

Rocket End Burning Simulator

Generated by Doxygen 1.9.2

Chapter 1

Todo List

Namespace `CPGF::Basics`

include length and area calculation.

Class `CPGF::Basics::BezierStroke2d`

Finish implementation.

Member `CPGF::PGFConf::dash_pattern`

Explain this better.

Member `CPGF::PGFConf::dash_phase`

Explain this.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chemistry

The objects of this library are used to store constants and to compute the speed of Chemical Reactions ??

CPGF

Every component of this library is part of the CTikZ namespace ??

CPGF::AffineSpace

This namespace contains all objects related to a mathematical affine space, i.e., vectors and points ??

CPGF::Basics

. ??

Math

Namespace that includes mathematical objects such as vectors, matrixes, dual numbers and useful numerical methods such as numerical integrators, ode solvers and solvers for algebraic equations ??

Math::AlgebraicSolvers

Contains solvers for algebraic equations ??

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

Math::Interpolation::AverageLinearInterpolation	??
Math::Interpolation::AverageQuadraticInterpolation	??
CPGF::Plot2d::Axis	??
BaseCell	??
GasCell	??
SolidCell	??
CPGF::Color	??
CPGF::DashPatterns	??
Math::DualNumber< K >	??
Solvers::Gas::ExactRiemannSolver	??
Solvers::Gas::ExactSteadySolver	??
Utilities::FileArray< T >	??
GasBoundaryConditions	??
CPGF::Plot2d::Graphic	??
CPGF::Plot2d::GraphicObject	??
CPGF::Plot2d::DataPlot	??
CPGF::Plot2d::LinePlot	??
CPGF::Plot2d::AveragePlot	??
Mesh< Cell >::Iterator	??
Utilities::FileArray< T >::Iterator	??
Math::Interpolation::LinearInterpolation	??
Math::AlgebraicSolvers::LinearSystemSolver	??
CPGF::LineWidth	??
Math::Matrix< K >	??
Mesh< Cell >	??
Mesh< GasCell >	??
GasMesh	??
Mesh< SolidCell >	??
SolidMesh	??
CPGF::Objects2d::Object2d	??
CPGF::Objects2d::Arrow	??
CPGF::Objects2d::Circle	??
CPGF::Objects2d::Line	??
CPGF::Basics::Path2d	??

CPGF::PGFConf	??
CPGF::AffineSpace::Point2d	??
Utilities::ProgressBase	??
Utilities::Progress< T >	??
Math::Rational< K >	??
CPGF::Scene2d	??
CPGF::Plot2d::Shapes	??
CPGF::Basics::SimpleStroke2d	??
CPGF::Basics::BezierStroke2d	??
CPGF::Basics::StraightStroke2d	??
Simulation	??
SolidBoundaryConditions	??
Chemistry::SolidGasReaction	??
Solvers::Rocket::SteadySolver	??
CPGF::Text	??
Math::Vector< K >	??
CPGF::AffineSpace::Vector2d	??

Chapter 4

Class Index

4.1 Class List

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

CPGF::Objects2d::Arrow	??
Math::Interpolation::AverageLinearInterpolation	??
CPGF::Plot2d::AveragePlot	??
Math::Interpolation::AverageQuadraticInterpolation	??
CPGF::Plot2d::Axis	??
BaseCell	
The basic structure of the space discretization	??
CPGF::Basics::BezierStroke2d	??
CPGF::Objects2d::Circle	??
CPGF::Color	
An object used to represent an rgb color. r, g and b must all be real numbers between 0 and 1	??
CPGF::DashPatterns	
This class provides a handful of useful predefined constants to express DashPatterns	??
CPGF::Plot2d::DataPlot	??
Math::DualNumber< K >	
Represents a dual number. A mathematical object written like $x = a + b$, where $\text{ satisfies }^2=0$??
Solvers::Gas::ExactRiemannSolver	
Solves the Riemann problem exactly for the 1D Euler Equations	??
Solvers::Gas::ExactSteadySolver	??
Utilities::FileArray< T >	??
GasBoundaryConditions	
This class is used to store the boundary conditions associated to a simulation	??
GasCell	??
GasMesh	??
CPGF::Plot2d::Graphic	??
CPGF::Plot2d::GraphicObject	??
Mesh< Cell >::Iterator	
Iterator object to loop through all the cells of the mesh	??
Utilities::FileArray< T >::Iterator	??
CPGF::Objects2d::Line	??
Math::Interpolation::LinearInterpolation	??
Math::AlgebraicSolvers::LinearSystemSolver	
Solves linear systems of the form $Ax=b$ through LU decomposition with partial pivoting	??
CPGF::Plot2d::LinePlot	??
CPGF::LineWidth	
This class provides a handful of useful predefined constants to express line width	??

Math::Matrix< K >	??
Mesh< Cell >	
This object represents a collection of cells. The number of cells used can be changed. Methods for adaptive refinement are included	??
CPGF::Objects2d::Object2d	??
CPGF::Basics::Path2d	??
CPGF::PGFConf	
This object is used to store all information needed to draw a path	??
CPGF::AffineSpace::Point2d	
A point of a 2D affine space	??
Utilities::Progress< T >	??
Utilities::ProgressBase	??
Math::Rational< K >	??
CPGF::Scene2d	??
CPGF::Plot2d::Shapes	??
CPGF::Basics::SimpleStroke2d	??
Simulation	
This object is used to create some initial conditions and let the domain evolve in time	??
SolidBoundaryConditions	
Object used to store left and right boundary conditions of the solid	??
SolidCell	??
Chemistry::SolidGasReaction	
This object is used to store all properties of a solid reactant and a gas product. It is also used to compute the speed of the combustion front	??
SolidMesh	??
Solvers::Rocket::SteadySolver	
This object can be used to compute chamber and exit values for a specific rocket geometry	??
CPGF::Basics::StraightStroke2d	??
CPGF::Text	??
Math::Vector< K >	
Generic Vector over the field K with any number of components	??
CPGF::AffineSpace::Vector2d	
An object that represents a 2D vector	??

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Simulation.hpp	??
Chemistry/ Reaction.hpp	
This file contains objects used to store constants and to compute the kinetics of chemical reactions	??
CPGF/ CPGF.hpp	??
CPGF/ Scene2d.hpp	??
CPGF/AffineSpace2d/ Point2d.hpp	??
CPGF/AffineSpace2d/ Vector2d.hpp	??
CPGF/Objects2d/ BasicGeometries.hpp	??
CPGF/Objects2d/ Object2d.hpp	??
CPGF/PGFBasics/ Path2d.hpp	??
CPGF/PGFBasics/ PGFConf.hpp	??
CPGF/PGFBasics/ Strokes2d.hpp	??
CPGF/Plot2d/ Axis.hpp	??
CPGF/Plot2d/ DataPlot.hpp	??
CPGF/Plot2d/ Graphic.hpp	??
CPGF/Plot2d/ GraphicObject.hpp	??
CPGF/Plot2d/ LinePlot.hpp	??
CPGF/Text/ Text.hpp	??
Math/ AlgebraicSolvers.hpp	
This file contains functions that can be used to solve non-linear algebraic equations	??
Math/ DualNumbers.hpp	
This file implements dual numbers in one variable to allow for automatic differentiation	??
Math/ Integration.hpp	??
Math/ Interpolation.hpp	??
Math/ Matrix.hpp	??
Math/ ODESolvers.hpp	??
Math/ Rational.hpp	??
Math/ Vector.hpp	
This file contains all prototypes related to class <code>Vector<K></code>	??
Mesh/ Cell.hpp	
This files contains all declarations related to the basic objects of all meshes: the cells	??
Mesh/ Mesh.hpp	
This files contains all declarations related to the mesh	??
Solvers/Gas/ ExactRiemannSolver.hpp	??

Solvers/Gas/ ExactSteadySolver.hpp	??
Solvers/Gas/ GasSolvers.hpp	??
Solvers/Rocket/ RocketSolver.hpp	??
Solvers/Solid/ SolidSolvers.hpp	??
Utilities/ FileArray.hpp	??
Utilities/ FileOperations.hpp	??
Utilities/ FormatNumber.hpp	??
Utilities/ Progress.hpp	??
Utilities/ ToString.hpp	??

Chapter 6

Namespace Documentation

6.1 Chemistry Namespace Reference

The objects of this library are used to store constants and to compute the speed of Chemical Reactions.

Classes

- class [SolidGasReaction](#)

This object is used to store all properties of a solid reactant and a gas product. It is also used to compute the speed of the combustion front.

6.1.1 Detailed Description

The objects of this library are used to store constants and to compute the speed of Chemical Reactions.

6.2 CPGF Namespace Reference

Every component of this library is part of the CTikZ namespace.

Namespaces

- namespace [AffineSpace](#)

This namespace contains all objects related to a mathematical affine space, i.e., vectors and points.

- namespace [Basics](#)

Classes

- class [Color](#)
An object used to represent an rgb color. r , g and b must all be real numbers between 0 and 1.
- class [DashPatterns](#)
This class provides a handful of useful predefined constants to express [DashPatterns](#).
- class [LineWidth](#)
This class provides a handful of useful predefined constants to express line width.
- class [PGFConf](#)
This object is used to store all information needed to draw a path.
- class [Scene2d](#)
- class [Text](#)

Enumerations

- enum class [DrawType](#) { **DRAW** , **FILL** }
Determines whether a path has to be drawn (only the contour is drawn and the interior remains white) or to be filled (the contour is not drawn but the interior is).
- enum class [LineCap](#) { **ROUND** , **RECT** , **BUTT** }
This determines how the extremes of an open path are drawn.
- enum class [LineJoin](#) { **ROUND** , **BEVEL** , **MITER** }
This determines how the junction between two non-parallel lines is drawn.
- enum class [TextAlignment](#) {
 [CENTER](#) , [LEFT](#) , [RIGHT](#) , [TOP](#) ,
 [BOTTOM](#) , [BASE](#) , [TOP_LEFT](#) , [TOP_RIGHT](#) ,
 [BOTTOM_LEFT](#) , [BOTTOM_RIGHT](#) , [BASE_LEFT](#) , [BASE_RIGHT](#) }
Part of the following comments are copied from the TikZ/PGF manual.

6.2.1 Detailed Description

Every component of this library is part of the CTikZ namespace.

6.2.2 Enumeration Type Documentation

6.2.2.1 TextAlignment

```
enum class CPGF::TextAlignment [strong]
```

Part of the following comments are copied from the TikZ/PGF manual.

Enumerator

CENTER	The default TextAlignment.
LEFT	The key causes the text box to be placed such that its left border is on the origin.
RIGHT	The key causes the text box to be placed such that its right border is on the origin.
TOP	This key causes the text box to be placed such that its top is on the origin.

Enumerator

BOTTOM	This key causes the text box to be placed such that its bottom is on the origin.
BASE	This key causes the text box to be placed such that its baseline is on the origin.
TOP_LEFT	This key causes the text box to be placed such that its top left corner is on the origin.
TOP_RIGHT	This key causes the text box to be placed such that its top right corner is on the origin.
BOTTOM_LEFT	This key causes the text box to be placed such that its bottom left corner is on the origin.
BOTTOM_RIGHT	This key causes the text box to be placed such that its bottom right corner is on the origin.
BASE_LEFT	This key causes the text box to be placed such that its base left corner is on the origin.
BASE_RIGHT	This key causes the text box to be placed such that its base right corner is on the origin.

6.3 CPGF::AffineSpace Namespace Reference

This namespace contains all objects related to a mathematical affine space, i.e., vectors and points.

Classes

- class [Point2d](#)
A point of a 2D affine space.
- class [Vector2d](#)
An object that represents a 2D vector.

Functions

- [Point2d operator+](#) (const [Point2d](#) &P, const [Vector2d](#) &v)
- [Vector2d operator-](#) (const [Point2d](#) &P, const [Point2d](#) &Q)
- bool [operator==](#) (const [Point2d](#) &P, const [Point2d](#) &Q)
- [Vector2d operator+](#) (const [Vector2d](#) &v, const [Vector2d](#) &w)
- [Vector2d operator-](#) (const [Vector2d](#) &v, const [Vector2d](#) &w)
- [Vector2d operator*](#) (const [Vector2d](#) &v, const [Vector2d](#) &w)
- [Vector2d operator/](#) (const [Vector2d](#) &v, const [Vector2d](#) &w)
- double [operator|](#) (const [Vector2d](#) &v, const [Vector2d](#) &w)

6.3.1 Detailed Description

This namespace contains all objects related to a mathematical affine space, i.e., vectors and points.

6.3.2 Function Documentation

6.3.2.1 operator*()

```
Vector2d CPGF::AffineSpace::operator* (
    const Vector2d & v,
    const Vector2d & w )
```

Parameters

v	a 2D vector.
w	a 2D vector.

Returns

[Vector2d](#)

6.3.2.2 operator+() [1/2]

```
Point2d CPGF::AffineSpace::operator+ (
    const Point2d & P,
    const Vector2d & v )
```

Returns the position of the end of the vector v when its start is placed at the point P .

Parameters

P	a 2D point.
v	a 2D vector.

Returns

Point2D

6.3.2.3 operator+() [2/2]

```
Vector2d CPGF::AffineSpace::operator+ (
    const Vector2d & v,
    const Vector2d & w )
```

Parameters

v	a 2D vector.
w	a 2D vector.

Returns

[Vector2d](#)

6.3.2.4 operator-() [1/2]

```
Vector2d CPGF::AffineSpace::operator- (
    const Point2d & P,
    const Point2d & Q )
```

Returns the vector whose start is at point P and whose end is at point Q.

Parameters

<i>P</i>	a 2D point.
<i>Q</i>	a 2D point.

Returns

Vector2d

6.3.2.5 operator-() [2/2]

```
Vector2d CPGF::AffineSpace::operator- (
    const Vector2d & v,
    const Vector2d & w )
```

Parameters

<i>v</i>	a 2D vector.
<i>w</i>	a 2D vector.

Returns

Vector2d

6.3.2.6 operator/()

```
Vector2d CPGF::AffineSpace::operator/ (
    const Vector2d & v,
    const Vector2d & w )
```

Parameters

<i>v</i>	a 2D vector.
<i>w</i>	a 2D vector.

Returns

[Vector2d](#)**6.3.2.7 operator==()**

```
bool CPGF::AffineSpace::operator== (
    const Point2d & P,
    const Point2d & Q )
```

Parameters

<i>P</i>	
<i>Q</i>	

Returns

true

false

6.3.2.8 operator" | ()

```
double CPGF::AffineSpace::operator| (
    const Vector2d & v,
    const Vector2d & w )
```

Parameters

<i>v</i>	a 2D vector.
<i>w</i>	a 2D vector.

Returns

double

6.4 CPGF::Basics Namespace Reference**Classes**

- class [BezierStroke2d](#)
- class [Path2d](#)
- class [SimpleStroke2d](#)
- class [StraightStroke2d](#)

6.4.1 Detailed Description

Todo include length and area calculation.

6.5 Math Namespace Reference

Namespace that includes mathematical objects such as vectors, matrixes, dual numbers and useful numerical methods such as numerical integrators, ode solvers and solvers for algebraic equations.

Namespaces

- namespace [AlgebraicSolvers](#)
Contains solvers for algebraic equations.

Classes

- class [DualNumber](#)
Represents a dual number. A mathematical object written like $x = a + b$, where $a^2 = 0$.
- class [Matrix](#)
- class [Rational](#)
- class [Vector](#)
Generic [Vector](#) over the field K with any number of components.

Functions

- `template<typename K>`
`DualNumber< K > operator+ (const DualNumber< K > &x, const DualNumber< K > &y)`
- `template<typename K>`
`DualNumber< K > operator+ (const DualNumber< K > &x, const K y)`
- `template<typename K>`
`DualNumber< K > operator+ (const K x, const DualNumber< K > &y)`
- `template<typename K>`
`DualNumber< K > operator- (const DualNumber< K > &x, const DualNumber< K > &y)`
- `template<typename K>`
`DualNumber< K > operator- (const DualNumber< K > &x, const K y)`
- `template<typename K>`
`DualNumber< K > operator- (const K x, const DualNumber< K > &y)`
- `template<typename K>`
`DualNumber< K > operator* (const DualNumber< K > &x, const DualNumber< K > &y)`
- `template<typename K>`
`DualNumber< K > operator* (const DualNumber< K > &x, const K y)`
- `template<typename K>`
`DualNumber< K > operator* (const K x, const DualNumber< K > &y)`
- `template<typename K>`
`DualNumber< K > operator/ (const DualNumber< K > &x, const DualNumber< K > &y)`
- `template<typename K>`
`DualNumber< K > operator/ (const DualNumber< K > &x, const K y)`
- `template<typename K>`
`DualNumber< K > operator/ (const K x, const DualNumber< K > &y)`

- `template<typename K >`
`DualNumber< K > cos` (const `DualNumber< K > &x`)
*Returns $\cos(x.a) - x.b * \sin(x.a)$*
- `template<typename K >`
`DualNumber< K > sin` (const `DualNumber< K > &x`)
*Returns $\sin(x.a) + x.b * \cos(x.a)$*
- `template<typename K >`
`DualNumber< K > tan` (const `DualNumber< K > &x`)
*Returns $\tan(x.a) + x.b / \cos(x.a) / \cos(x.a) *$*
- `template<typename K >`
`DualNumber< K > acos` (const `DualNumber< K > &x`)
*Returns $\arccos(x.a) - 1 / \sqrt{1 - x.a * x.a} * x.b *$*
- `template<typename K >`
`DualNumber< K > asin` (const `DualNumber< K > &x`)
*Returns $\arcsin(x.a) + 1 / \sqrt{1 - x.a * x.a} * x.b *$*
- `template<typename K >`
`DualNumber< K > atan` (const `DualNumber< K > &x`)
*Returns $\arctan(x.a) + 1 / (1 + x.a * x.a) * x.b *$*
- `template<typename K >`
`DualNumber< K > cosh` (const `DualNumber< K > &x`)
*Returns $\cosh(x.a) + \sinh(x.a) * x.b *$*
- `template<typename K >`
`DualNumber< K > sinh` (const `DualNumber< K > &x`)
*Returns $\sinh(x.a) + \cosh(x.a) * x.b *$*
- `template<typename K >`
`DualNumber< K > tanh` (const `DualNumber< K > &x`)
*Returns $\tanh(x.a) + 1 / \cosh(x.a) / \cosh(x.a) * x.b *$*
- `template<typename K >`
`DualNumber< K > acosh` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > asinh` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > atanh` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > exp` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > log` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > log10` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > exp2` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > expm1` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > log1p` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > log2` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > logb` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > pow` (const `DualNumber< K > &x`, const `DualNumber< K > &y`)
- `template<typename K >`
`DualNumber< K > pow` (const `DualNumber< K > &x`, const `K &y`)
- `template<typename K >`
`DualNumber< K > pow` (const `K &x`, const `DualNumber< K > &y`)

- [illegible]

