mandatory)
    // Note: make sure to call the base class implementation Termination::Reset()
    // from within your implementation of MyTermination::Reset(), so that the base class internal variable is also reset!
    void Reset() final;

    // Check the specific termination condition
    Term CheckTermination() final;

    // Description string
    std::string getFullDescriptionStr() const final;

    // The options that the Termination keeps (will probably be a descendant struct instead, which specifies
    // any additional options the Termination needs)
    static std::unique_ptr<TerminationOptions> TermOptions;

private:
    // any private member variables that are needed to keep track of things
};
*/


// Base class for TerminationOptions. Other Terminations can inherit from here if they require more options.
struct TerminationOptions
{
public:
    // Basic constructor only sets the number of steps between updates
    TerminationOptions(largecounter Nsteps) : UpdateEveryNSteps{Nsteps}
    {
    }

    // virtual destructor to ensure correct destruction of descendants
    virtual ~TerminationOptions() = default;

    const largecounter UpdateEveryNSteps;
};

// Options class for HorizonTermination; keeps track of location of horizon radius and the epsilon to terminate away from the horizon
struct HorizonTermOptions : public TerminationOptions
{
public:
    HorizonTermOptions(real theHorizonRadius, bool therLogScale, real theAtHorizonEps, largecounter Nsteps) : HorizonRadius{theHorizonRadius}, AtHorizonEps{theAtHorizonEps}, rLogScale{therLogScale}, TerminationOptions(Nsteps)
    {
    }

    const real HorizonRadius;
    const real AtHorizonEps;
    const bool rLogScale;
};

// Options class for BoundarySphere; has to keep track of the BoundarySphere's radius
struct BoundarySphereTermOptions : public TerminationOptions
{
public:
    BoundarySphereTermOptions(real theRadius, bool therLogScale, largecounter Nsteps) : SphereRadius{theRadius}, rLogScale{therLogScale},

TerminationOptions(Nsteps)
    {
    }

    const real SphereRadius;
    const bool rLogScale;
};

// Options class for TimeOut; has to keep track of the max. number of integration steps allowed
struct TimeOutTermOptions : public TerminationOptions
{
public:
    TimeOutTermOptions(largecounter MaxStepsAllowed, largecounter Nsteps) : MaxSteps{MaxStepsAllowed}, TerminationOptions(Nsteps)
    {
    }

    const largecounter MaxSteps;
};

// Options class for ThetaSingularityTermination
struct ThetaSingularityTermOptions : public TerminationOptions
```

```
{
public:
    ThetaSingularityTermOptions(real epsilon, largecounter Nsteps) : ThetaSingEpsilon{epsilon}, TerminationOptions(Nsteps)
    {
    }

    const real ThetaSingEpsilon;
};

// Options class for TimeOut; has to keep track of the max. number of integration steps allowed
struct NaNTermOptions : public TerminationOptions
{
public:
    NaNTermOptions(bool consoleoutputon, largecounter Nsteps) : OutputToConsole{consoleoutputon}, TerminationOptions(Nsteps)
    {
    }

    const bool OutputToConsole;
};

struct GeneralSingularityTermOptions : public TerminationOptions
{
public:
    GeneralSingularityTermOptions(std::vector<Singularity> sings,
                                  real eps, bool consoleoutputon, bool therlogscale, largecounter Nsteps)
        : Singularities{std::move(sings)}, Epsilon{eps}, OutputToConsole{consoleoutputon},
          rLogScale{therlogscale},
          TerminationOptions(Nsteps) {}

    const std::vector<Singularity> Singularities;
    const real Epsilon;
    const bool OutputToConsole;
    const bool rLogScale;
};

// Add your new TerminationOptions struct here, inheriting from TerminationOptions (if needed)
// Sample code:
/*
struct MyTermOptions : public TerminationOptions
{
public:
    MyTermOptions(..., largecounter Nsteps) : TerminationOptions(Nsteps) //, other initialization
    {}