- template [Vector](#)< double > **asinh** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **atanh** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **exp** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **frexp** (const [Vector](#)< double > &v, [Vector](#)< int > *exp)
- template [Vector](#)< double > **ldexp** (const [Vector](#)< double > &v, const int exp)
- template [Vector](#)< double > **ldexp** (const [Vector](#)< double > &v, const [Vector](#)< int > &exp)
- template [Vector](#)< double > **log** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **log10** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **modf** (const [Vector](#)< double > &v, [Vector](#)< double > *intpart)
- template [Vector](#)< double > **exp2** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **expm1** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **ilogb** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **log1p** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **log2** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **logb** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **scalbn** (const [Vector](#)< double > &v, const int n)
- template [Vector](#)< double > **scalbn** (const [Vector](#)< double > &v, const [Vector](#)< int > &n)
- template [Vector](#)< double > **scalbln** (const [Vector](#)< double > &v, const long int n)
- template [Vector](#)< double > **pow** (const double v, const [Vector](#)< double > &exponent)
- template [Vector](#)< double > **pow** (const [Vector](#)< double > &v, const double exponent)
- template [Vector](#)< double > **pow** (const [Vector](#)< double > &v, const [Vector](#)< double > &exponent)
- template [Vector](#)< double > **sqrt** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **cbrt** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **hypot** (const double x, const [Vector](#)< double > &y)
- template [Vector](#)< double > **hypot** (const [Vector](#)< double > &x, const double y)
- template [Vector](#)< double > **hypot** (const [Vector](#)< double > &x, const [Vector](#)< double > &y)
- template [Vector](#)< double > **erf** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **erfc** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **tgamma** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **lgamma** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **ceil** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **floor** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **fmod** (const double numer, const [Vector](#)< double > &denom)
- template [Vector](#)< double > **fmod** (const [Vector](#)< double > &numer, const double denom)
- template [Vector](#)< double > **fmod** (const [Vector](#)< double > &numer, const [Vector](#)< double > &denom)
- template [Vector](#)< double > **trunc** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **round** (const [Vector](#)< double > &v)
- template [Vector](#)< long int > **lround** (const [Vector](#)< double > &v)
- template [Vector](#)< long long int > **llround** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **rint** (const [Vector](#)< double > &v)
- template [Vector](#)< long int > **lrint** (const [Vector](#)< double > &v)
- template [Vector](#)< long long int > **llrint** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **nearbyint** (const [Vector](#)< double > &v)
- template [Vector](#)< double > **remainder** (const double numer, const [Vector](#)< double > &denom)
- template [Vector](#)< double > **remainder** (const [Vector](#)< double > &numer, const double denom)
- template [Vector](#)< double > **remainder** (const [Vector](#)< double > &numer, const [Vector](#)< double > &denom)
- template [Vector](#)< double > **remquo** (const double numer, const [Vector](#)< double > &denom, [Vector](#)< int > *quot)
- template [Vector](#)< double > **remquo** (const [Vector](#)< double > &numer, const double denom, [Vector](#)< int > *quot)
- template [Vector](#)< double > **remquo** (const [Vector](#)< double > &numer, const [Vector](#)< double > &denom, [Vector](#)< int > *quot)
- template [Vector](#)< double > **copysign** (const [Vector](#)< double > &x, const double y)
- template [Vector](#)< double > **copysign** (const [Vector](#)< double > &x, const [Vector](#)< double > &y)
- template [Vector](#)< double > **nextafter** (const [Vector](#)< double > &x, const [Vector](#)< double > &y)

- template `Vector< float > vector_product_3d` (const `Vector< float > &v`, const `Vector< float > &w`)
- template `float min` (const `Vector< float > &v`)
- template `float max` (const `Vector< float > &v`)
- template `float sum` (const `Vector< float > &v`)
- template `float multiply` (const `Vector< float > &v`)
- template `Vector< float > cos` (const `Vector< float > &v`)
- template `Vector< float > sin` (const `Vector< float > &v`)
- template `Vector< float > tan` (const `Vector< float > &v`)
- template `Vector< float > acos` (const `Vector< float > &v`)
- template `Vector< float > asin` (const `Vector< float > &v`)
- template `Vector< float > atan` (const `Vector< float > &v`)
- template `Vector< float > atan2` (const float `v`, const `Vector< float > &w`)
- template `Vector< float > atan2` (const `Vector< float > &v`, const float `w`)
- template `Vector< float > atan2` (const `Vector< float > &v`, const `Vector< float > &w`)
- template `Vector< float > cosh` (const `Vector< float > &v`)
- template `Vector< float > sinh` (const `Vector< float > &v`)
- template `Vector< float > tanh` (const `Vector< float > &v`)
- template `Vector< float > acosh` (const `Vector< float > &v`)
- template `Vector< float > asinh` (const `Vector< float > &v`)
- template `Vector< float > atanh` (const `Vector< float > &v`)
- template `Vector< float > exp` (const `Vector< float > &v`)
- template `Vector< float > frexp` (const `Vector< float > &v`, `Vector< int > *exp`)
- template `Vector< float > ldexp` (const `Vector< float > &v`, const int `exp`)
- template `Vector< float > ldexp` (const `Vector< float > &v`, const `Vector< int > &exp`)
- template `Vector< float > log` (const `Vector< float > &v`)
- template `Vector< float > log10` (const `Vector< float > &v`)
- template `Vector< float > exp2` (const `Vector< float > &v`)
- template `Vector< float > expm1` (const `Vector< float > &v`)
- template `Vector< float > ilogb` (const `Vector< float > &v`)
- template `Vector< float > log1p` (const `Vector< float > &v`)
- template `Vector< float > log2` (const `Vector< float > &v`)
- template `Vector< float > logb` (const `Vector< float > &v`)
- template `Vector< float > scalbn` (const `Vector< float > &v`, const int `n`)
- template `Vector< float > scalbn` (const `Vector< float > &v`, const `Vector< int > &n`)
- template `Vector< float > scalbln` (const `Vector< float > &v`, const long int `n`)
- template `Vector< float > pow` (const float `v`, const `Vector< float > &exponent`)
- template `Vector< float > pow` (const `Vector< float > &v`, const float `exponent`)
- template `Vector< float > pow` (const `Vector< float > &v`, const `Vector< float > &exponent`)
- template `Vector< float > sqrt` (const `Vector< float > &v`)
- template `Vector< float > cbrt` (const `Vector< float > &v`)
- template `Vector< float > hypot` (const float `x`, const `Vector< float > &y`)
- template `Vector< float > hypot` (const `Vector< float > &x`, const float `y`)
- template `Vector< float > hypot` (const `Vector< float > &x`, const `Vector< float > &y`)
- template `Vector< float > erf` (const `Vector< float > &v`)
- template `Vector< float > erfc` (const `Vector< float > &v`)
- template `Vector< float > tgamma` (const `Vector< float > &v`)
- template `Vector< float > lgamma` (const `Vector< float > &v`)
- template `Vector< float > ceil` (const `Vector< float > &v`)
- template `Vector< float > floor` (const `Vector< float > &v`)
- template `Vector< float > fmod` (const float `numer`, const `Vector< float > &denom`)
- template `Vector< float > fmod` (const `Vector< float > &numer`, const float `denom`)
- template `Vector< float > fmod` (const `Vector< float > &numer`, const `Vector< float > &denom`)
- template `Vector< float > trunc` (const `Vector< float > &v`)
- template `Vector< float > round` (const `Vector< float > &v`)
- template `Vector< long int > lround` (const `Vector< float > &v`)

- template bool **operator>=** (const int &alpha, const [Vector](#)< int > &v)
- template bool **operator>=** (const [Vector](#)< int > &v, const [Vector](#)< int > &w)
- template bool **operator==** (const [Vector](#)< int > &v, const [Vector](#)< int > &w)
- template bool **operator==** (const [Vector](#)< int > &v, const int &alpha)
- template bool **operator==** (const int &alpha, const [Vector](#)< int > &v)
- template bool **operator!=** (const [Vector](#)< int > &v, const [Vector](#)< int > &w)
- template bool **operator!=** (const [Vector](#)< int > &v, const int &alpha)
- template bool **operator!=** (const int &alpha, const [Vector](#)< int > &v)
- template [Vector](#)< int > **operator&** (const [Vector](#)< int > &v, const [Vector](#)< int > &w)
- template [Vector](#)< int > **operator&** (const [Vector](#)< int > &v, const int &alpha)
- template [Vector](#)< int > **operator&** (const int &alpha, const [Vector](#)< int > &v)
- template int **operator|** (const [Vector](#)< int > &v, const [Vector](#)< int > &w)
- template int **vector_product_2d** (const [Vector](#)< int > &v, const [Vector](#)< int > &w)
- template [Vector](#)< int > **vector_product_3d** (const [Vector](#)< int > &v, const [Vector](#)< int > &w)
- template int **min** (const [Vector](#)< int > &v)
- template int **max** (const [Vector](#)< int > &v)
- template int **sum** (const [Vector](#)< int > &v)
- template int **multiply** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **cos** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **sin** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **tan** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **acos** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **asin** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **atan** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **atan2** (const int v, const [Vector](#)< int > &w)
- template [Vector](#)< int > **atan2** (const [Vector](#)< int > &v, const int w)
- template [Vector](#)< int > **atan2** (const [Vector](#)< int > &v, const [Vector](#)< int > &w)
- template [Vector](#)< int > **cosh** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **sinh** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **tanh** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **acosh** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **asinh** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **atanh** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **exp** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **frexp** (const [Vector](#)< int > &v, [Vector](#)< int > *exp)
- template [Vector](#)< int > **ldexp** (const [Vector](#)< int > &v, const int exp)
- template [Vector](#)< int > **ldexp** (const [Vector](#)< int > &v, const [Vector](#)< int > &exp)
- template [Vector](#)< int > **log** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **log10** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **exp2** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **expm1** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **ilogb** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **log1p** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **log2** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **logb** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **scalbn** (const [Vector](#)< int > &v, const int n)
- template [Vector](#)< int > **scalbn** (const [Vector](#)< int > &v, const [Vector](#)< int > &n)
- template [Vector](#)< int > **scalbln** (const [Vector](#)< int > &v, const long int n)
- template [Vector](#)< int > **pow** (const int v, const [Vector](#)< int > &exponent)
- template [Vector](#)< int > **pow** (const [Vector](#)< int > &v, const int exponent)
- template [Vector](#)< int > **pow** (const [Vector](#)< int > &v, const [Vector](#)< int > &exponent)
- template [Vector](#)< int > **sqrt** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **cbrt** (const [Vector](#)< int > &v)
- template [Vector](#)< int > **hypot** (const int x, const [Vector](#)< int > &y)
- template [Vector](#)< int > **hypot** (const [Vector](#)< int > &x, const int y)

- [illegible]

- template `Vector< unsigned int > nextafter` (const `Vector< unsigned int >` &x, const `Vector< unsigned int >` &y)
- template `Vector< unsigned int > fdim` (const unsigned int x, const `Vector< unsigned int >` &y)
- template `Vector< unsigned int > fdim` (const `Vector< unsigned int >` &x, unsigned int y)
- template `Vector< unsigned int > fdim` (const `Vector< unsigned int >` &x, const `Vector< unsigned int >` &y)
- template `Vector< unsigned int > fabs` (const `Vector< unsigned int >` &v)
- template `Vector< unsigned int > fma` (const unsigned int x, const `Vector< unsigned int >` &y, const `Vector< unsigned int >` &z)
- template `Vector< unsigned int > fma` (const `Vector< unsigned int >` &x, const unsigned int y, const `Vector< unsigned int >` &z)
- template `Vector< unsigned int > fma` (const `Vector< unsigned int >` &x, const `Vector< unsigned int >` &y, const unsigned int z)
- template `Vector< unsigned int > fma` (const unsigned int x, const unsigned int y, const `Vector< unsigned int >` &z)
- template `Vector< unsigned int > fma` (const unsigned int x, const `Vector< unsigned int >` &y, const unsigned int z)
- template `Vector< unsigned int > fma` (const `Vector< unsigned int >` &x, const unsigned int y, const unsigned int z)
- template `Vector< unsigned int > fma` (const `Vector< unsigned int >` &x, const `Vector< unsigned int >` &y, const `Vector< unsigned int >` &z)
- template `Vector< int > fpclassify` (const `Vector< unsigned int >` &v)
- template bool `isfinite` (const `Vector< unsigned int >` &v)
- template bool `isinf` (const `Vector< unsigned int >` &v)
- template bool `isnan` (const `Vector< unsigned int >` &v)
- template bool `isnormal` (const `Vector< unsigned int >` &v)
- template<typename K >
`Matrix< K > operator+` (const `Matrix< K >` &A, const `Matrix< K >` &B)
- template<typename K >
`Matrix< K > operator+` (const `Matrix< K >` &A, const K &alpha)
- template<typename K >
`Matrix< K > operator+` (const K &alpha, const `Matrix< K >` &A)
- template<typename K >
`Matrix< K > operator-` (const `Matrix< K >` &A, const `Matrix< K >` &B)
- template<typename K >
`Matrix< K > operator-` (const `Matrix< K >` &A, const K &alpha)
- template<typename K >
`Matrix< K > operator-` (const K &alpha, const `Matrix< K >` &A)
- template<typename K >
`Matrix< K > operator*` (const `Matrix< K >` &A, const K &alpha)
- template<typename K >
`Matrix< K > operator*` (const K &alpha, const `Matrix< K >` &A)
- template<typename K >
`Matrix< K > operator/` (const `Matrix< K >` &A, const K &alpha)
- template<typename K >
`Matrix< K > operator|` (const `Matrix< K >` &A, const `Matrix< K >` &B)
- template<typename K >
`Vector< K > operator|` (const `Matrix< K >` &A, const `Vector< K >` &v)
- template<typename K >
`Vector< K > operator|` (const `Vector< K >` &v, const `Matrix< K >` &A)
- template<typename K >
`Rational< K > & operator+` (const `Rational< K >` &p, const `Rational< K >` &q)
- template<typename K >
`Rational< K > & operator-` (const `Rational< K >` &p, const `Rational< K >` &q)
- template<typename K >
`Rational< K > & operator*` (const `Rational< K >` &p, const `Rational< K >` &q)

- `template<typename K >`
`Rational< K > & operator/ (const Rational< K > &p, const Rational< K > &q)`
- `template<typename K >`
`bool operator== (const Rational< K > &p, const Rational< K > &q)`
- `template<typename K >`
`bool operator!= (const Rational< K > &p, const Rational< K > &q)`
- `template<typename K >`
`bool operator< (const Rational< K > &p, const Rational< K > &q)`
- `template<typename K >`
`bool operator<= (const Rational< K > &p, const Rational< K > &q)`
- `template<typename K >`
`bool operator> (const Rational< K > &p, const Rational< K > &q)`
- `template<typename K >`
`bool operator>= (const Rational< K > &p, const Rational< K > &q)`
- `char conj (const char &val)`
This function exists to make the definition of the scalar product more general. It just returns val .
- `unsigned char conj (const unsigned char &val)`
This function exists to make the definition of the scalar product more general. It just returns val .
- `short int conj (const short int &val)`
This function exists to make the definition of the scalar product more general. It just returns val .
- `unsigned short int conj (const unsigned short int &val)`
This function exists to make the definition of the scalar product more general. It just returns val .
- `int conj (const int &val)`
This function exists to make the definition of the scalar product more general. It just returns val .
- `unsigned int conj (const unsigned int &val)`
This function exists to make the definition of the scalar product more general. It just returns val .
- `long int conj (const long int &val)`
This function exists to make the definition of the scalar product more general. It just returns val .
- `unsigned long int conj (const unsigned long int &val)`
This function exists to make the definition of the scalar product more general. It just returns val .
- `long long int conj (const long long int &val)`
This function exists to make the definition of the scalar product more general. It just returns val .
- `unsigned long long int conj (const unsigned long long int &val)`
This function exists to make the definition of the scalar product more general. It just returns val .
- `float conj (const float &val)`
This function exists to make the definition of the scalar product more general. It just returns val .
- `double conj (const double &val)`
This function exists to make the definition of the scalar product more general. It just returns val .
- `long double conj (const long double &val)`
This function exists to make the definition of the scalar product more general. It just returns val .
- `wchar_t conj (const wchar_t &val)`
This function exists to make the definition of the scalar product more general. It just returns val .
- `template<typename K >`
`std::complex< K > conj (const std::complex< K > &val)`
This function exists to make the definition of the scalar product more general. It just returns the complex conjugate of val .
- `template<typename K >`
`std::string to_string (const std::complex< K > &z)`
- `template<typename K >`
`Vector< K > operator+ (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`Vector< K > operator+ (const Vector< K > &v, const K &alpha)`

- Generated by Doxygen

- `template<typename K >`
`Vector< K > operator& (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`Vector< K > operator& (const Vector< K > &v, const K &w)`
- `template<typename K >`
`Vector< K > operator& (const K &v, const Vector< K > &w)`
- `template<typename K >`
`K operator| (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`K vector_product_2d (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`Vector< K > vector_product_3d (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`std::ostream & operator<< (std::ostream &os, const Vector< K > &v)`
- `template<typename K >`
`K min (const Vector< K > &v)`
- `template<typename K >`
`K max (const Vector< K > &v)`
- `template<typename K >`
`K sum (const Vector< K > &v)`
- `template<typename K >`
`K multiply (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > cos (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > sin (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > tan (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > acos (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > asin (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > atan (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > atan2 (const K v, const Vector< K > &w)`
- `template<typename K >`
`Vector< K > atan2 (const Vector< K > &v, const K w)`
- `template<typename K >`
`Vector< K > atan2 (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`Vector< K > cosh (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > sinh (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > tanh (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > acosh (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > asinh (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > atanh (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > exp (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > frexp (const Vector< K > &v, Vector< int > *exp)`

- `template<typename K >`
`Vector< K > ldexp` (const `Vector< K >` &`v`, const int `exp`)
- `template<typename K >`
`Vector< K > ldexp` (const `Vector< K >` &`v`, const `Vector< int >` &`exp`)
- `template<typename K >`
`Vector< K > log` (const `Vector< K >` &`v`)
- `template<typename K >`
`Vector< K > log10` (const `Vector< K >` &`v`)
- `template<typename K >`
`Vector< K > modf` (const `Vector< K >` &`v`, `Vector< K >` *`intpart`)
- `template<typename K >`
`Vector< K > exp2` (const `Vector< K >` &`v`)
- `template<typename K >`
`Vector< K > expm1` (const `Vector< K >` &`v`)
- `template<typename K >`
`Vector< K > ilogb` (const `Vector< K >` &`v`)
- `template<typename K >`
`Vector< K > log1p` (const `Vector< K >` &`v`)
- `template<typename K >`
`Vector< K > log2` (const `Vector< K >` &`v`)
- `template<typename K >`
`Vector< K > logb` (const `Vector< K >` &`v`)
- `template<typename K >`
`Vector< K > scalbn` (const `Vector< K >` &`v`, const int `n`)
- `template<typename K >`
`Vector< K > scalbn` (const `Vector< K >` &`v`, const `Vector< int >` &`n`)
- `template<typename K >`
`Vector< K > scalbn` (const `Vector< K >` &`v`, const long int `n`)
- `template<typename K >`
`Vector< K > scalbn` (const `Vector< K >` &`v`, const `Vector< long int >` &`n`)
- `template<typename K >`
`Vector< K > pow` (const K `v`, const `Vector< K >` &`exponent`)
- `template<typename K >`
`Vector< K > pow` (const `Vector< K >` &`v`, const K `exponent`)
- `template<typename K >`
`Vector< K > pow` (const `Vector< K >` &`v`, const `Vector< K >` &`exponent`)
- `template<typename K >`
`Vector< K > sqrt` (const `Vector< K >` &`v`)
- `template<typename K >`
`Vector< K > cbrt` (const `Vector< K >` &`v`)
- `template<typename K >`
`Vector< K > hypot` (const K `x`, const `Vector< K >` &`y`)
- `template<typename K >`
`Vector< K > hypot` (const `Vector< K >` &`x`, const K `y`)
- `template<typename K >`
`Vector< K > hypot` (const `Vector< K >` &`x`, const `Vector< K >` &`y`)
- `template<typename K >`
`Vector< K > erf` (const `Vector< K >` &`v`)
- `template<typename K >`
`Vector< K > erfc` (const `Vector< K >` &`v`)
- `template<typename K >`
`Vector< K > tgamma` (const `Vector< K >` &`v`)
- `template<typename K >`
`Vector< K > lgamma` (const `Vector< K >` &`v`)
- `template<typename K >`
`Vector< K > ceil` (const `Vector< K >` &`v`)

- `template<typename K >`
`Vector< K > floor` (const `Vector< K > &v`)
- `template<typename K >`
`Vector< K > fmod` (const K numer, const `Vector< K > &denom`)
- `template<typename K >`
`Vector< K > fmod` (const `Vector< K > &number`, const K denom)
- `template<typename K >`
`Vector< K > fmod` (const `Vector< K > &number`, const `Vector< K > &denom`)
- `template<typename K >`
`Vector< K > trunc` (const `Vector< K > &v`)
- `template<typename K >`
`Vector< K > round` (const `Vector< K > &v`)
- `template<typename K >`
`Vector< long int > lround` (const `Vector< K > &v`)
- `template<typename K >`
`Vector< long long int > llround` (const `Vector< K > &v`)
- `template<typename K >`
`Vector< K > rint` (const `Vector< K > &v`)
- `template<typename K >`
`Vector< long int > lrint` (const `Vector< K > &v`)
- `template<typename K >`
`Vector< long long int > llrint` (const `Vector< K > &v`)
- `template<typename K >`
`Vector< K > nearbyint` (const `Vector< K > &v`)
- `template<typename K >`
`Vector< K > remainder` (const K numer, const `Vector< K > &denom`)
- `template<typename K >`
`Vector< K > remainder` (const `Vector< K > &number`, const K denom)
- `template<typename K >`
`Vector< K > remainder` (const `Vector< K > &number`, const `Vector< K > &denom`)
- `template<typename K >`
`Vector< K > remquo` (const K numer, const `Vector< K > &denom`, `Vector< int > *quot`)
- `template<typename K >`
`Vector< K > remquo` (const `Vector< K > &number`, const K denom, `Vector< int > *quot`)
- `template<typename K >`
`Vector< K > remquo` (const `Vector< K > &number`, const `Vector< K > &denom`, `Vector< int > *quot`)
- `template<typename K >`
`Vector< K > copysign` (const `Vector< K > &x`, const K y)
- `template<typename K >`
`Vector< K > copysign` (const `Vector< K > &x`, const `Vector< K > &y`)
- `template<typename K >`
`Vector< K > nextafter` (const `Vector< K > &x`, const `Vector< K > &y`)
- `template<typename K >`
`Vector< K > fdim` (const K x, const `Vector< K > &y`)
- `template<typename K >`
`Vector< K > fdim` (const `Vector< K > &x`, K y)
- `template<typename K >`
`Vector< K > fdim` (const `Vector< K > &x`, const `Vector< K > &y`)
- `template<typename K >`
`Vector< double > fabs` (const `Vector< K > &v`)
- `template<typename K >`
`Vector< double > abs` (const `Vector< K > &v`)
- `template<typename K >`
`Vector< K > fma` (const K x, const `Vector< K > &y`, const `Vector< K > &z`)
- `template<typename K >`
`Vector< K > fma` (const `Vector< K > &x`, const K y, const `Vector< K > &z`)

- `template<typename K >`
`Vector< K > fma` (const `Vector< K >` &x, const `Vector< K >` &y, const K z)
- `template<typename K >`
`Vector< K > fma` (const K x, const K y, const `Vector< K >` &z)
- `template<typename K >`
`Vector< K > fma` (const K x, const `Vector< K >` &y, const K z)
- `template<typename K >`
`Vector< K > fma` (const `Vector< K >` &x, const K y, const K z)
- `template<typename K >`
`Vector< K > fma` (const `Vector< K >` &x, const `Vector< K >` &y, const `Vector< K >` &z)
- `template<typename K >`
`Vector< int > fpclassify` (const `Vector< K >` &v)
- `template<typename K >`
`bool isfinite` (const `Vector< K >` &v)
- `template<typename K >`
`bool isinf` (const `Vector< K >` &v)
- `template<typename K >`
`bool isnan` (const `Vector< K >` &v)
- `template<typename K >`
`bool isnormal` (const `Vector< K >` &v)
- `template<typename K >`
`Vector< K > real` (const `Vector< std::complex< K >` &v)
- `template<typename K >`
`Vector< K > imag` (const `Vector< std::complex< K >` &v)
- `template<typename K >`
`Vector< K > arg` (const `Vector< std::complex< K >` &v)
- `template<typename K >`
`Vector< K > conj` (const `Vector< K >` &v)

6.5.1 Detailed Description

Namespace that includes mathematical objects such as vectors, matrixes, dual numbers and useful numerical methods such as numerical integrators, ode solvers and solvers for algebraic equations.

6.5.2 Function Documentation

6.5.2.1 `abs()`

```
template<typename K >
Vector< double > Math::abs (
    const Vector< K > & v )
```

Parameters

<code>v</code>	
----------------	--

Returns

Vector<K>

6.5.2.2 acos() [1/2]

```
template<typename K >
DualNumber< K > Math::acos (
    const DualNumber< K > & x )
```

Returns $\text{acos}(x.a) - 1/\sqrt{1-x.a*x.a}*x.b*$ **Template Parameters**

<i>K</i>	
----------	--

Parameters

<i>x</i>	
----------	--

Returns

DualNumber<K>

6.5.2.3 acos() [2/2]

```
template<typename K >
Vector< K > Math::acos (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.4 acosh()

```
template<typename K >
Vector< K > Math::acosh (
    const Vector< K > & v )
```


Parameters

v	
-----	--

Returns

Vector<K>

6.5.2.5 asin() [1/2]

```
template<typename K >
DualNumber< K > Math::asin (
    const DualNumber< K > & x )
```

Returns $\text{asin}(x.a) + 1/\sqrt{1-x.a*x.a}*x.b*$

Template Parameters

K	
-----	--

Parameters

x	
-----	--

Returns

DualNumber<K>

6.5.2.6 asin() [2/2]

```
template<typename K >
Vector< K > Math::asin (
    const Vector< K > & v )
```

Parameters

v	
-----	--

Returns

Vector<K>

6.5.2.7 asinh()

```
template<typename K >
Vector< K > Math::asinh (
    const Vector< K > & v )
```

Parameters

v	
-----	--

Returns

Vector<K>

6.5.2.8 atan() [1/2]

```
template<typename K >
DualNumber< K > Math::atan (
    const DualNumber< K > & x )
```

Returns $\text{atan}(x.a) + 1/(1+x.a*x.a)*x.b*$

Template Parameters

K	
-----	--

Parameters

x	
-----	--

Returns

DualNumber<K>

6.5.2.9 atan() [2/2]

```
template<typename K >
Vector< K > Math::atan (
    const Vector< K > & v )
```

Parameters

v	
-----	--

Returns

Vector<K>

6.5.2.10 atan2() [1/3]

```
template<typename K >
Vector< K > Math::atan2 (
    const K v,
    const Vector< K > & w )
```

Parameters

<i>v</i>	
<i>w</i>	

Returns

Vector<K>

6.5.2.11 atan2() [2/3]

```
template<typename K >
Vector< K > Math::atan2 (
    const Vector< K > & v,
    const K w )
```

Parameters

<i>v</i>	
<i>w</i>	

Returns

Vector<K>

6.5.2.12 atan2() [3/3]

```
template<typename K >
Vector< K > Math::atan2 (
    const Vector< K > & v,
    const Vector< K > & w )
```

Exceptions

<i>Throws</i>	an exception if <i>v</i> and <i>w</i> have different lengths.
---------------	---

Parameters

<i>v</i>	
<i>w</i>	

Returns

Vector<K>

6.5.2.13 atanh()

```
template<typename K >
Vector< K > Math::atanh (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.14 cbrt()

```
template<typename K >
Vector< K > Math::cbrt (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.15 `ceil()`

```
template<typename K >
Vector< K > Math::ceil (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.16 `conj()` [1/16]

```
char Math::conj (
    const char & val ) [inline]
```

This function exists to make the definition of the scalar product more general. It just returns *val* .

Parameters

<i>val</i>	
------------	--

Returns

char

6.5.2.17 `conj()` [2/16]

```
double Math::conj (
    const double & val ) [inline]
```

This function exists to make the definition of the scalar product more general. It just returns *val* .

Parameters

<i>val</i>	
------------	--

Returns

double

6.5.2.18 conj() [3/16]

```
float Math::conj (
    const float & val ) [inline]
```

This function exists to make the definition of the scalar product more general. It just returns *val* .

Parameters

<i>val</i>	
------------	--

Returns

float

6.5.2.19 conj() [4/16]

```
int Math::conj (
    const int & val ) [inline]
```

This function exists to make the definition of the scalar product more general. It just returns *val* .

Parameters

<i>val</i>	
------------	--

Returns

int

6.5.2.20 conj() [5/16]

```
long double Math::conj (
    const long double & val ) [inline]
```

This function exists to make the definition of the scalar product more general. It just returns *val* .

Parameters

<i>val</i>	
------------	--

Returns

long double

6.5.2.21 conj() [6/16]

```
long int Math::conj (
    const long int & val ) [inline]
```

This function exists to make the definition of the scalar product more general. It just returns *val* .

Parameters

<i>val</i>	
------------	--

Returns

long int

6.5.2.22 conj() [7/16]

```
long long int Math::conj (
    const long long int & val ) [inline]
```

This function exists to make the definition of the scalar product more general. It just returns *val* .

Parameters

<i>val</i>	
------------	--

Returns

long long int

6.5.2.23 conj() [8/16]

```
short int Math::conj (
    const short int & val ) [inline]
```

This function exists to make the definition of the scalar product more general. It just returns *val* .

Parameters

<i>val</i>	
------------	--

Returns

short int

6.5.2.24 conj() [9/16]

```
template<typename K >
std::complex< K > Math::conj (
    const std::complex< K > & val ) [inline]
```

This function exists to make the definition of the scalar product more general. It just returns the complex conjugate of *val*.

Template Parameters

<i>K</i>	
----------	--

Parameters

<i>val</i>	
------------	--

Returns

std::complex<K>

6.5.2.25 conj() [10/16]

```
unsigned char Math::conj (
    const unsigned char & val ) [inline]
```

This function exists to make the definition of the scalar product more general. It just returns *val*.

Parameters

<i>val</i>	
------------	--

Returns

unsigned char

6.5.2.26 conj() [11/16]

```
unsigned int Math::conj (
    const unsigned int & val ) [inline]
```


This function exists to make the definition of the scalar product more general. It just returns *val* .

Parameters

<i>val</i>	
------------	--

Returns

unsigned int

6.5.2.27 conj() [12/16]

```
unsigned long int Math::conj (  
    const unsigned long int & val ) [inline]
```

This function exists to make the definition of the scalar product more general. It just returns *val* .

Parameters

<i>val</i>	
------------	--

Returns

unsigned long int

6.5.2.28 conj() [13/16]

```
unsigned long long int Math::conj (  
    const unsigned long long int & val ) [inline]
```

This function exists to make the definition of the scalar product more general. It just returns *val* .

Parameters

<i>val</i>	
------------	--

Returns

unsigned long long int

6.5.2.29 conj() [14/16]

```
unsigned short int Math::conj (
    const unsigned short int & val ) [inline]
```

This function exists to make the definition of the scalar product more general. It just returns *val* .

Parameters

<i>val</i>	
------------	--

Returns

unsigned short int

6.5.2.30 conj() [15/16]

```
template<typename K >
Vector< K > Math::conj (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.31 conj() [16/16]

```
wchar_t Math::conj (
    const wchar_t & val ) [inline]
```

This function exists to make the definition of the scalar product more general. It just returns *val* .

Parameters

<i>val</i>	
------------	--

Returns

wchar_t

6.5.2.32 copysign() [1/2]

```
template<typename K >
Vector< K > Math::copysign (
    const Vector< K > & x,
    const K y )
```

Parameters

<i>x</i>	
<i>y</i>	

Returns

Vector<K>

6.5.2.33 copysign() [2/2]

```
template<typename K >
Vector< K > Math::copysign (
    const Vector< K > & x,
    const Vector< K > & y )
```

Exceptions

<i>Throws</i>	an exception if <i>x</i> and <i>y</i> have different lengths.
---------------	---

Parameters

<i>x</i>	
<i>y</i>	

Returns

Vector<K>

6.5.2.34 cos() [1/2]

```
template<typename K >
DualNumber< K > Math::cos (
    const DualNumber< K > & x )
```

Returns $\cos(x.a) - x.b \cdot \sin(x.a)$

Template Parameters

K	
-----	--

Parameters

x	
-----	--

ReturnsDualNumber< K >**6.5.2.35 cos() [2/2]**

```
template<typename K >
Vector< K > Math::cos (
    const Vector< K > & v )
```

Parameters

v	
-----	--

ReturnsVector< K >**6.5.2.36 cosh() [1/2]**

```
template<typename K >
DualNumber< K > Math::cosh (
    const DualNumber< K > & x )
```

Returns $\cosh(x.a) + \sinh(x.a)*x.b*$ **Template Parameters**

K	
-----	--

Parameters

x	
-----	--

Returns

DualNumber<K>

6.5.2.37 cosh() [2/2]

```
template<typename K >
Vector< K > Math::cosh (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.38 erf()

```
template<typename K >
Vector< K > Math::erf (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.39 erfc()

```
template<typename K >
Vector< K > Math::erfc (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.40 exp()

```
template<typename K >
Vector< K > Math::exp (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.41 exp2()

```
template<typename K >
Vector< K > Math::exp2 (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.42 expm1()

```
template<typename K >
Vector< K > Math::expm1 (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.43 fabs()

```
template<typename K >
Vector< double > Math::fabs (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.44 fdim() [1/3]

```
template<typename K >
Vector< K > Math::fdim (
    const K x,
    const Vector< K > & y )
```

Parameters

<i>x</i>	
<i>y</i>	

Returns

Vector<K>

6.5.2.45 fdim() [2/3]

```
template<typename K >
Vector< K > Math::fdim (
    const Vector< K > & x,
    const Vector< K > & y )
```

Exceptions

<i>Throws</i>	an exception if x and y have different lengths.
---------------	---

Parameters

x	
y	

Returns

Vector<K>

6.5.2.46 fdim() [3/3]

```
template<typename K >
Vector< K > Math::fdim (
    const Vector< K > & x,
    K y )
```

Parameters

x	
y	

Returns

Vector<K>

6.5.2.47 floor()

```
template<typename K >
Vector< K > Math::floor (
    const Vector< K > & v )
```

Parameters

v	
-----	--

Returns

Vector<K>

6.5.2.48 fma() [1/7]

```
template<typename K >
Vector< K > Math::fma (
    const K x,
    const K y,
    const Vector< K > & z )
```

Parameters

<i>x</i>	
<i>y</i>	
<i>z</i>	

Returns

Vector<K>

6.5.2.49 fma() [2/7]

```
template<typename K >
Vector< K > Math::fma (
    const K x,
    const Vector< K > & y,
    const K z )
```

Parameters

<i>x</i>	
<i>y</i>	
<i>z</i>	

Returns

Vector<K>

6.5.2.50 fma() [3/7]

```
template<typename K >
Vector< K > Math::fma (
    const K x,
    const Vector< K > & y,
    const Vector< K > & z )
```

Exceptions

<i>Throws</i>	an exception if the length of <i>y</i> does not equal the length of <i>z</i> .
---------------	--

Parameters

<i>x</i>	
<i>y</i>	
<i>z</i>	

Returns

Vector<K>

6.5.2.51 fma() [4/7]

```
template<typename K >
Vector< K > Math::fma (
    const Vector< K > & x,
    const K y,
    const K z )
```

Parameters

<i>x</i>	
<i>y</i>	
<i>z</i>	

Returns

Vector<K>

6.5.2.52 fma() [5/7]

```
template<typename K >
Vector< K > Math::fma (
    const Vector< K > & x,
    const K y,
    const Vector< K > & z )
```

Exceptions

<i>Throws</i>	an exception if the length of <i>x</i> does not equal the length of <i>z</i> .
---------------	--

Parameters

x	
y	
z	

Returns

Vector<K>

6.5.2.53 fma() [6/7]

```
template<typename K >
Vector< K > Math::fma (
    const Vector< K > & x,
    const Vector< K > & y,
    const K z )
```

Exceptions

<i>Throws</i>	an exception if the length of x does not equal the length of y .
---------------	--

Parameters

x	
y	
z	

Returns

Vector<K>

6.5.2.54 fma() [7/7]

```
template<typename K >
Vector< K > Math::fma (
    const Vector< K > & x,
    const Vector< K > & y,
    const Vector< K > & z )
```

Exceptions

<i>Throws</i>	and exception if the lenghts of x , y and z are not equal.
---------------	--

Parameters

<i>x</i>	
<i>y</i>	
<i>z</i>	

Returns

Vector<K>

6.5.2.55 fmod() [1/3]

```
template<typename K >
Vector< K > Math::fmod (
    const K numer,
    const Vector< K > & denom )
```

Parameters

<i>numer</i>	
<i>denom</i>	

Returns

Vector<K>

6.5.2.56 fmod() [2/3]

```
template<typename K >
Vector< K > Math::fmod (
    const Vector< K > & numer,
    const K denom )
```

Parameters

<i>numer</i>	
<i>denom</i>	

Returns

Vector<K>

6.5.2.57 fmod() [3/3]

```
template<typename K >
Vector< K > Math::fmod (
    const Vector< K > & numer,
    const Vector< K > & denom )
```

Exceptions

<i>Throws</i>	an expception if <i>v</i> and <i>exponent</i> have different lenghts.
---------------	---

Parameters

<i>numer</i>	
<i>denom</i>	

Returns

Vector<K>

6.5.2.58 fpclassify()

```
template<typename K >
Vector< int > Math::fpclassify (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<int>

6.5.2.59 frexp()

```
template<typename K >
Vector< K > Math::frexp (
    const Vector< K > & v,
    Vector< int > * exp )
```

Parameters

<i>v</i>	
<i>exp</i>	

Returns

Vector<K>

6.5.2.60 hypot() [1/3]

```
template<typename K >
Vector< K > Math::hypot (
    const K x,
    const Vector< K > & y )
```

Parameters

<i>x</i>	
<i>y</i>	

Returns

Vector<K>

6.5.2.61 hypot() [2/3]

```
template<typename K >
Vector< K > Math::hypot (
    const Vector< K > & x,
    const K y )
```

Parameters

<i>x</i>	
<i>y</i>	

Returns

Vector<K>

6.5.2.62 hypot() [3/3]

```
template<typename K >
Vector< K > Math::hypot (
    const Vector< K > & x,
    const Vector< K > & y )
```

Exceptions

<i>Throws</i>	an expception if <i>v</i> and <i>exponent</i> have different lenghts.
---------------	---

Parameters

<i>x</i>	
<i>y</i>	

Returns

Vector<K>

6.5.2.63 ilogb()

```
template<typename K >
Vector< K > Math::ilogb (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.64 isfinite()

```
template<typename K >
bool Math::isfinite (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

true

false

6.5.2.65 isinf()

```
template<typename K >
bool Math::isinf (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

true
false

6.5.2.66 isnan()

```
template<typename K >
bool Math::isnan (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

true
false

6.5.2.67 isnormal()

```
template<typename K >
bool Math::isnormal (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

true
false

6.5.2.68 ldexp() [1/2]

```
template<typename K >
Vector< K > Math::ldexp (
    const Vector< K > & v,
    const int exp )
```

Parameters

<i>v</i>	
<i>exp</i>	

Returns

Vector<K>

6.5.2.69 ldexp() [2/2]

```
template<typename K >
Vector< K > Math::ldexp (
    const Vector< K > & v,
    const Vector< int > & exp )
```