    // member variables (const!) here
};
*/

#endif
```

7.38 FOORT/src/Utilities.cpp File Reference

```
#include "Utilities.h"
#include <sstream>
#include <iomanip>
```

7.39 FOORT/src/Utilities.h File Reference

```
#include "Metric.h"
#include "Diagnostics.h"
#include "Terminations.h"
#include "Geodesic.h"
#include "ViewScreen.h"
#include "Integrators.h"
#include <chrono>
#include <string>
#include <vector>
```

Classes

- class Utilities::Timer

Namespaces

- namespace Utilities

Functions

- std::string Utilities::GetTimeStampString ()

  Other functions in Utilities.
- std::vector< std::string > Utilities::GetDiagNameStrings (DiagBitflag alldiags, DiagBitflag valdiag)
- std::string Utilities::GetFirstLineInfoString (const Metric ∗theMetric, const Source ∗theSource, DiagBitflag alldiags, DiagBitflag valdiag, TermBitflag allterms, const ViewScreen ∗theView)

7.40 Utilities.h

Go to the documentation of this file.

```
#ifndef _FOORT_UTILITIES_H
#define _FOORT_UTILITIES_H


// We use Metric, Diagnostic, Termination, Geodesic, ViewScreen, and Integrator declarations here
#include "Metric.h"
#include "Diagnostics.h"
#include "Terminations.h"
#include "Geodesic.h"
#include "ViewScreen.h"
#include "Integrators.h"

#include <chrono> // for timer functionality
#include <string> // for strings
#include <vector> // for std::vector

// Namespace that contains our utility functions
namespace Utilities
{
    // Timer class to keep track of elapsed time
    class Timer
    {
    private:
        // Type aliases to make accessing nested type easier
        using Clock = std::chrono::steady_clock;
        using Second = std::chrono::duration<double, std::ratio<1»;

        // begin time
        std::chrono::time_point<Clock> m_beg{Clock::now()};

    public:
        // reset begin time
        void reset();

        // returns time elapsed since begin time
        double elapsed() const;
    };

    // Returns a string of the current time (in a format that can be used to append to file names)
    std::string GetTimeStampString();

    // Helper function to get all Diagnostic Names (for outputting to files)
    std::vector<std::string> GetDiagNameStrings(DiagBitflag alldiags, DiagBitflag valdiag);

    // This returns the full string to be written to every output file as its first line
    // It contains information about all the settings used to produce the output
    std::string GetFirstLineInfoString(const Metric *theMetric, const Source *theSource,
                                       DiagBitflag alldiags, DiagBitflag valdiag, TermBitflag allterms, const ViewScreen *theView);
}

#endif
```

7.41 FOORT/src/ViewScreen.cpp File Reference

```
#include "ViewScreen.h"
#include <algorithm>
```

7.42 FOORT/src/ViewScreen.h File Reference

```
#include "Geometry.h"
#include "Metric.h"
#include "Mesh.h"
#include <memory>
#include <utility>
#include <array>
#include <string>
```

Classes

- class ViewScreen

Enumerations

- enum class GeodesicType { Null = 0 , Timelike = -1 , Spacelike = 1 }

7.42.1 Enumeration Type Documentation

7.42.1.1 GeodesicType

```
enum class GeodesicType  [strong]
```

Enumerator

Null	
Timelike	
Spacelike	

7.43 ViewScreen.h

Go to the documentation of this file.