Parameters

<i>v</i>	
<i>exp</i>	

Returns

Vector<K>

6.5.2.70 lgamma()

```
template<typename K >
Vector< K > Math::lgamma (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.71 llrint()

```
template<typename K >
Vector< long long int > Math::llrint (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<long long int>

6.5.2.72 llround()

```
template<typename K >
Vector< long long int > Math::llround (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<long long int>

6.5.2.73 log()

```
template<typename K >
Vector< K > Math::log (
    const Vector< K > & v )
```

Exceptions

<i>Throws</i>	an exception if <i>v</i> and <i>exp</i> have different lengths.
---------------	---

Parameters

v	
-----	--

Returns

Vector<K>

6.5.2.74 log10()

```
template<typename K >
Vector< K > Math::log10 (
    const Vector< K > & v )
```

Parameters

v	
-----	--

Returns

Vector<K>

6.5.2.75 log1p()

```
template<typename K >
Vector< K > Math::log1p (
    const Vector< K > & v )
```

Parameters

v	
-----	--

Returns

Vector<K>

6.5.2.76 log2()

```
template<typename K >
Vector< K > Math::log2 (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.77 logb()

```
template<typename K >
Vector< K > Math::logb (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.78 lrint()

```
template<typename K >
Vector< long int > Math::lrint (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<long int>

6.5.2.79 lround()

```
template<typename K >
Vector< long int > Math::lround (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<long int>

6.5.2.80 max()

```
template<typename K >
K Math::max (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

K

6.5.2.81 min()

```
template<typename K >
K Math::min (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

K

6.5.2.82 modf()

```
template<typename K >
Vector< K > Math::modf (
    const Vector< K > & v,
    Vector< K > * intpart )
```

Parameters

<i>v</i>	
<i>intpart</i>	

Returns

Vector<K>

6.5.2.83 multiply()

```
template<typename K >
K Math::multiply (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

K

6.5.2.84 nan()

```
template Vector< unsigned int > Math::nan (
    const unsigned int N,
    const char * tagp )
```

Parameters

<i>N</i>	
<i>tagp</i>	

Returns

Vector<K>

6.5.2.85 nearbyint()

```
template<typename K >
Vector< K > Math::nearbyint (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.86 nextafter()

```
template<typename K >
Vector< K > Math::nextafter (
    const Vector< K > & x,
    const Vector< K > & y )
```

Exceptions

<i>Throws</i>	an exception if <i>x</i> and <i>y</i> have different lengths.
---------------	---

Parameters

<i>x</i>	
<i>y</i>	

Returns

Vector<K>

6.5.2.87 operator"!="() [1/3]

```
template<typename K >
bool Math::operator!= (
    const K & alpha,
    const Vector< K > & v )
```

Parameters

<i>alpha</i>	
<i>v</i>	

Returns

true

false

6.5.2.88 operator"!=() [2/3]

```
template<typename K >
bool Math::operator!= (
    const Vector< K > & v,
    const K & alpha )
```

Parameters

<i>v</i>	
<i>alpha</i>	

Returns

true

false

6.5.2.89 operator"!=() [3/3]

```
template<typename K >
bool Math::operator!= (
    const Vector< K > & v,
    const Vector< K > & w )
```

Parameters

<i>v</i>	
<i>w</i>	

Returns

true

false

6.5.2.90 operator&() [1/3]

```
template<typename K >
Vector< K > Math::operator& (
    const K & v,
    const Vector< K > & w )
```

Creates a new **Vector** whose components are the scalar *v* followed by the components of *w*.

Parameters

v	
w	

Returns

Vector<K>

6.5.2.91 operator&() [2/3]

```
template<typename K >
Vector< K > Math::operator& (
    const Vector< K > & v,
    const K & w )
```

Creates a new [Vector](#) whose components are the components of v followed by the scalar w .

Parameters

v	
w	

Returns

Vector<K>

6.5.2.92 operator&() [3/3]

```
template<typename K >
Vector< K > Math::operator& (
    const Vector< K > & v,
    const Vector< K > & w )
```

Creates a new [Vector](#) whose components are the components of v followed by the components of w .

Parameters

v	
w	

Returns

Vector<K>

6.5.2.93 operator*() [1/6]

```
template<typename K >
DualNumber< K > Math::operator* (
    const DualNumber< K > & x,
    const DualNumber< K > & y )
```

Parameters

x	
y	

Returns

DualNumber<K>

6.5.2.94 operator*() [2/6]

```
template<typename K >
DualNumber< K > Math::operator* (
    const DualNumber< K > & x,
    const K y )
```

Parameters

x	
y	

Returns

DualNumber<K>

6.5.2.95 operator*() [3/6]

```
template<typename K >
Vector< K > Math::operator* (
    const K & alpha,
    const Vector< K > & v )
```

$v * \alpha$ is equivalent to $v[i] * \alpha$ for all i .

Parameters

α	
v	

Returns

Vector<K>

6.5.2.96 operator*() [4/6]

```
template<typename K >
DualNumber< K > Math::operator* (
    const K x,
    const DualNumber< K > & y )
```

Parameters

x	
y	

Returns

DualNumber<K>

6.5.2.97 operator*() [5/6]

```
template<typename K >
Vector< K > Math::operator* (
    const Vector< K > & v,
    const K & alpha )
```

$alpha * v$ is equivalent to $alpha * v[i]$ for all i .

Parameters

v	
$alpha$	

Returns

Vector<K>

6.5.2.98 operator*() [6/6]

```
template<typename K >
Vector< K > Math::operator* (
```

```
const Vector< K > & v,  
const Vector< K > & w )
```

$v * w$ is equivalent to $v[i] * w[i]$ for all i .

Exceptions

<i>An</i>	exception is thrown if v and w have a different number of components.
-----------	---

Parameters

v	
w	

Returns

Vector<K>

6.5.2.99 operator+() [1/6]

```
template<typename K >
DualNumber< K > Math::operator+ (
    const DualNumber< K > & x,
    const DualNumber< K > & y )
```

Parameters

x	
y	

Returns

DualNumber<K>

6.5.2.100 operator+() [2/6]

```
template<typename K >
DualNumber< K > Math::operator+ (
    const DualNumber< K > & x,
    const K y )
```

Parameters

x	
y	

Returns

DualNumber<K>

6.5.2.101 operator+() [3/6]

```
template<typename K >
Vector< K > Math::operator+ (
    const K & alpha,
    const Vector< K > & w )
```

$\alpha + v$ is equivalent to $\alpha + v[i]$ for all i .

Parameters

α	
w	

Returns

Vector<K>

6.5.2.102 operator+() [4/6]

```
template<typename K >
DualNumber< K > Math::operator+ (
    const K x,
    const DualNumber< K > & y )
```

Parameters

x	
y	

Returns

DualNumber<K>

6.5.2.103 operator+() [5/6]

```
template<typename K >
Vector< K > Math::operator+ (
    const Vector< K > & v,
    const K & alpha )
```

$v + \alpha$ is equivalent to $v[i] + \alpha$ for all i .

Parameters

v	
α	

Returns

Vector<K>

6.5.2.104 operator+() [6/6]

```
template<typename K >
Vector< K > Math::operator+ (
    const Vector< K > & v,
    const Vector< K > & w )
```

$v + w$ is equivalent to $v[i] + w[i]$ for all i . If any of the vectors has length zero, then the other one is returned.

Exceptions

A	runtime exception is thrown if both vectors have different number of components and their lengths are both non-zero.
---	--

Parameters

v	
w	

Returns

Vector<K>

6.5.2.105 operator-() [1/6]

```
template<typename K >
DualNumber< K > Math::operator- (
    const DualNumber< K > & x,
    const DualNumber< K > & y )
```

Parameters

x	
y	

Returns

DualNumber<K>

6.5.2.106 operator-() [2/6]

```
template<typename K >
DualNumber< K > Math::operator- (
    const DualNumber< K > & x,
    const K y )
```

Parameters

x	
y	

Returns

DualNumber<K>

6.5.2.107 operator-() [3/6]

```
template<typename K >
Vector< K > Math::operator- (
    const K & alpha,
    const Vector< K > & w )
```

$\alpha - v$ is equivalent to $\alpha - v[i]$ for all i .

Parameters

α	
w	

Returns

Vector<K>

6.5.2.108 operator-() [4/6]

```
template<typename K >
DualNumber< K > Math::operator- (
    const K x,
    const DualNumber< K > & y )
```


Parameters

x	
y	

Returns

DualNumber<K>

6.5.2.109 operator-() [5/6]

```
template<typename K >
Vector< K > Math::operator- (
    const Vector< K > & v,
    const K & alpha )
```

$v - \alpha$ is equivalent to $v[i] - \alpha$ for all i .

Parameters

v	
α	

Returns

Vector<K>

6.5.2.110 operator-() [6/6]

```
template<typename K >
Vector< K > Math::operator- (
    const Vector< K > & v,
    const Vector< K > & w )
```

$v - w$ is equivalent to $v[i] - w[i]$ for all i . If v has null length, then $-w$ is returned. If w has length zero, then v is returned.

Exceptions

A	runtime exception is thrown if both vectors have different number of components and their lengths are both non-zero.
---	--

Parameters

v	
-----	--

Parameters

<i>w</i>	
----------	--

Returns

Vector<K>

6.5.2.111 operator/() [1/6]

```
template<typename K >
DualNumber< K > Math::operator/ (
    const DualNumber< K > & x,
    const DualNumber< K > & y )
```

Parameters

<i>x</i>	
<i>y</i>	

Returns

DualNumber<K>

6.5.2.112 operator/() [2/6]

```
template<typename K >
DualNumber< K > Math::operator/ (
    const DualNumber< K > & x,
    const K y )
```

Parameters

<i>x</i>	
<i>y</i>	

Returns

DualNumber<K>

6.5.2.113 operator/() [3/6]

```
template<typename K >
Vector< K > Math::operator/ (
    const K & alpha,
    const Vector< K > & w )
```

α / v is equivalent to $\alpha / v[i]$ for all i .

Parameters

<i>alpha</i>	
<i>w</i>	

Returns

Vector<K>

6.5.2.114 operator/() [4/6]

```
template<typename K >
DualNumber< K > Math::operator/ (
    const K x,
    const DualNumber< K > & y )
```

Parameters

<i>x</i>	
<i>y</i>	

Returns

DualNumber<K>

6.5.2.115 operator/() [5/6]

```
template<typename K >
Vector< K > Math::operator/ (
    const Vector< K > & v,
    const K & alpha )
```

v / α is equivalent to $v[i] / \alpha$ for all i .

Parameters

<i>v</i>	
<i>alpha</i>	

Returns

Vector<K>

6.5.2.116 operator/() [6/6]

```
template<typename K >
Vector< K > Math::operator/ (
    const Vector< K > & v,
    const Vector< K > & w )
```

v / w is equivalent to $v[i] / w[i]$ for all i .

Exceptions

<i>An</i>	exception is thrown if v and w have a different number of components.
-----------	---

Parameters

v	
w	

Returns

Vector<K>

6.5.2.117 operator==() [1/3]

```
template<typename K >
bool Math::operator== (
    const K & alpha,
    const Vector< K > & v )
```

Parameters

α	
v	

Returns

true

false

6.5.2.118 operator==() [2/3]

```
template<typename K >
bool Math::operator== (
    const Vector< K > & v,
    const K & alpha )
```

Parameters

<i>v</i>	
<i>alpha</i>	

Returns

true

false

6.5.2.119 operator==() [3/3]

```
template<typename K >
bool Math::operator== (
    const Vector< K > & v,
    const Vector< K > & w )
```

Parameters

<i>v</i>	
<i>w</i>	

Returns

true

false

6.5.2.120 operator>() [1/3]

```
template<typename K >
bool Math::operator> (
    const K & alpha,
    const Vector< K > & v )
```

Parameters

<i>alpha</i>	
<i>v</i>	

Returns

true
false

6.5.2.121 operator>() [2/3]

```
template<typename K >
bool Math::operator> (
    const Vector< K > & v,
    const K & alpha )
```

Parameters

<i>v</i>	
<i>alpha</i>	

Returns

true
false

6.5.2.122 operator>() [3/3]

```
template<typename K >
bool Math::operator> (
    const Vector< K > & v,
    const Vector< K > & w )
```

Exceptions

<i>An</i>	exception is raised if <i>v</i> and <i>w</i> have a different number of components.
-----------	---

Parameters

<i>v</i>	
<i>w</i>	

Returns

true
false

6.5.2.123 operator>=() [1/3]

```
template<typename K >
bool Math::operator>= (
    const K & alpha,
    const Vector< K > & v )
```

Parameters

<i>alpha</i>	
<i>v</i>	

Returns

true
false

6.5.2.124 operator>=() [2/3]

```
template<typename K >
bool Math::operator>= (
    const Vector< K > & v,
    const K & alpha )
```

Parameters

<i>v</i>	
<i>alpha</i>	

Returns

true
false

6.5.2.125 operator>=() [3/3]

```
template<typename K >
bool Math::operator>= (
    const Vector< K > & v,
    const Vector< K > & w )
```

Exceptions

<i>An</i>	exception is raised if <i>v</i> and <i>w</i> have a different number of components.
-----------	---

Parameters

v	
w	

Returns

true

false

6.5.2.126 operator" | ()

```
template<typename K >
K Math::operator| (
    const Vector< K > & v,
    const Vector< K > & w )
```

Performs the following operation:

$$(v|w) = \sum_{i=0}^{N-1} v_i^* w_i$$

which is the canonical scalar product of \mathbb{K}^N .

Parameters

v	
w	

Returns

K

6.5.2.127 pow() [1/3]

```
template<typename K >
Vector< K > Math::pow (
    const K v,
    const Vector< K > & exponent )
```

Parameters

v	
$exponent$	

Returns

Vector<K>

6.5.2.128 pow() [2/3]

```
template<typename K >
Vector< K > Math::pow (
    const Vector< K > & v,
    const K exponent )
```

Parameters

<i>v</i>	
<i>exponent</i>	

Returns

Vector<K>

6.5.2.129 pow() [3/3]

```
template<typename K >
Vector< K > Math::pow (
    const Vector< K > & v,
    const Vector< K > & exponent )
```

Exceptions

<i>Throws</i>	an expection if <i>v</i> and <i>exponent</i> have different lenghts.
---------------	--

Parameters

<i>v</i>	
<i>exponent</i>	

Returns

Vector<K>

6.5.2.130 remainder() [1/3]

```
template<typename K >
Vector< K > Math::remainder (
```

```
const K numer,
const Vector< K > & denom )
```

Parameters

<i>numer</i>	
<i>denom</i>	

Returns

Vector<K>

6.5.2.131 remainder() [2/3]

```
template<typename K >
Vector< K > Math::remainder (
    const Vector< K > & numer,
    const K denom )
```

Parameters

<i>numer</i>	
<i>denom</i>	

Returns

Vector<K>

6.5.2.132 remainder() [3/3]

```
template<typename K >
Vector< K > Math::remainder (
    const Vector< K > & numer,
    const Vector< K > & denom )
```

Exceptions

<i>Throws</i>	an exception if <i>numer</i> and <i>denom</i> have different lengths.
---------------	---

Parameters

<i>numer</i>	
<i>denom</i>	

Returns

Vector<K>

6.5.2.133 remquo() [1/3]

```
template<typename K >
Vector< K > Math::remquo (
    const K numer,
    const Vector< K > & denom,
    Vector< int > * quot )
```

Parameters

<i>numer</i>	
<i>denom</i>	
<i>quot</i>	

Returns

Vector<K>

6.5.2.134 remquo() [2/3]

```
template<typename K >
Vector< K > Math::remquo (
    const Vector< K > & numer,
    const K denom,
    Vector< int > * quot )
```

Parameters

<i>numer</i>	
<i>denom</i>	
<i>quot</i>	

Returns

Vector<K>

6.5.2.135 remquo() [3/3]

```
template<typename K >
Vector< K > Math::remquo (
```

```

const Vector< K > & numer,
const Vector< K > & denom,
Vector< int > * quot )

```

Exceptions

Throws	an exception if <i>numer</i> and <i>denom</i> have different lengths.
---------------	---

Parameters

<i>numer</i>	
<i>denom</i>	
<i>quot</i>	

Returns

Vector<K>

6.5.2.136 rint()

```

template<typename K >
Vector< K > Math::rint (
    const Vector< K > & v )

```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.137 round()

```

template<typename K >
Vector< K > Math::round (
    const Vector< K > & v )

```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.138 scalbn() [1/2]

```
template<typename K >
Vector< K > Math::scalbn (
    const Vector< K > & v,
    const long int n )
```

Parameters

<i>v</i>	
<i>n</i>	

Returns

Vector<K>

6.5.2.139 scalbn() [2/2]

```
template<typename K >
Vector< K > Math::scalbn (
    const Vector< K > & v,
    const Vector< long int > & n )
```

Exceptions

<i>Throws</i>	an exception if <i>v</i> and <i>n</i> have different lengths.
---------------	---

Parameters

<i>v</i>	
<i>n</i>	

Returns

Vector<K>

6.5.2.140 scalbn() [1/2]

```
template<typename K >
Vector< K > Math::scalbn (
```

```
const Vector< K > & v,
const int n )
```

Parameters

<i>v</i>	
<i>n</i>	

Returns

Vector<K>

6.5.2.141 scalbn() [2/2]

```
template<typename K >
Vector< K > Math::scalbn (
    const Vector< K > & v,
    const Vector< int > & n )
```

Exceptions

<i>Throws</i>	an expception if <i>v</i> and <i>n</i> have different lenglths.
---------------	---

Parameters

<i>v</i>	
<i>n</i>	

Returns

Vector<K>

6.5.2.142 sin() [1/2]

```
template<typename K >
DualNumber< K > Math::sin (
    const DualNumber< K > & x )
```

Returns $\sin(x.a) + x.b \cdot \cos(x.a)$

Template Parameters

<i>K</i>	
----------	--

Parameters

x	
-----	--

Returns

DualNumber<K>

6.5.2.143 `sin()` [2/2]

```
template<typename K >
Vector< K > Math::sin (
    const Vector< K > & v )
```

Parameters

v	
-----	--

Returns

Vector<K>

6.5.2.144 `sinh()` [1/2]

```
template<typename K >
DualNumber< K > Math::sinh (
    const DualNumber< K > & x )
```

Returns $\sinh(x.a) + \cosh(x.a)*x.b*$

Template Parameters

K	
-----	--

Parameters

x	
-----	--

Returns

DualNumber<K>

6.5.2.145 sinh() [2/2]

```
template<typename K >
Vector< K > Math::sinh (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.146 sqrt()

```
template<typename K >
Vector< K > Math::sqrt (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.147 sum()

```
template<typename K >
K Math::sum (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

K

6.5.2.148 tan() [1/2]

```
template<typename K >
DualNumber< K > Math::tan (
    const DualNumber< K > & x )
```

Returns $\tan(x.a) + x.b/\cos(x.a)/\cos(x.a)*$

Template Parameters

<i>K</i>	
----------	--

Parameters

<i>x</i>	
----------	--

Returns

DualNumber<K>

6.5.2.149 tan() [2/2]

```
template<typename K >
Vector< K > Math::tan (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.150 tanh() [1/2]

```
template<typename K >
DualNumber< K > Math::tanh (
    const DualNumber< K > & x )
```

Returns $\tanh(x.a) + 1/\cosh(x.a)/\cosh(x.a)*x.b*$

Template Parameters

<i>K</i>	
----------	--

Parameters

x	
-----	--

Returns

DualNumber<K>

6.5.2.151 tanh() [2/2]

```
template<typename K >
Vector< K > Math::tanh (
    const Vector< K > & v )
```

Parameters

v	
-----	--

Returns

Vector<K>

6.5.2.152 tgamma()

```
template<typename K >
Vector< K > Math::tgamma (
    const Vector< K > & v )
```

Parameters

v	
-----	--

Returns

Vector<K>

6.5.2.153 trunc()

```
template<typename K >
Vector< K > Math::trunc (
    const Vector< K > & v )
```

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

6.5.2.154 vector_product_2d()

```
template<typename K >
K Math::vector_product_2d (
    const Vector< K > & v,
    const Vector< K > & w )
```

Exceptions

<i>Throws</i>	an exception if either the dimension of <i>v</i> or the dimension of <i>w</i> is different from two.
---------------	--

Parameters

<i>v</i>	
<i>w</i>	

Returns

K

6.5.2.155 vector_product_3d()

```
template<typename K >
Vector< K > Math::vector_product_3d (
    const Vector< K > & v,
    const Vector< K > & w )
```

Exceptions

<i>Throws</i>	an exception if either the dimension of <i>v</i> or the dimension of <i>w</i> is different from three.
---------------	--

Parameters

<i>v</i>	
<i>w</i>	

Returns

Vector<K>

6.6 Math::AlgebraicSolvers Namespace Reference

Contains solvers for algebraic equations.

Classes

- class [LinearSystemSolver](#)

Solves linear systems of the form $Ax=b$ through LU decomposition with partial pivoting.

Functions

- double [bisection](#) (std::function< double(const double x)> f, double a, double b, const unsigned int iter_max, const double abs_tol, const double rel_tol)
Applies the bisection method to the function f in the interval (a,b).
- double [secant](#) (std::function< double(const double x)> f, double x1, double x2, const unsigned int iter_max, const double abs_tol, const double rel_tol)
Applies the secant method to the function f with initial guesses x1 and x2.
- double [newton_raphson](#) (std::function< double(const double x)>f, std::function< double(const double x)>dfdx, const double x0, const unsigned int iter_max, const double abs_tol, const double rel_tol)
Applies the Newton-Raphson method to the function f with initial guess x0.
- double [newton_raphson](#) (std::function< [DualNumber](#)< double >(const [DualNumber](#)< double > x)>f, const double x0, const unsigned int iter_max, const double abs_tol, const double rel_tol)
Applies the Newton-Raphson method to the function f with initial guess x0.

6.6.1 Detailed Description

Contains solvers for algebraic equations.

6.6.2 Function Documentation

6.6.2.1 bisection()

```
double Math::AlgebraicSolvers::bisection (
    std::function< double(const double x)> f,
    double a,
    double b,
    const unsigned int iter_max,
    const double abs_tol,
    const double rel_tol )
```

Applies the bisection method to the function f in the interval (a,b).

Returns the current estimation to the solution as soon as one of the following things happens:

1. The maximum number of iterations is reached.
2. The absolute error is smaller than the tolerance given.
3. The relative error is smaller than the tolerance given.

To disable either the absolute or relative error check, set the corresponding tolerance to zero.

Parameters

<i>f</i>	the function.
<i>a</i>	the left side of the interval.
<i>b</i>	the right side of the interval.
<i>iter_max</i>	the maximum number of iterations.
<i>abs_tol</i>	the tolerance for the absolute error.
<i>rel_tol</i>	the tolerance for the relative error.

Returns

double

6.6.2.2 newton_raphson() [1/2]

```
double Math::AlgebraicSolvers::newton_raphson (  
    std::function< double(const double x)> f,  
    std::function< double(const double x)> dfdx,  
    const double x0,  
    const unsigned int iter_max,  
    const double abs_tol,  
    const double rel_tol )
```

Applies the Newton-Raphson method to the function *f* with initial guess *x0*.

Returns the current estimation to the solution as soon as one of the following things happens:

1. The maximum number of iterations is reached.
2. The absolute error is smaller than the tolerance given.
3. The relative error is smaller than the tolerance given.

To disable either the absolute or relative error check, set the corresponding tolerance to zero.

Parameters

<i>f</i>	the function.
<i>dfdx</i>	the derivative of the function.
<i>x0</i>	the initial estimation.
<i>iter_max</i>	the maximum number of iterations.
<i>abs_tol</i>	the tolerance for the absolute error.
<i>rel_tol</i>	the tolerance for the relative error.

Returns

double

6.6.2.3 newton_raphson() [2/2]

```
double Math::AlgebraicSolvers::newton_raphson (
    std::function< DualNumber< double > (const DualNumber< double > x)> f,
    const double x0,
    const unsigned int iter_max,
    const double abs_tol,
    const double rel_tol )
```

Applies the Newton-Raphson method to the function *f* with initial guess *x0*.

Returns the current estimation to the solution as soon as one of the following things happens:

1. The maximum number of iterations is reached.
2. The absolute error is smaller then the tolerance given.
3. The relative error is smaller then the tolerance given.

To disable either the absolute or relative error check, set the corresponding tolerance to zero.

Parameters

<i>f</i>	the function (returning a dual number).
<i>x0</i>	the initial estimation.
<i>iter_max</i>	the maximum number of iterations.
<i>abs_tol</i>	the tolerance for the absolute error.
<i>rel_tol</i>	the tolerance for the relative error.

Returns

double

6.6.2.4 secant()

```
double Math::AlgebraicSolvers::secant (
    std::function< double (const double x)> f,
    double x1,
    double x2,
    const unsigned int iter_max,
    const double abs_tol,
    const double rel_tol )
```

Applies the secant method to the function *f* with initial guesses *x1* and *x2*.

Returns the current estimation to the solution as soon as one of the following things happens:

1. The maximum number of iterations is reached.
2. The absolute error is smaller then the tolerance given.
3. The relative error is smaller then the tolerance given.

To disable either the absolute or relative error check, set the corresponding tolerance to zero.

Parameters

<i>f</i>	the function.
<i>x1</i>	the first initial estimation.
<i>x2</i>	the second initial estimation.
<i>iter_max</i>	the maximum number of iterations.
<i>abs_tol</i>	the tolerance for the absolute error.
<i>rel_tol</i>	the tolerance for the relative error.

Returns

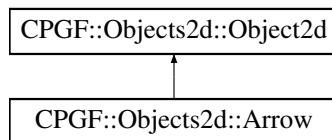
double

Chapter 7

Class Documentation

7.1 CPGF::Objects2d::Arrow Class Reference

Inheritance diagram for CPGF::Objects2d::Arrow:



Public Member Functions

- **Arrow** (const [AffineSpace::Point2d](#) &start, const [AffineSpace::Point2d](#) &end, const double arrow_head_length=0.15, const double arrow_head_width=0.3, const [Color](#) &color=[Color::BLACK](#), const double opacity=1, const double line_width=[LineWidth::SEMITHICK](#), const std::vector< double > &dash_pattern=[DashPatterns::SOLID](#))

Static Protected Member Functions

- static **Object2d builder** (const [AffineSpace::Point2d](#) &start, const [AffineSpace::Point2d](#) &end, const double arrow_head_length, const double arrow_head_width, const [Color](#) &color, const double opacity, const double line_width, const std::vector< double > &dash_pattern)

Additional Inherited Members

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

- CPGF/Objects2d/BasicGeometries.hpp
- CPGF/Objects2d/BasicGeometries.cpp

7.2 Math::Interpolation::AverageLinearInterpolation Class Reference

Public Member Functions

- **AverageLinearInterpolation** (const std::vector< double > &averages, const std::vector< double > &partition)
- **AverageLinearInterpolation** (double *averages, double *partition, unsigned int N_cells)
- **AverageLinearInterpolation** (const [AverageLinearInterpolation](#) &f)
- [AverageLinearInterpolation](#) & **operator=** (const [AverageLinearInterpolation](#) &f)
- double **operator()** (const double x) const
We do linear extrapolation at the ends.

Public Attributes

- unsigned int **N_cells**
- double * **x_part**
- double * **b**
- double * **c**

7.2.1 Member Function Documentation

7.2.1.1 operator()

```
double AverageLinearInterpolation::operator() (
    const double x ) const
```

We do linear extrapolation at the ends.

Parameters

<i>x</i>	
----------	--

Returns

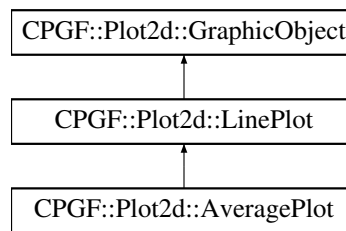
double

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

- Math/Interpolation.hpp
- Math/Interpolation.cpp

7.3 CPGF::Plot2d::AveragePlot Class Reference

Inheritance diagram for CPGF::Plot2d::AveragePlot:



Public Member Functions

- **AveragePlot** (const std::vector< double > &Y, const std::vector< double > &partition, const [Color](#) &color=[Color::BLUE](#), const double line_width=[LineWidth::THIN](#), const double opacity=1, const std::vector< double > &dash_pattern=[DashPatterns::SOLID](#), const std::string &legend="")
Allows to average values over a partition of the real line. Returns a [LinePlot](#).
- **AveragePlot** (const std::vector< double > &Y, const std::vector< double > &partition, std::function< [Color](#)(unsigned int)> color, std::function< double(unsigned int)> line_width, std::function< double(unsigned int)> opacity, std::function< std::vector< double >(unsigned int)> dash_pattern, const std::string &legend="")
- **AveragePlot** (const std::vector< double > &Y, const std::vector< double > &partition, std::function< [Color](#)([AffineSpace::Point2d](#) &)> color, std::function< double([AffineSpace::Point2d](#) &)> line_width, std::function< double([AffineSpace::Point2d](#) &)> opacity, std::function< std::vector< double >(affineSpace::Point2d &)> dash_pattern, const std::string &legend="")

Static Protected Member Functions

- static **LinePlot builder** (const std::vector< double > &Y, const std::vector< double > &partition, const [Color](#) &color, const double line_width, const double opacity, const std::vector< double > &dash_pattern, const std::string &legend)
- static **LinePlot builder** (const std::vector< double > &Y, const std::vector< double > &partition, std::function< [Color](#)(unsigned int)> color, std::function< double(unsigned int)> line_width, std::function< double(unsigned int)> opacity, std::function< std::vector< double >(unsigned int)> dash_pattern, const std::string &legend)
- static **LinePlot builder** (const std::vector< double > &Y, const std::vector< double > &partition, std::function< [Color](#)([AffineSpace::Point2d](#) &)> color, std::function< double([AffineSpace::Point2d](#) &)> line_width, std::function< double([AffineSpace::Point2d](#) &)> opacity, std::function< std::vector< double >(affineSpace::Point2d &)> dash_pattern, const std::string &legend)

Additional Inherited Members

7.3.1 Constructor & Destructor Documentation

7.3.1.1 AveragePlot()

```

AveragePlot::AveragePlot (
    const std::vector< double > & Y,
    const std::vector< double > & partition,
    const Color & color = Color::BLUE,
    const double line_width = LineWidth::THIN,
    const double opacity = 1,
    const std::vector< double > & dash_pattern = DashPatterns::SOLID,
    const std::string & legend = "" )

```

Allows to average values over a partition of the real line. Returns a [LinePlot](#).

Parameters

<i>Y</i>	
<i>partition</i>	
<i>color</i>	
<i>line_width</i>	
<i>opacity</i>	
<i>dash_pattern</i>	
<i>legend</i>	

Returns

[LinePlot](#)

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

- CPGF/Plot2d/LinePlot.hpp
- CPGF/Plot2d/LinePlot.cpp

7.4 Math::Interpolation::AverageQuadraticInterpolation Class Reference

Public Member Functions

- **AverageQuadraticInterpolation** (const std::vector< double > &averages, const std::vector< double > &partition)
- **AverageQuadraticInterpolation** (double *averages, double *partition, unsigned int N_cells)
- **AverageQuadraticInterpolation** (const [AverageQuadraticInterpolation](#) &f)
- [AverageQuadraticInterpolation](#) & **operator=** (const [AverageQuadraticInterpolation](#) &f)
- double **operator()** (const double x) const
We do linear extrapolation at the ends.

Protected Attributes

- unsigned int **N_cells**
- double * **x_part**
- double * **a**
- double * **b**
- double * **c**

7.4.1 Member Function Documentation

7.4.1.1 operator()()

```
double AverageQuadraticInterpolation::operator() (
    const double x ) const
```

We do linear extrapolation at the ends.

Parameters

x	
---	--

Returns

double

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

- Math/Interpolation.hpp
- Math/Interpolation.cpp

7.5 CPGF::Plot2d::Axis Class Reference

Public Member Functions

- void **set_max_value** (const double value)
- void **reset_max_value** ()
- void **set_min_value** (const double value)
- void **reset_min_value** ()
- void **update_max_min** (const double max_value, const double min_value)
- void **calculate_transformations** ()
- double **get_min_value** () const
- double **get_max_value** () const
- std::function< double(double)> **axis_transform** () const
Applied to a coordiante, it returns the coordinate on the final drawing.
- std::function< double(double)> **axis_transform_inverse** () const
- **Scene2d render_to_scene** () const
- **Axis** (const AxisType **axis_type**, const std::string &**label**="", const AxisScale **axis_scale**=AxisScale::↔ LINEAR, const double **scale**=1, const bool **visible**=true, const bool **inverted**=false, const double **position**=0, const **Color** &**color**=Color::BLACK, const double **line_width**=LineWidth::THIN, const double **opacity**=1, const std::vector< double > &**dash_pattern**=DashPatterns::SOLID, const double **aspect_ratio**=0.6, const double **arrow_head_length**=0.3, const double **arrow_head_width**=0.75, const double **arrow_length**=0.5, std::function< Objects2d::Object2d(const AffineSpace::Point2d &start, const AffineSpace::Point2d &end, const double **arrow_head_length**, const double **arrow_head_width**, const **Color** &**color**, const double **opacity**, const double **line_width**, const std::vector< double > &**dash_pattern**)> **arrow**=[])(const AffineSpace::Point2d &start, const AffineSpace::Point2d &end, const double **arrow_head_length**, const double **arrow_head_width**, const **Color** &**color**, const double **opacity**, const double **line_width**, const std::vector< double > &**dash_pattern**) {return Objects2d::Arrow(start, end, **arrow_head_length**, **arrow_head_width**, **color**, **opacity**, **line_width**, **dash_pattern**);}, const unsigned int **N_major_ticks**=9, const double **major_tick_deviation**=0.↔ 25, const double **major_tick_line_width_divisor**=3, const bool **show_medium_ticks**=true, const double **medium_tick_deviation**=0.20, const double **medium_tick_line_width_divisor**=4, const bool **show_small_ticks**=true, const double **small_tick_deviation**=0.15, const double **small_tick_line_width_divisor**=5, const bool **show_numbers**=true, const bool **show_major_grid_lines**=false, const std::vector< double > &**major_grid_lines_dash_pattern**=Dash↔ Patterns::SOLID, const double **major_grid_line_line_width_divisor**=2, const double **major_grid_line_opacity**=0.↔ 75, const bool **show_medium_grid_lines**=false, const std::vector< double > &**medium_grid_lines_dash_pattern**=Dash↔ Patterns::SOLID, const double **medium_grid_line_line_width_divisor**=4, const double **medium_grid_line_opacity**=0.↔ 5, const bool **show_small_grid_lines**=false, const std::vector< double > &**small_grid_lines_dash_pattern**=Dash↔ Patterns::SOLID, const double **small_grid_line_line_width_divisor**=8, const double **small_grid_line_opacity**=0.↔ 25)

Public Attributes

- AxisType **axis_type**
Determines whether the axis is horizontal or vertical.
- AxisScale **axis_scale**
Represents the scale of the axis.
- double **scale**
Determines the relative size between the graph and the labels. Use scale=1 for full-paged graphics and scale=0.5 for half-paged graphics.
- bool **visible**
Determines whether the axis is rendered or not.
- bool **inverted**
Controls whether the axis is inverted, a.k.a. points in the opposite direction.
- std::string **label**
The axis label.
- double **position**
A real number between zero and one which represents the position of the axis on the graph.
- Color **color**
Color of the axis.
- double **line_width**
Axis line width.
- double **opacity**
Axis opacity.
- std::vector< double > **dash_pattern**
Axis dash pattern.
- double **aspect_ratio**
Determines the length of the axis if it is vertical. To obtain the axis length, we have to multiply the aspect_ratio by the default length of all X axes.
- double **arrow_head_length**
Arrow head length.
- double **arrow_head_width**
Arrow width.
- double **arrow_length**
Determines how much the arrow sticks out of the axis.
- std::function< Objects2d::Object2d(const AffineSpace::Point2d &start, const AffineSpace::Point2d &end, const double arrow_head_length, const double arrow_head_width, const Color &color, const double opacity, const double line_width, const std::vector< double > dash_pattern)> **arrow**
Function that creates the arrow head.
- unsigned int **N_major_ticks**
Number of major ticks.
- double **major_tick_deviation**
Controls how large the major ticks are in the perpendicular direction to the axis.
- double **major_tick_line_width_divisor**
The width of the major ticks will be the axis width divided by this factor.
- bool **show_medium_ticks**
Controls whether medium ticks are displayed.
- double **medium_tick_deviation**
Controls how large the medium ticks are in the perpendicular direction to the axis.
- double **medium_tick_line_width_divisor**
The width of the medium ticks will be the axis width divided by this factor.
- bool **show_small_ticks**
Controls whether small ticks are displayed.

- double **small_tick_deviation**
Controls how large the small ticks are in the perpendicular direction to the axis.
- double **small_tick_line_width_divisor**
The width of the small ticks will be the axis width divided by this factor.
- bool **show_numbers**
Controls whether the number scale is displayed.
- NumberPosition **number_position**
Determines the position of the numbers.
- bool **show_major_grid_lines**
Determines whether the major grid lines are drawn.
- std::vector< double > **major_grid_lines_dash_pattern**
The dash pattern of the major grid lines.
- double **major_grid_line_line_width_divisor**
The width of the major grid lines will be the axis width divided by this factor.
- double **major_grid_line_opacity**
The opacity of the major grid lines.
- bool **show_medium_grid_lines**
Determines whether the medium grid lines are drawn.
- std::vector< double > **medium_grid_lines_dash_pattern**
The dash pattern of the medium grid lines.
- double **medium_grid_line_line_width_divisor**
The width of the medium grid lines will be the axis width divided by this factor.
- double **medium_grid_line_opacity**
The opacity of the medium grid lines.
- bool **show_small_grid_lines**
Determines whether the small grid lines are drawn.
- std::vector< double > **small_grid_lines_dash_pattern**
The dash pattern of the medium grid lines.
- double **small_grid_line_line_width_divisor**
The width of the small grid lines will be the axis width divided by this factor.
- double **small_grid_line_opacity**
The opacity of the small grid lines.

Static Public Attributes

- constexpr static const double **X_MAX** = 15
End position of the axis (if it is horizontal) on the final drawing.
- constexpr static const double **X_MIN** = 0
Start position of the axis (if it is horizontal) on the final drawing.
- constexpr static const double **Y_MIN** = 0
Start position of the axis (if it is vertical) on the final drawing.
- constexpr static const double **NUMBER_DISPLACEMENT** = 0.5
Determines how much the numbers are displaced with respect to the axis.
- constexpr static const double **LABEL_DISPLACEMENT_HORIZONTAL** = 1.2
Determines how much the label is displaced with respect to the axis.
- constexpr static const double **LABEL_DISPLACEMENT_VERTICAL** = 2

Protected Attributes

- bool **user_defined_max_value**
- double **max_value**
- bool **user_defined_min_value**
- double **min_value**
- double **x_max**
- double **_a**
- double **_b**
- int **digit_max**
- unsigned int **precision**

7.5.1 Member Function Documentation

7.5.1.1 axis_transform()

```
std::function< double(double)> Axis::axis_transform ( ) const
```

Applied to a coordiante, it returns the coordinate on the final drawing.

Returns

```
std::function<double(double)>
```

7.5.2 Member Data Documentation

7.5.2.1 aspect_ratio

```
double CPGF::Plot2d::Axis::aspect_ratio
```

Determines the length of the axis if it is vertical. To obtain the axis length, we have to multiply the `aspect_ratio` by the default length of all X axes.

Default value is $1/\sqrt{2}$.

7.5.2.2 N_major_ticks

```
unsigned int CPGF::Plot2d::Axis::N_major_ticks
```

Number of major ticks.

Set to 0 to disable.

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

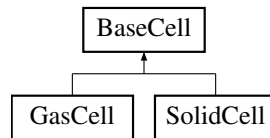
- CPGF/Plot2d/Axis.hpp
- CPGF/Plot2d/Axis.cpp

7.6 BaseCell Class Reference

The basic structure of the space discretization.