```
#ifndef _FOORT_VIEWSCREEN_H
#define _FOORT_VIEWSCREEN_H


#include "Geometry.h" // For basic tensor objects
#include "Metric.h"   // For the Metric object
#include "Mesh.h"   // For the Mesh object

#include <memory>  // std::unique_ptr
#include <utility> // std::move
#include <array>   // std::array
#include <string>  // strings

// Type of geodesic being integrated. NOTE: only Null supported/implemented at the moment!
enum class GeodesicType
{
    Null = 0,
    Timelike = -1,
    Spacelike = 1,
};

// ViewScreen class: this class is in charge of converting a pixel on the screen (which the Mesh wants to integrate)
// to physical initial conditions for the position and velocity of a geodesic. It owns a Mesh instance, which will tell it
// which pixels to integrate etc.
class ViewScreen
{
public:
    // No default constructor possible
    ViewScreen() = delete;
    // constructor must pass following arguments along:
    // - physical position and looking direction;
    // - screen dimensions (in dimensions of length)
    // - the Mesh used (ViewScreen must become a owner of this object!)
    // - the Metric used (ViewScreen is NOT the owner of the Metric)
    // - the geodesic type to be integrated (null, timelike, spacelike)
    ViewScreen(Point pos, OneIndex dir, ScreenPoint screensize, ScreenPoint screencenter,
               std::unique_ptr<Mesh> theMesh, const Metric *const theMetric, GeodesicType thegeodtype = GeodesicType::Null)
        : m_Pos{pos}, m_Direction{dir}, m_ScreenSize{screensize}, m_ScreenCenter{screencenter},
          m_theMesh{std::move(theMesh)},
          m_theMetric{theMetric}, m_GeodType{thegeodtype},
          m_rLogScale{theMetric->getrLogScale()}
    {
        // At the moment, we don't even use the direction; we are always pointed towards the origin
        if (m_Direction != Point{0, -1, 0, 0})
        {
            ScreenOutput("ViewScreen is only supported pointing inwards at the moment; Direction = {0, -1, 0, 0} will be used",
                         OutputLevel::Level_0_WARNING);
        }
        // At the moment, we are only integrating null geodesics
        if (m_GeodType != GeodesicType::Null)
        {
            ScreenOutput("ViewScreen only supports null geodesics at the moment; geodesics integrated will be null.",
                         OutputLevel::Level_0_WARNING);
        }

        // Construct the vielbein now
        ConstructVielbein();
    }

    // Heart of the ViewScreen: here, the ViewScreen is asked to provide initial conditions
    // for the geodesic nr index of the current iteration; based on the screen index
    // that the Mesh gives, it sets up these physical initial conditions.
    void SetNewInitialConditions(largecounter index, Point &pos, OneIndex &vel, ScreenIndex &scrIndex) const;

    // These member functions essentially pass on information to/from the Mesh
    bool IsFinished() const;              // Does the ViewScreen (i.e. the Mesh) want to integrate more geodesics or not?
    largecounter getCurNrGeodesics() const; // Current number of geodesics in this iteration
    void EndCurrentLoop();                    // The current iteration of geodesics is finished; prepare the next one
    // NOTE: despite not being const, this function has been designed to be threadsafe!
    void GeodesicFinished(largecounter index, std::vector<real> finalValues); // This geodesic has been integrated, returning its final "values"

    // Description string getter (spaces allowed), also will contain information about the Mesh
    std::string getFullDescriptionStr() const;

private:
    // The metric at the position of the viewscreen (we only need indices down)
    TwoIndex m_Metric_dd{};
    // The vielbein used to transform from the curved spacetime at the viewscreen to a locally flat frame
    TwoIndex m_Vielbein{};

    // Helper function to construct the vielbein given the metric
    void ConstructVielbein();

    // The position and looking direction of the camera
    const Point m_Pos;
    const OneIndex m_Direction;
    // The screensize (in physical units of length)
    const ScreenPoint m_ScreenSize;
    // The screen center
    const ScreenPoint m_ScreenCenter;

    // Whether the metric uses a logarithmic r coordinate or not
    const bool m_rLogScale;

    // const pointer to const Metric
    const Metric *const m_theMetric;

    // The geodesic type to be integrated
    const GeodesicType m_GeodType{GeodesicType::Null};

    // The const pointer to the Mesh we are using to determine pixels to be integrated
    const std::unique_ptr<Mesh> m_theMesh;
};

#endif
```

Index