```
#include <Cell.hpp>
```

Inheritance diagram for BaseCell:



Public Member Functions

- **BaseCell** (const **Math::Vector**< double > &**U**, const double **a**, const double **b**, const double **A**, **Chemistry::SolidGasReaction** ***QR**)
Construct a new Base Cell object.
- **BaseCell** (FILE *file, **Chemistry::SolidGasReaction** ***QR**)
Construct a new Cell object from file.
- double **x** () const
Returns cell average position.
- double **len** () const
Returns cell length.
- void **read_from_file** (FILE *file)
Reads the cell values from file.
- void **write_to_file** (FILE *file) const
Writes the cell values to file.
- std::string **to_string** () const
Returns a string representation of the cell.

Public Attributes

- **Math::Vector**< double > **U**
Vector of cell conserved variables.
- double **a**
Left limit of the cell.
- double **b**
Right limit of the cell.
- double **A**
Area of the cell.
- **Chemistry::SolidGasReaction** * **QR**
Pointer to the chemical reaction.

7.6.1 Detailed Description

The basic structure of the space discretization.

7.6.2 Constructor & Destructor Documentation

7.6.2.1 BaseCell() [1/2]

```
BaseCell::BaseCell (
    const Math::Vector< double > & U,
    const double a,
    const double b,
    const double A,
    Chemistry::SolidGasReaction * QR )
```

Construct a new Base Cell object.

Parameters

<i>U</i>	the vector of preserved variables.
<i>a</i>	the left limit of the cell.
<i>b</i>	the right limit of the cell.
<i>A</i>	the area of the cell.
<i>QR</i>	a pointer to the chemical reaction.

7.6.2.2 BaseCell() [2/2]

```
BaseCell::BaseCell (
    FILE * file,
    Chemistry::SolidGasReaction * QR )
```

Construct a new Cell object from file.

Parameters

<i>file</i>	
-------------	--

7.6.3 Member Function Documentation

7.6.3.1 len()

```
double BaseCell::len ( ) const
```

Returns cell length.

Returns

double

7.6.3.2 read_from_file()

```
void BaseCell::read_from_file (
    FILE * file )
```

Reads the cell values from file.

Parameters

<i>file</i>	
-------------	--

7.6.3.3 to_string()

```
std::string BaseCell::to_string ( ) const
```

Returns a string representation of the cell.

Returns

std::string

7.6.3.4 write_to_file()

```
void BaseCell::write_to_file (
    FILE * file ) const
```

Writes the cell values to file.

Parameters

<i>file</i>	
-------------	--

7.6.3.5 x()

```
double BaseCell::x ( ) const
```

Returns cell average position.

Returns

double

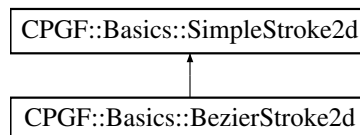
The documentation for this class was generated from the following files:

- [Mesh/Cell.hpp](#)
- [Mesh/Cell.cpp](#)

7.7 CPGF::Basics::BezierStroke2d Class Reference

```
#include <Strokes2d.hpp>
```

Inheritance diagram for CPGF::Basics::BezierStroke2d:

**Public Member Functions**

- **BezierStroke2d** (const [AffineSpace::Point2d](#) &P1, const [AffineSpace::Point2d](#) &P2, const [AffineSpace::Point2d](#) &Q1, const [AffineSpace::Point2d](#) &Q2)
- [BezierStroke2d](#) * **clone** () const override
- [AffineSpace::Point2d](#) & **start** () override
- [AffineSpace::Point2d](#) **start** () const override
- [AffineSpace::Point2d](#) & **end** () override
- [AffineSpace::Point2d](#) **end** () const override
- [BezierStroke2d](#) & **translate** (const [AffineSpace::Vector2d](#) &v) override
- [BezierStroke2d](#) & **rotate_with_respect_to** (const [AffineSpace::Point2d](#) &Q, const double theta) override
- [BezierStroke2d](#) & **scale_with_respect_to** (const [AffineSpace::Point2d](#) &Q, const [AffineSpace::Vector2d](#) &s) override
- double **length** () const override
- double **area** () const override
- std::vector< [AffineSpace::Point2d](#) > **operator/** (const [SimpleStroke2d](#) &B) override
- std::string **render_to_string** () const override

Public Attributes

- [AffineSpace::Point2d](#) P1
- [AffineSpace::Point2d](#) P2
- [AffineSpace::Point2d](#) Q1
- [AffineSpace::Point2d](#) Q2

7.7.1 Detailed Description

Todo Finish implementation.

7.7.2 Member Function Documentation

7.7.2.1 area()

```
double BezierStroke2d::area ( ) const [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.7.2.2 clone()

```
BezierStroke2d * BezierStroke2d::clone ( ) const [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.7.2.3 end() [1/2]

```
Point2d BezierStroke2d::end ( ) const [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.7.2.4 end() [2/2]

```
Point2d & BezierStroke2d::end ( ) [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.7.2.5 length()

```
double BezierStroke2d::length ( ) const [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.7.2.6 operator/()

```
std::vector< Point2d > BezierStroke2d::operator/ (
    const SimpleStroke2d & B ) [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.7.2.7 render_to_string()

```
std::string BezierStroke2d::render_to_string ( ) const [override], [virtual]
```

Parameters

<i>alpha</i>	a number between zero and one.
--------------	--------------------------------

Returns

[AffineSpace::Point2d](#)

Implements [CPGF::Basics::SimpleStroke2d](#).

7.7.2.8 rotate_with_respect_to()

```
BezierStroke2d & BezierStroke2d::rotate_with_respect_to (
    const AffineSpace::Point2d & Q,
    const double theta ) [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.7.2.9 scale_with_respect_to()

```
BezierStroke2d & BezierStroke2d::scale_with_respect_to (
    const AffineSpace::Point2d & Q,
    const AffineSpace::Vector2d & s ) [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.7.2.10 start() [1/2]

```
Point2d BezierStroke2d::start ( ) const [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.7.2.11 start() [2/2]

```
Point2d & BezierStroke2d::start ( ) [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.7.2.12 translate()

```
BezierStroke2d & BezierStroke2d::translate (
    const AffineSpace::Vector2d & v ) [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.7.3 Member Data Documentation

7.7.3.1 P1

[AffineSpace::Point2d](#) CPGF::Basics::BezierStroke2d::P1

Starting point.

7.7.3.2 P2

[AffineSpace::Point2d](#) CPGF::Basics::BezierStroke2d::P2

Ending point

7.7.3.3 Q1

[AffineSpace::Point2d](#) CPGF::Basics::BezierStroke2d::Q1

First control point.

7.7.3.4 Q2

[AffineSpace::Point2d](#) CPGF::Basics::BezierStroke2d::Q2

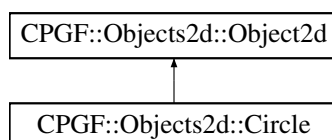
Second control point.

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

- CPGF/PGFBasics/Strokes2d.hpp
- CPGF/PGFBasics/Strokes2d.cpp

7.8 CPGF::Objects2d::Circle Class Reference

Inheritance diagram for CPGF::Objects2d::Circle:



Public Member Functions

- **Circle** (const [AffineSpace::Point2d](#) &pos, const double radius, const bool draw=true, const bool fill=false, const [Color](#) &draw_color=[Color::BLACK](#), const [Color](#) &fill_color=[Color::WHITE](#), const double opacity=1, const double line_width=[LineWidth::SEMITHICK](#), const std::vector< double > &dash_pattern=[DashPatterns::↔](#) SOLID)
- [AffineSpace::Point2d](#) **center** () const
- double **radius** () const

Static Protected Member Functions

- static [Object2d](#) **builder** (const [AffineSpace::Point2d](#) &pos, const double radius, const bool draw, const bool fill, const [Color](#) &draw_color, const [Color](#) &fill_color, const double opacity, const double line_width, const std::vector< double > &dash_pattern)

Additional Inherited Members

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

- CPGF/Objects2d/BasicGeometries.hpp
- CPGF/Objects2d/BasicGeometries.cpp

7.9 CPGF::Color Class Reference

An object used to represent an rgb color. r, g and b must all be real numbers between 0 and 1.

```
#include <PGFConf.hpp>
```

Public Member Functions

- std::string **to_string** ()
Returns a string representation of the color.
- **Color** ()
Returns the red color (1,0,0).
- **Color** (const double r, const double g, const double b)
Returns the (r,g,b) color.

Static Public Member Functions

- static [Color](#) **mix** (const [Color](#) &A, const [Color](#) &B, const double alpha)
Returns a mix of colors A and B in proportions alpha and 1 - alpha.
- static [Color](#) **from_RGB** (const unsigned char R, const unsigned char G, const unsigned char B)

Public Attributes

- double **r**
The amount of red of the color. $0 \leq r \leq 1$.
- double **g**
The amount of green of the color. $0 \leq g \leq 1$.
- double **b**
The amount of blue of the color. $0 \leq b \leq 1$.

Static Public Attributes

- static const [Color](#) **MAROON** = Color::from_RGB(128, 0, 0)
- static const [Color](#) **DARK_RED** = Color::from_RGB(139, 0, 0)
- static const [Color](#) **BROWN** = Color::from_RGB(165, 42, 42)
- static const [Color](#) **FIREBRICK** = Color::from_RGB(178, 34, 34)
- static const [Color](#) **CRIMSON** = Color::from_RGB(220, 20, 60)
- static const [Color](#) **RED** = Color::from_RGB(255, 0, 0)
- static const [Color](#) **TOMATO** = Color::from_RGB(255, 99, 71)
- static const [Color](#) **CORAL** = Color::from_RGB(255, 127, 80)
- static const [Color](#) **INDIAN_RED** = Color::from_RGB(205, 92, 92)
- static const [Color](#) **LIGHT_CORAL** = Color::from_RGB(240, 128, 128)
- static const [Color](#) **DARK_SALMON** = Color::from_RGB(233, 150, 122)
- static const [Color](#) **SALMON** = Color::from_RGB(250, 128, 114)
- static const [Color](#) **LIGHT_SALMON** = Color::from_RGB(255, 160, 122)
- static const [Color](#) **ORANGE_RED** = Color::from_RGB(255, 69, 0)
- static const [Color](#) **DARK_ORANGE** = Color::from_RGB(255, 140, 0)
- static const [Color](#) **ORANGE** = Color::from_RGB(255, 165, 0)
- static const [Color](#) **GOLD** = Color::from_RGB(255, 215, 0)
- static const [Color](#) **DARK_GOLDEN_ROD** = Color::from_RGB(184, 134, 11)
- static const [Color](#) **GOLDEN_ROD** = Color::from_RGB(218, 165, 32)
- static const [Color](#) **PALE_GOLDEN_ROD** = Color::from_RGB(238, 232, 170)
- static const [Color](#) **DARK_KHAKI** = Color::from_RGB(189, 183, 107)
- static const [Color](#) **KHAKI** = Color::from_RGB(240, 230, 140)
- static const [Color](#) **OLIVE** = Color::from_RGB(128, 128, 0)
- static const [Color](#) **YELLOW** = Color::from_RGB(255, 255, 0)
- static const [Color](#) **YELLOW_GREEN** = Color::from_RGB(154, 205, 50)
- static const [Color](#) **DARK_OLIVE_GREEN** = Color::from_RGB(85, 107, 47)
- static const [Color](#) **OLIVE_DRAB** = Color::from_RGB(107, 142, 35)
- static const [Color](#) **LAWN_GREEN** = Color::from_RGB(124, 252, 0)
- static const [Color](#) **CHART_REUSE** = Color::from_RGB(127, 255, 0)
- static const [Color](#) **GREEN_YELLOW** = Color::from_RGB(173, 255, 47)
- static const [Color](#) **DARK_GREEN** = Color::from_RGB(0, 100, 0)
- static const [Color](#) **GREEN** = Color::from_RGB(0, 128, 0)
- static const [Color](#) **FOREST_GREEN** = Color::from_RGB(34, 139, 34)
- static const [Color](#) **LIME** = Color::from_RGB(0, 255, 0)
- static const [Color](#) **LIME_GREEN** = Color::from_RGB(50, 205, 50)
- static const [Color](#) **LIGHT_GREEN** = Color::from_RGB(144, 238, 144)
- static const [Color](#) **PALE_GREEN** = Color::from_RGB(152, 251, 152)
- static const [Color](#) **DARK_SEA_GREEN** = Color::from_RGB(143, 188, 143)
- static const [Color](#) **MEDIUM_SPRING_GREEN** = Color::from_RGB(0, 250, 154)
- static const [Color](#) **SPRING_GREEN** = Color::from_RGB(0, 255, 127)
- static const [Color](#) **SEA_GREEN** = Color::from_RGB(46, 139, 87)
- static const [Color](#) **MEDIUM_AQUA_MARINE** = Color::from_RGB(102, 205, 170)

- static const [Color](#) **MEDIUM_SEA_GREEN** = Color::from_RGB(60, 179, 113)
- static const [Color](#) **LIGHT_SEA_GREEN** = Color::from_RGB(32, 178, 170)
- static const [Color](#) **DARK_SLATE_GRAY** = Color::from_RGB(47, 79, 79)
- static const [Color](#) **TEAL** = Color::from_RGB(0, 128, 128)
- static const [Color](#) **DARK_CYAN** = Color::from_RGB(0, 139, 139)
- static const [Color](#) **CYAN** = Color::from_RGB(0, 255, 255)
- static const [Color](#) **LIGHT_CYAN** = Color::from_RGB(244, 255, 255)
- static const [Color](#) **DARK_TURQUOISE** = Color::from_RGB(0, 206, 209)
- static const [Color](#) **TURQUOISE** = Color::from_RGB(64, 224, 208)
- static const [Color](#) **MEDIUM_TURQUOISE** = Color::from_RGB(72, 209, 204)
- static const [Color](#) **PALE_TORQUOISE** = Color::from_RGB(175, 238, 238)
- static const [Color](#) **AQUA_MARINE** = Color::from_RGB(127, 255, 212)
- static const [Color](#) **POWDER_BLUE** = Color::from_RGB(176, 224, 230)
- static const [Color](#) **CADET_BLUE** = Color::from_RGB(95, 158, 160)
- static const [Color](#) **STEEL_BLUE** = Color::from_RGB(70, 130, 180)
- static const [Color](#) **CORN_FLOWER_BLUE** = Color::from_RGB(100, 149, 237)
- static const [Color](#) **DEEP_SKY_BLUE** = Color::from_RGB(0, 191, 255)
- static const [Color](#) **DODGER_BLUE** = Color::from_RGB(30, 144, 255)
- static const [Color](#) **LIGHT_BLUE** = Color::from_RGB(173, 216, 230)
- static const [Color](#) **SKY_BLUE** = Color::from_RGB(135, 206, 235)
- static const [Color](#) **LIGHT_SKY_BLUE** = Color::from_RGB(135, 206, 250)
- static const [Color](#) **MIDNIGHT_BLUE** = Color::from_RGB(25, 25, 112)
- static const [Color](#) **NAVY** = Color::from_RGB(0, 0, 128)
- static const [Color](#) **DARK_BLUE** = Color::from_RGB(0, 0, 139)
- static const [Color](#) **MEDIUM_BLUE** = Color::from_RGB(0, 0, 205)
- static const [Color](#) **BLUE** = Color::from_RGB(0, 0, 255)
- static const [Color](#) **ROYAL_BLUE** = Color::from_RGB(65, 105, 225)
- static const [Color](#) **BLUE_VIOLET** = Color::from_RGB(138, 43, 226)
- static const [Color](#) **INDIGO** = Color::from_RGB(75, 0, 130)
- static const [Color](#) **DARK_SLATE_BLUE** = Color::from_RGB(72, 61, 139)
- static const [Color](#) **SLATE_BLUE** = Color::from_RGB(106, 90, 205)
- static const [Color](#) **MEDIUM_SLATE_BLUE** = Color::from_RGB(123, 104, 238)
- static const [Color](#) **MEDIUM_PURPLE** = Color::from_RGB(147, 112, 219)
- static const [Color](#) **DARK_MAGENTA** = Color::from_RGB(139, 0, 139)
- static const [Color](#) **DARK_VIOLET** = Color::from_RGB(148, 0, 211)
- static const [Color](#) **DARK_ORCHID** = Color::from_RGB(153, 50, 204)
- static const [Color](#) **MEDIUM_ORCHID** = Color::from_RGB(186, 85, 211)
- static const [Color](#) **PURPLE** = Color::from_RGB(128, 0, 128)
- static const [Color](#) **THISTLE** = Color::from_RGB(216, 191, 216)
- static const [Color](#) **PLUM** = Color::from_RGB(221, 160, 221)
- static const [Color](#) **VIOLET** = Color::from_RGB(238, 130, 238)
- static const [Color](#) **MAGENTA** = Color::from_RGB(255, 0, 255)
- static const [Color](#) **ORCHID** = Color::from_RGB(218, 112, 214)
- static const [Color](#) **MEDIUM_VIOLET_RED** = Color::from_RGB(199, 21, 133)
- static const [Color](#) **PALE_VIOLET_RED** = Color::from_RGB(219, 112, 147)
- static const [Color](#) **DEEP_PINK** = Color::from_RGB(255, 20, 147)
- static const [Color](#) **HOT_PINK** = Color::from_RGB(255, 105, 180)
- static const [Color](#) **LIGHT_PINK** = Color::from_RGB(255, 182, 193)
- static const [Color](#) **PINK** = Color::from_RGB(255, 192, 203)
- static const [Color](#) **ANTIQUE_WHITE** = Color::from_RGB(250, 235, 215)
- static const [Color](#) **BEIGE** = Color::from_RGB(245, 245, 220)
- static const [Color](#) **BISQUE** = Color::from_RGB(255, 228, 196)
- static const [Color](#) **BLANCHED_ALMOND** = Color::from_RGB(255, 235, 205)
- static const [Color](#) **WHEAT** = Color::from_RGB(245, 222, 179)
- static const [Color](#) **CORN_SILK** = Color::from_RGB(255, 248, 220)

- static const [Color](#) **LEMON_CHIFFON** = Color::from_RGB(255, 250, 205)
- static const [Color](#) **LIGHT_GOLDEN_ROD_YELLOW** = Color::from_RGB(250, 250, 210)
- static const [Color](#) **LIGHT_YELLOW** = Color::from_RGB(255, 255, 224)
- static const [Color](#) **SADDLE_BROWN** = Color::from_RGB(139, 69, 19)
- static const [Color](#) **SIENNA** = Color::from_RGB(160, 82, 45)
- static const [Color](#) **CHOCOLATE** = Color::from_RGB(210, 105, 30)
- static const [Color](#) **PERU** = Color::from_RGB(205, 133, 63)
- static const [Color](#) **SANDY_BROWN** = Color::from_RGB(244, 164, 96)
- static const [Color](#) **BURLY_WOOD** = Color::from_RGB(222, 184, 135)
- static const [Color](#) **TAN** = Color::from_RGB(210, 180, 140)
- static const [Color](#) **ROSY_BROWN** = Color::from_RGB(188, 143, 143)
- static const [Color](#) **MOCCASIN** = Color::from_RGB(255, 228, 181)
- static const [Color](#) **NAVAJO_WHITE** = Color::from_RGB(255, 222, 173)
- static const [Color](#) **PEACH_STUFF** = Color::from_RGB(255, 218, 185)
- static const [Color](#) **MISTY_ROSE** = Color::from_RGB(255, 228, 225)
- static const [Color](#) **LAVENDER_BLUSH** = Color::from_RGB(255, 240, 245)
- static const [Color](#) **LINEN** = Color::from_RGB(250, 240, 230)
- static const [Color](#) **OLD_LACE** = Color::from_RGB(253, 245, 230)
- static const [Color](#) **PAPAYA_WHIP** = Color::from_RGB(255, 239, 213)
- static const [Color](#) **SEA_SHELL** = Color::from_RGB(255, 245, 238)
- static const [Color](#) **MINT_CREAM** = Color::from_RGB(245, 255, 250)
- static const [Color](#) **SLATE_GRAY** = Color::from_RGB(112, 128, 144)
- static const [Color](#) **LIGHT_SLATE_GRAY** = Color::from_RGB(119, 136, 153)
- static const [Color](#) **LIGHT_STEEL_BLUE** = Color::from_RGB(176, 196, 222)
- static const [Color](#) **LAVENDER** = Color::from_RGB(230, 230, 250)
- static const [Color](#) **FLORAL_WHITE** = Color::from_RGB(255, 250, 240)
- static const [Color](#) **ALICE_BLUE** = Color::from_RGB(240, 248, 255)
- static const [Color](#) **GHOST_WHITE** = Color::from_RGB(248, 248, 255)
- static const [Color](#) **HONEYDEW** = Color::from_RGB(240, 255, 240)
- static const [Color](#) **IVORY** = Color::from_RGB(255, 255, 240)
- static const [Color](#) **AZURE** = Color::from_RGB(240, 255, 255)
- static const [Color](#) **SNOW** = Color::from_RGB(255, 250, 250)
- static const [Color](#) **BLACK** = Color::from_RGB(0, 0, 0)
- static const [Color](#) **DIM_GRAY** = Color::from_RGB(105, 105, 105)
- static const [Color](#) **GRAY** = Color::from_RGB(128, 128, 128)
- static const [Color](#) **DARK_GRAY** = Color::from_RGB(169, 169, 169)
- static const [Color](#) **SILVER** = Color::from_RGB(192, 192, 192)
- static const [Color](#) **LIGHT_GRAY** = Color::from_RGB(211, 211, 211)
- static const [Color](#) **GAINSBORO** = Color::from_RGB(220, 220, 220)
- static const [Color](#) **WHITE_SMOKE** = Color::from_RGB(245, 245, 245)
- static const [Color](#) **WHITE** = Color::from_RGB(255, 255, 255)

7.9.1 Detailed Description

An object used to represent an rgb color. r, g and b must all be real numbers between 0 and 1.

7.9.2 Constructor & Destructor Documentation

7.9.2.1 Color()

```
Color::Color (
    const double r,
    const double g,
    const double b )
```

Returns the (r,g,b) color.

Parameters

<i>r</i>	a real number between zero and one.
<i>g</i>	a real number between zero and one.
<i>b</i>	a real number between zero and one.

7.9.3 Member Function Documentation

7.9.3.1 mix()

```
Color Color::mix (
    const Color & A,
    const Color & B,
    const double alpha ) [static]
```

Returns a mix of colors A and B in proportions alpha and 1 - alpha.

Parameters

<i>A</i>	
<i>B</i>	
<i>alpha</i>	

Returns

Color

7.9.3.2 to_string()

```
std::string Color::to_string ( )
```

Returns a string representation of the color.

Returns

std::string

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

- CPGF/PGFBasics/PGFConf.hpp
- CPGF/PGFBasics/PGFConf.cpp

7.10 CPGF::DashPatterns Class Reference

This class provides a handful of useful predefined constants to express [DashPatterns](#).

```
#include <PGFConf.hpp>
```

Static Public Attributes

- static const std::vector< double > **SOLID** = std::vector<double>()
- static const std::vector< double > **DASHED** = std::vector<double>({0.3, 0.3})
- static const std::vector< double > **DOTTED** = std::vector<double>({0.1, 0.1})

7.10.1 Detailed Description

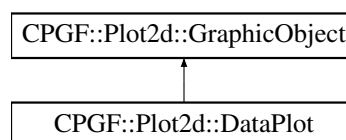
This class provides a handful of useful predefined constants to express [DashPatterns](#).

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

- CPGF/PGFBasics/PGFConf.hpp
- CPGF/PGFBasics/PGFConf.cpp

7.11 CPGF::Plot2d::DataPlot Class Reference

Inheritance diagram for CPGF::Plot2d::DataPlot:



Public Member Functions

- **DataPlot** (const std::vector< double > &Y, const std::vector< double > &X, const [Color](#) &color=[Color::BLUE](#), const double size=1, const double opacity=1, std::function< [Objects2d::Object2d](#)(const [AffineSpace::Point2d](#) pos, const [Color](#) &color, const double opacity, const double size)> shape=[Shapes::Circle](#), const std::string &legend="")
- **DataPlot** (const std::vector< double > &Y, const std::vector< double > &X, std::function< [Color](#)(unsigned int)> color, std::function< double(unsigned int)> size, std::function< double(unsigned int)> opacity, std::function< std::function< [Objects2d::Object2d](#)(const [AffineSpace::Point2d](#) pos, const [Color](#) &color, const double opacity, const double size)>(unsigned int)> shape, const std::string &legend="")
- **DataPlot** (const std::vector< double > &Y, const std::vector< double > &X, std::function< [Color](#)([AffineSpace::Point2d](#) &)> color, std::function< double([AffineSpace::Point2d](#) &)> size, std::function< double([AffineSpace::Point2d](#) &)> opacity, std::function< std::function< [Objects2d::Object2d](#)(const [AffineSpace::Point2d](#) pos, const [Color](#) &color, const double opacity, const double size)>([AffineSpace::Point2d](#) &)> shape, const std::string &legend="")
- **DataPlot** (const std::vector< [AffineSpace::Point2d](#) > &data, const [Color](#) &color=[Color::BLUE](#), const double size=1, const double opacity=1, std::function< [Objects2d::Object2d](#)(const [AffineSpace::Point2d](#) pos, const [Color](#) &color, const double opacity, const double size)> shape=[Shapes::Circle](#), const std::string &legend="")
- **DataPlot** (const std::vector< [AffineSpace::Point2d](#) > &data, std::function< [Color](#)(unsigned int)> color, std::function< double(unsigned int)> size, std::function< double(unsigned int)> opacity, std::function< std::function< [Objects2d::Object2d](#)(const [AffineSpace::Point2d](#) pos, const [Color](#) &color, const double opacity, const double size)>(unsigned int)> shape, const std::string &legend="")
- **DataPlot** (const std::vector< [AffineSpace::Point2d](#) > &data, std::function< [Color](#)([AffineSpace::Point2d](#) &)> color, std::function< double([AffineSpace::Point2d](#) &)> size, std::function< double([AffineSpace::Point2d](#) &)> opacity, std::function< std::function< [Objects2d::Object2d](#)(const [AffineSpace::Point2d](#) pos, const [Color](#) &color, const double opacity, const double size)>([AffineSpace::Point2d](#) &)> shape, const std::string &legend="")
- double [x_min](#) () const override
- double [x_max](#) () const override
- double [y_min](#) () const override
- double [y_max](#) () const override
- [Objects2d::Object2d](#) miniature (const [AffineSpace::Point2d](#) &pos) const override
- [Scene2d](#) render_to_scene (std::function< [AffineSpace::Point2d](#)(const [AffineSpace::Point2d](#) &P)> transform, const double x_min, const double x_max, const double y_min, const double y_max) const override

Public Attributes

- std::vector< [AffineSpace::Point2d](#) > **points**
- std::function< [Color](#)(unsigned int)> **color**
- std::function< double(unsigned int)> **size**
- std::function< double(unsigned int)> **opacity**
- std::function< std::function< [Objects2d::Object2d](#)(const [AffineSpace::Point2d](#) pos, const [Color](#) &color, const double opacity, const double size)>(unsigned int)> **shape**

Additional Inherited Members

7.11.1 Member Function Documentation

7.11.1.1 miniature()

```
Object2d DataPlot::miniature (
    const AffineSpace::Point2d & pos ) const [override], [virtual]
```

Implements [CPGF::Plot2d::GraphicObject](#).

7.11.1.2 render_to_scene()

```
Scene2d DataPlot::render_to_scene (
    std::function< AffineSpace::Point2d(const AffineSpace::Point2d &P)> transform,
    const double x_min,
    const double x_max,
    const double y_min,
    const double y_max ) const [override], [virtual]
```

Implements [CPGF::Plot2d::GraphicObject](#).

7.11.1.3 x_max()

```
double DataPlot::x_max ( ) const [override], [virtual]
```

Implements [CPGF::Plot2d::GraphicObject](#).

7.11.1.4 x_min()

```
double DataPlot::x_min ( ) const [override], [virtual]
```

Implements [CPGF::Plot2d::GraphicObject](#).

7.11.1.5 y_max()

```
double DataPlot::y_max ( ) const [override], [virtual]
```

Implements [CPGF::Plot2d::GraphicObject](#).

7.11.1.6 y_min()

```
double DataPlot::y_min ( ) const [override], [virtual]
```

Implements [CPGF::Plot2d::GraphicObject](#).

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

- CPGF/Plot2d/DataPlot.hpp
- CPGF/Plot2d/DataPlot.cpp

7.12 Math::DualNumber< K > Class Template Reference

Represents a dual number. A mathematical object written like $x = a + b$, where $b^2=0$.

```
#include <DualNumbers.hpp>
```

Public Member Functions

- **DualNumber ()**
Constructs the dual number zero.
- [DualNumber](#) (const K a)
Constructs a dual number whose real part is a and whose infinitesimal part equal to zero.
- [DualNumber](#) (const K a, const K b)
Construct a dual number with a as real part and b as infinitesimal part.
- `std::string to_string () const`
Returns a string representation of the dual number: a + b.

Public Attributes

- K a
Real part.
- K b
Infinitesimal part.

Static Public Attributes

- static const [DualNumber](#)< K > **epsilon** = [DualNumber](#)(0, 1)
The pure infinitesimal dual number.

Friends

- `DualNumber< K > operator+` (const `DualNumber< K >` &x, const `DualNumber< K >` &y)
Returns the sum of two dual numbers. $x + y = (x.a + y.a) + (x.b + y.b)$
- `DualNumber< K > operator+` (const `DualNumber< K >` &x, const K y)
Returns the sum of a dual number and a real number. $x + y = (x.a + y) + x.b$*
- `DualNumber< K > operator+` (const K x, const `DualNumber< K >` &y)
Returns the sum of a dual number and a real number. $x + y = (x + y.a) + y.b$*
- `DualNumber< K > operator-` (const `DualNumber< K >` &x, const `DualNumber< K >` &y)
Returns the subtraction of two dual numbers. $x - y = (x.a - y.a) + (x.b - y.b)$
- `DualNumber< K > operator-` (const `DualNumber< K >` &x, const K y)
Returns the subtraction of a dual number and a real number. $x - y = (x.a - y) + x.b$*
- `DualNumber< K > operator-` (const K x, const `DualNumber< K >` &y)
Returns the subtraction of a real number and a dual number. $x - y = (x - y.a) + y.b$*
- `DualNumber< K > operator*` (const `DualNumber< K >` &x, const `DualNumber< K >` &y)
*Returns the product of two dual numbers. $x*y = (x.a*y.a) + (x.a*y.b + x.b*y.a)$*
- `DualNumber< K > operator*` (const `DualNumber< K >` &x, const K y)
*Returns the product of a dual number times a real number. $x*y = (x.a*y) + (x.b*y)$*
- `DualNumber< K > operator*` (const K x, const `DualNumber< K >` &y)
*Returns the product of a dual number times a real number. $x*y = (x*y.a) + (x*y.b)$*
- `DualNumber< K > operator/` (const `DualNumber< K >` &x, const `DualNumber< K >` &y)
*Performs the division of two dual numbers. $x/y = (x.a/y.a) + (x.b*y.a - x.a*y.b)/(y.a*y.a) *$*
- `DualNumber< K > operator/` (const `DualNumber< K >` &x, const K y)
Returns the division of a dual number and a real number. $x/y = (x.a/y) + (x.b/y)$
- `DualNumber< K > operator/` (const K x, const `DualNumber< K >` &y)
*Returns the division of a real number by a dual number. $x/y = (x/y.a) - x*y.b/(y.a*y.a) *$*

7.12.1 Detailed Description

```
template<typename K>
class Math::DualNumber< K >
```

Represents a dual number. A mathematical object written like $x = a + b$, where $a^2 = 0$.

Every mathematical function on the C++ math library has been extended to act on DualNumbers. This allows automatic differentiation.

Template Parameters

<code>K</code>	
----------------	--

7.12.2 Constructor & Destructor Documentation

7.12.2.1 DualNumber() [1/2]

```
template<typename K>
DualNumber::DualNumber (
```

```
const K a )
```

Constructs a dual number whose real part is a and whose infinitesimal part equal to zero.

Parameters

a	the real part.
-----	----------------

7.12.2.2 DualNumber() [2/2]

```
template<typename K >
DualNumber::DualNumber (
    const K a,
    const K b )
```

Construct a dual number with a as real part and b as infinitesimal part.

Parameters

a	
b	

7.12.3 Member Function Documentation

7.12.3.1 to_string()

```
template<typename K >
std::string DualNumber::to_string
```

Returns a string representation of the dual number: $a + b$.

Returns

`std::string`

7.12.4 Friends And Related Function Documentation

7.12.4.1 operator* [1/3]

```
template<typename K >
DualNumber< K > operator* (
    const DualNumber< K > & x,
    const DualNumber< K > & y ) [friend]
```

Returns the product of two dual numbers. $x*y = (x.a*y.a) + (x.a*y.b + x.b*y.a)$

Parameters

x	
y	

Returns

DualNumber<K>

7.12.4.2 operator* [2/3]

```
template<typename K >
DualNumber< K > operator* (
    const DualNumber< K > & x,
    const K y ) [friend]
```

Returns the product of a dual number times a real number. $x*y = (x.a*y) + (x.b*y)$

Parameters

x	
y	

Returns

DualNumber<K>

7.12.4.3 operator* [3/3]

```
template<typename K >
DualNumber< K > operator* (
    const K x,
    const DualNumber< K > & y ) [friend]
```

Returns the product of a dual number times a real number. $x*y = (x*y.a) + (x*y.b)$

Parameters

x	
y	

Returns

DualNumber<K>

7.12.4.4 operator+ [1/3]

```
template<typename K >
DualNumber< K > operator+ (
    const DualNumber< K > & x,
    const DualNumber< K > & y ) [friend]
```

Returns the sum of two dual numbers. $x + y = (x.a + y.a) + (x.b + y.b)$

Parameters

x	
y	

Returns

DualNumber<K>

7.12.4.5 operator+ [2/3]

```
template<typename K >
DualNumber< K > operator+ (
    const DualNumber< K > & x,
    const K y ) [friend]
```

Returns the sum of a dual number and a real number. $x + y = (x.a + y) + x.b*$

Parameters

x	
y	

Returns

DualNumber<K>

7.12.4.6 operator+ [3/3]

```
template<typename K >
DualNumber< K > operator+ (
    const K x,
    const DualNumber< K > & y ) [friend]
```

Returns the sum of a dual number and a real number. $x + y = (x + y.a) + y.b*$

Parameters

x	
y	

Returns

DualNumber<K>

7.12.4.7 operator- [1/3]

```
template<typename K >
DualNumber< K > operator- (
    const DualNumber< K > & x,
    const DualNumber< K > & y ) [friend]
```

Returns the substraction of two dual numbers. $x - y = (x.a - y.a) + (x.b - y.b)$

Parameters

x	
y	

Returns

DualNumber<K>

7.12.4.8 operator- [2/3]

```
template<typename K >
DualNumber< K > operator- (
    const DualNumber< K > & x,
    const K y ) [friend]
```

Returns the substraction of a dual number and a real number. $x - y = (x.a - y) + x.b*$

Parameters

x	
y	

Returns

DualNumber<K>

7.12.4.9 operator- [3/3]

```
template<typename K >
DualNumber< K > operator- (
    const K x,
    const DualNumber< K > & y ) [friend]
```

Returns the subtraction of a real number and a dual number. $x - y = (x - y.a) + y.b*$

Parameters

x	
y	

Returns

DualNumber<K>

7.12.4.10 operator/ [1/3]

```
template<typename K >
DualNumber< K > operator/ (
    const DualNumber< K > & x,
    const DualNumber< K > & y ) [friend]
```

Performs the division of two dual numbers. $x/y = (x.a/y.a) + (x.b*y.a - x.a*y.b)/(y.a*y.a) *$

Parameters

x	
y	

Returns

DualNumber<K>

7.12.4.11 operator/ [2/3]

```
template<typename K >
DualNumber< K > operator/ (
    const DualNumber< K > & x,
    const K y ) [friend]
```

Returns the division of a dual number and a real number. $x/y = (x.a/y) + (x.b/y)$

Parameters

x	
y	

Returns

DualNumber<K>

7.12.4.12 operator/ [3/3]

```
template<typename K >
DualNumber< K > operator/ (
    const K x,
    const DualNumber< K > & y ) [friend]
```

Returns the division of a real number by a dual number. $x/y = (x/y.a) - x*y.b/(y.a*y.a) *$

Parameters

x	
y	

Returns

DualNumber<K>

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

- [Math/DualNumbers.hpp](#)
- [Math/DualNumbers.cpp](#)

7.13 Solvers::Gas::ExactRiemannSolver Class Reference

Solves the Riemann problem exactly for the 1D Euler Equations.

```
#include <ExactRiemannSolver.hpp>
```

Public Member Functions

- double [rho](#) (const double x, const double t) const
Gas density as a function of space and time.
- double [v](#) (const double x, const double t) const
Gas speed as a function of space and time.
- double [P](#) (const double x, const double t) const
Gas pressure as a function of space and time.
- double [S_max](#) () const
Returns the maximum absolute value of all wave speeds of the solution.
- [ExactRiemannSolver](#) (const double [rho_L](#), const double [v_L](#), const double [P_L](#), const double [rho_R](#), const double [v_R](#), const double [P_R](#), const double [gamma](#), const double tol)
Construct a new Exact Riemann Solver object.

Protected Member Functions

- double `f_L` (const double `P`) const
Function used in the computation of the pressure equation (left part).
- double `f_R` (const double `P`) const
Function used in the computation of the pressure equation (right part).
- double `f` (const double `P`) const
The pressure equation. P_{star} is the only solution of $f(P)=0$.
- double `df_LdP` (const double `P`) const
The derivative of `f_L` with respect to `P`.
- double `df_RdP` (const double `P`) const
The derivative of `f_R` with respect to `P`.
- double `dfdP` (const double `P`) const
The derivative of `f` with respect to `P`.
- double `rho_Lrf` (const double `S`) const
Returns the density in the left rarefaction region (between the head and the tail).
- double `v_Lrf` (const double `S`) const
Returns the speed in the left rarefaction region (between the head and the tail).
- double `P_Lrf` (const double `S`) const
Returns the pressure in the left rarefaction region (between the head and the tail).
- double `rho_Rrf` (const double `S`) const
Returns the density in the right rarefaction region (between the head and the tail).
- double `v_Rrf` (const double `S`) const
Returns the speed in the right rarefaction region (between the head and the tail).
- double `P_Rrf` (const double `S`) const
Returns the pressure in the right rarefaction region (between the head and the tail).
- double `rho_L0` (const double `S`) const
Returns the density for the right vacuum state Riemann problem.
- double `v_L0` (const double `S`) const
Returns the speed for the right vacuum state Riemann problem.
- double `P_L0` (const double `S`) const
Returns the pressure for the right vacuum state Riemann problem.
- double `rho_R0` (const double `S`) const
Returns the density for the left vacuum state Riemann problem.
- double `v_R0` (const double `S`) const
Returns the speed for the left vacuum state Riemann problem.
- double `P_R0` (const double `S`) const
Returns the pressure for the left vacuum state Riemann problem.

Protected Attributes

- double `gamma`
Gas adiabatic expansion coefficient.
- VacuumState `vacuum_state`
Stores what type of problem we have to solve.
- double `rho_L`
The gas density to the left of $x=0$.
- double `v_L`
The gas speed to the left of $x=0$.
- double `P_L`

- The gas pressure to the left of $x=0$.*

 - double **rho_R**
- The gas density to the right of $x=0$.*

 - double **v_R**
- The gas speed to the right of $x=0$.*

 - double **P_R**
- The gas pressure to the right of $x=0$.*

 - double **G1**
- $\gamma + 1$*

 - double **G2**
- $\gamma - 1$*

 - double **G3**
- $(\gamma - 1)/(\gamma + 1)$*

 - double **G4**
- $(\gamma - 1)/(2*\gamma)$*

 - double **G5**
- $(\gamma + 1)/(2*\gamma)$*

 - double **G6**
- $1/\gamma$*

 - double **G7**
- $2 / (\gamma + 1)$*

 - double **G8**
- $2 / (\gamma - 1)$*

 - double **A_L**
- $2 / (\gamma + 1) / \rho_L$*

 - double **A_R**
- $2 / (\gamma + 1) / \rho_R$*

 - double **B_L**
- $(\gamma - 1)/(\gamma + 1) * P_L$*

 - double **B_R**
- $(\gamma - 1)/(\gamma + 1) * P_R$*

 - double **c_L**
- The gas sound speed to the left of $x=0$.*

 - double **c_R**
- The gas sound speed to the right of $x=0$.*

 - double **S_L**
- Speed of the left shock wave.*

 - double **S_HL**
- Speed of the head of the left rarefaction wave.*

 - double **S_TL**
- Speed of the tail of the left rarefaction wave.*

 - double **S_starL**
- Speed of the vacuum front to the right of $x=0$.*

 - double **rho_starL**
- Density in the star region to the left of the contact discontinuity.*

 - double **v_star**
- Speed in the star region.*

 - double **P_star**
- Pressure in the star region.*

 - double **S_R**
- Speed of the right shock wave.*

- double **S_HR**
Speed of the head of the right rarefaction wave.
- double **S_TR**
Speed of the tail of the right rarefaction wave.
- double **S_starR**
Speed of the vacuum front to the left of $x=0$.
- double **rho_starR**
Density in the star region to the right of the contact discontinuity.

7.13.1 Detailed Description

Solves the Riemann problem exactly for the 1D Euler Equations.

Only valid when the area doesn't change. Computes the solution as a function of x and t .

The algorithm works by first computing the pressure in the star region numerically. Afterwards, the speed in the star region is calculated. Next, the character (shock wave or rarefaction wave) of the left and right waves is found out. Lastly, depending on the value of the quotient x/t , the correct zone is selected and the associated formulae are used to calculate the variables.

7.13.2 Constructor & Destructor Documentation

7.13.2.1 ExactRiemannSolver()

```
ExactRiemannSolver::ExactRiemannSolver (
    const double rho_L,
    const double v_L,
    const double P_L,
    const double rho_R,
    const double v_R,
    const double P_R,
    const double gamma,
    const double tol )
```

Construct a new Exact Riemann Solver object.

Parameters

<i>rho_L</i>	the gas density to the left of $x=0$.
<i>v_L</i>	the gas speed to the left of $x=0$.
<i>P_L</i>	the gas pressure to the left of $x=0$.
<i>rho_R</i>	the gas density to the right of $x=0$.
<i>v_R</i>	the gas speed to the right of $x=0$.
<i>P_R</i>	the gas pressure to the right of $x=0$.
<i>gamma</i>	the gas adiabatic expansion coefficient.
<i>tol</i>	the precision with which the pressure in the star region is determined.

7.13.3 Member Function Documentation

7.13.3.1 df_LdP()

```
double ExactRiemannSolver::df_LdP (
    const double P ) const [protected]
```

The derivative of f_L with respect to P .

Parameters

P	
-----	--

Returns

double

7.13.3.2 df_RdP()

```
double ExactRiemannSolver::df_RdP (
    const double P ) const [protected]
```

The derivative of f_R with respect to P .

Parameters

P	
-----	--

Returns

double

7.13.3.3 dfdP()

```
double ExactRiemannSolver::dfdP (
    const double P ) const [protected]
```

The derivative of f with respect to P .

Parameters

P	
-----	--

Returns

double

7.13.3.4 f()

```
double ExactRiemannSolver::f (
    const double P ) const [protected]
```

The pressure equation. P_{star} is the only solution of $f(P)=0$.

Parameters

P	
-----	--

Returns

double

7.13.3.5 f_L()

```
double ExactRiemannSolver::f_L (
    const double P ) const [protected]
```

Function used in the computation of the pressure equation (left part).

Parameters

P	
-----	--

Returns

double

7.13.3.6 f_R()

```
double ExactRiemannSolver::f_R (
    const double P ) const [protected]
```

Function used in the computation of the pressure equation (right part).

Parameters

P	
-----	--

Returns

double

7.13.3.7 $P()$

```
double ExactRiemannSolver::P (  
    const double x,  
    const double t ) const
```

Gas pressure as a function of space and time.

Parameters

x	spatial coordinate.
t	time coordinate.

Returns

double

7.13.3.8 $P_L0()$

```
double ExactRiemannSolver::P_L0 (  
    const double S ) const [protected]
```

Returns the pressure for the right vacuum state Riemann problem.

Parameters

S	
-----	--

Returns

double

7.13.3.9 P_Lrf()

```
double ExactRiemannSolver::P_Lrf (
    const double S ) const [protected]
```

Returns the pressure in the left rarefaction region (between the head and the tail).

Parameters

<i>S</i>	
----------	--

Returns

double

7.13.3.10 P_R0()

```
double ExactRiemannSolver::P_R0 (
    const double S ) const [protected]
```

Returns the pressure for the left vacuum state Riemann problem.

Parameters

<i>S</i>	
----------	--

Returns

double

7.13.3.11 P_Rrf()

```
double ExactRiemannSolver::P_Rrf (
    const double S ) const [protected]
```

Returns the pressure in the right rarefaction region (between the head and the tail).

Parameters

<i>S</i>	
----------	--

Returns

double

7.13.3.12 rho()

```
double ExactRiemannSolver::rho (
    const double x,
    const double t ) const
```

Gas density as a function of space and time.

Parameters

x	spatial coordinate.
t	time coordinate.

Returns

double

7.13.3.13 rho_L0()

```
double ExactRiemannSolver::rho_L0 (
    const double S ) const [protected]
```

Returns the density for the right vacuum state Riemann problem.

Parameters

S	
-----	--

Returns

double

7.13.3.14 rho_Lrf()

```
double ExactRiemannSolver::rho_Lrf (
    const double S ) const [protected]
```

Returns the density in the left rarefaction region (between the head and the tail).

Parameters

S	
-----	--

Returns

double

7.13.3.15 rho_R0()

```
double ExactRiemannSolver::rho_R0 (
    const double S ) const [protected]
```

Returns the density for the left vacuum state Riemann problem.

Parameters

S	
---	--

Returns

double

7.13.3.16 rho_Rrf()

```
double ExactRiemannSolver::rho_Rrf (
    const double S ) const [protected]
```

Returns the density in the right rarefaction region (between the head and the tail).

Parameters

S	
---	--

Returns

double

7.13.3.17 S_max()

```
double ExactRiemannSolver::S_max ( ) const
```

Returns the maximum absolute value of all wave speeds of the solution.

Returns

double

7.13.3.18 v()

```
double ExactRiemannSolver::v (
    const double x,
    const double t ) const
```

Gas speed as a function of space and time.

Parameters

x	spatial coordinate.
t	time coordinate.

Returns

double

7.13.3.19 v_L0()

```
double ExactRiemannSolver::v_L0 (
    const double S ) const [protected]
```

Returns the speed for the right vacuum state Riemann problem.

Parameters

S	
-----	--

Returns

double

7.13.3.20 v_Lrf()

```
double ExactRiemannSolver::v_Lrf (
    const double S ) const [protected]
```

Returns the speed in the left rarefaction region (between the head and the tail).

Parameters

S	
-----	--

Returns

double

7.13.3.21 v_R0()

```
double ExactRiemannSolver::v_R0 (
    const double S ) const [protected]
```

Returns the speed for the left vacuum state Riemann problem.

Parameters

<i>S</i>	
----------	--

Returns

double

7.13.3.22 v_Rrf()

```
double ExactRiemannSolver::v_Rrf (
    const double S ) const [protected]
```

Returns the speed in the right rarefaction region (between the head and the tail).

Parameters

<i>S</i>	
----------	--

Returns

double

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

- Solvers/Gas/ExactRiemannSolver.hpp
- Solvers/Gas/ExactRiemannSolver.cpp

7.14 Solvers::Gas::ExactSteadySolver Class Reference**Public Member Functions**

- [ExactSteadySolver](#) (const double [rho_0](#), const double [P_0](#), const double [T_0](#), const double [M_x](#), const double [A_x](#), const double [gamma](#), const SolutionType [solution_type](#))

- *Construct a new Exact Steady Solver object.*
- double **M** (const double A) const
Returns Mach number as a function of area.
- double **rho** (const double A) const
Returns density as a function of area.
- double **v** (const double A) const
Returns speed as a function of area.
- double **P** (const double A) const
Returns pressure as a function of area.
- double **T** (const double A) const
Returns temperature as a function of area.

Protected Attributes

- SolutionType **solution_type**
Stores whether the subsonic or the supersonic solution is found.
- double **C**
The constant $f(M(x))A(x)$.
- double **rho_0**
Rest density of the gas.
- double **P_0**
Rest pressure of the gas.
- double **T_0**
Rest temperature of the gas.
- double **R**
Gas constant.
- double **gamma**
Gas adiabatic expansion coefficient.
- double **G1**
 $(\text{gamma} - 1) / 2$
- double **G2**
 $\text{gamma} / (\text{gamma} - 1)$
- double **G3**
 $1 / (\text{gamma} - 1)$
- double **G4**
 $(\text{gamma} + 1) / (2 * (\text{gamma} - 1))$

7.14.1 Constructor & Destructor Documentation

7.14.1.1 ExactSteadySolver()

```
ExactSteadySolver::ExactSteadySolver (
    const double rho_0,
    const double P_0,
    const double T_0,
    const double M_x,
    const double A_x,
    const double gamma,
    const SolutionType solution_type )
```

Construct a new Exact Steady Solver object.

Parameters

<i>rho_0</i>	rest density.
<i>P_0</i>	rest pressure.
<i>T_0</i>	rest temperature.
<i>M_x</i>	Mach number at a point x.
<i>A_x</i>	area at a point x.
<i>gamma</i>	gas adiabatic expansion coefficient.
<i>solution_type</i>	whether to compute the subsonic or the supersonic solution.

7.14.2 Member Function Documentation

7.14.2.1 M()

```
double ExactSteadySolver::M (  
    const double A ) const
```

Returns Mach number as a function of area.

Parameters

<i>A</i>	area.
----------	-------

Returns

double

7.14.2.2 P()

```
double ExactSteadySolver::P (  
    const double A ) const
```

Returns pressure as a function of area.

Parameters

<i>A</i>	area.
----------	-------

Returns

double

7.14.2.3 rho()

```
double ExactSteadySolver::rho (  
    const double A ) const
```

Returns density as a function of area.

Parameters

<i>A</i>	area.
----------	-------

Returns

double

7.14.2.4 T()

```
double ExactSteadySolver::T (  
    const double A ) const
```

Returns temperature as a function of area.

Parameters

<i>A</i>	area.
----------	-------

Returns

double

7.14.2.5 v()

```
double ExactSteadySolver::v (  
    const double A ) const
```

Returns speed as a function of area.

Parameters

<i>A</i>	area.
----------	-------

Returns

double

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

- Solvers/Gas/ExactSteadySolver.hpp
- Solvers/Gas/ExactSteadySolver.cpp

7.15 Utilities::FileArray< T > Class Template Reference

Classes

- class [Iterator](#)

Public Member Functions

- unsigned long [size](#) () const
Returns number of elements in the array.
- T **operator[]** (const unsigned long i) const
- [Iterator](#) **operator[]** (const unsigned long i)
- T **at** (const unsigned long i) const
- [Iterator](#) **at** (const unsigned long i)
- T **front** () const
- [Iterator](#) **front** ()
- T **back** () const
- [Iterator](#) **back** ()
- [Iterator](#) **begin** ()
- [Iterator](#) **end** ()
- [Iterator](#) **rbegin** ()
- [Iterator](#) **rend** ()
- void **push_back** (const T &val)
- void **pop_back** ()
- void **clear** ()
- **FileArray** (char *file, const bool temp=false)
- **FileArray** (const std::string &file, const bool temp=false)
- **FileArray** (const std::vector< T > &vector)
- **FileArray** (char *file, const std::vector< T > &vector, const bool temp=false)
- **FileArray** (const std::string &file, const std::vector< T > &vector, const bool temp=false)
- **FileArray** (T *array, unsigned int [N](#))
- **FileArray** (char *file, T *array, unsigned int [N](#), const bool temp=false)
- **FileArray** (const std::string &file, T *array, unsigned int [N](#), const bool temp=false)
- **FileArray** (const std::initializer_list< T > &list)
- **FileArray** (char *file, const std::initializer_list< T > &list, const bool temp=false)
- **FileArray** (const std::string &file, const std::initializer_list< T > &list, const bool temp=false)
- **FileArray** (const [FileArray](#)< T > &f_array)=delete
- [FileArray](#) & **operator=** (const [FileArray](#)< T > &f_array)=delete

Public Attributes

- FILE * **file**

Protected Attributes

- unsigned long **N**
Length of the array.
- std::string **filename**
- bool **temp**
- std::vector< unsigned long > **indexes**

7.15.1 Member Function Documentation

7.15.1.1 size()

```
template<typename T >
unsigned long FileArray::size
```

Returns number of elements in the array.

Returns

unsigned long

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

- Utilities/FileArray.hpp
- Utilities/FileArray.cpp

7.16 GasBoundaryConditions Class Reference

This class is used to store the boundary conditions associated to a simulation.

```
#include <Simulation.hpp>
```

Public Member Functions

- [GasBoundaryConditions](#) (const GasBoundaryConditionsType &[left](#)=GasBoundaryConditionsType::WALL, const GasBoundaryConditionsType &[right](#)=GasBoundaryConditionsType::WALL, std::function< double(double t)> [left_condition_1](#)=[](double t){return 0;}, std::function< double(double t)> [left_condition_2](#)=[](double t){return 0;}, std::function< double(double t)> [left_condition_3](#)=[](double t){return 0;}, std::function< double(double t)> [right_condition_1](#)=[](double t){return 0;}, std::function< double(double t)> [right_condition_2](#)=[](double t){return 0;}, std::function< double(double t)> [right_condition_3](#)=[](double t){return 0;})

Construct a new Gas Boundary Conditions object.

Public Attributes

- **GasBoundaryConditionsType left**
The type of boundary conditions to the left of the domain.
- **GasBoundaryConditionsType right**
The type of boundary conditions to the right of the domain.
- **std::function< double(double t)> left_condition_1**
Function of time used to return the value of the first left boundary condition.
- **std::function< double(double t)> left_condition_2**
Function of time used to return the value of the second left boundary condition.
- **std::function< double(double t)> left_condition_3**
Function of time used to return the value of the third left boundary condition.
- **std::function< double(double t)> right_condition_1**
Function of time used to return the value of the first right boundary condition.
- **std::function< double(double t)> right_condition_2**
Function of time used to return the value of the second right boundary condition.
- **std::function< double(double t)> right_condition_3**
Function of time used to return the value of the third right boundary condition.

7.16.1 Detailed Description

This class is used to store the boundary conditions associated to a simulation.

7.16.2 Constructor & Destructor Documentation

7.16.2.1 GasBoundaryConditions()

```
GasBoundaryConditions::GasBoundaryConditions (
    const GasBoundaryConditionsType & left = GasBoundaryConditionsType::WALL,
    const GasBoundaryConditionsType & right = GasBoundaryConditionsType::WALL,
    std::function< double(double t)> left_condition_1 = [] (double t){return 0;},
    std::function< double(double t)> left_condition_2 = [] (double t){return 0;},
    std::function< double(double t)> left_condition_3 = [] (double t){return 0;},
    std::function< double(double t)> right_condition_1 = [] (double t){return 0;},
    std::function< double(double t)> right_condition_2 = [] (double t){return 0;},
    std::function< double(double t)> right_condition_3 = [] (double t){return 0;} )
```

Construct a new Gas Boundary Conditions object.

Parameters

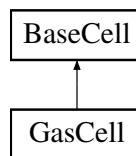
<i>left</i>	the type of the left boundary condition.
<i>right</i>	the type of the right boundary condition.
<i>left_condition_1</i>	function of time used for the first left boundary condition.
<i>left_condition_2</i>	function of time used for the second left boundary condition.
<i>left_condition_3</i>	function of time used for the third left boundary condition.
<i>right_condition_1</i>	function of time used for the first right boundary condition.
<i>right_condition_2</i>	function of time used for the second right boundary condition.
<i>right_condition_3</i>	function of time used for the third right boundary condition.

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

- Simulation.hpp
- Simulation.cpp

7.17 GasCell Class Reference

Inheritance diagram for GasCell:



Public Member Functions

- [GasCell](#) (const [Math::Vector](#)< double > &[U](#), double [a](#), double [b](#), const double [A](#), [Chemistry::SolidGasReaction](#) *[QR](#))
Construct a new Cell object.
- [GasCell](#) (FILE *file, [Chemistry::SolidGasReaction](#) *[QR](#))
Construct a new Cell object from file.
- [Math::Vector](#)< double > [F](#) () const
Returns the flux evaluated at the conserved variables of the cell, that is, $F(U)$.
- void [update](#) ()
Calculates all cell variables (ρ , T , P , ...) from the conserved quantities.
- void [read_from_file](#) (FILE *file)
Reads all cell values from file and calls [update\(\)](#)
- std::string [to_string](#) () const
Returns a string representation of the [GasCell](#).

Public Attributes

- [GasCell](#) * [right_neighbour](#)
Right neighbour of the cell.
- [GasCell](#) * [left_neighbour](#)
Left neighbour of the cell.
- double [rho](#)
Gas density.
- double [c](#)
Gas sound speed.
- double [v](#)
Gas speed.
- double [T](#)
Gas temperature.
- double [P](#)
Gas pressure.
- double [H](#)
Gas total enthalpy.
- double [E](#)
Gas total energy.
- double [M](#)
Mach number.

7.17.1 Constructor & Destructor Documentation

7.17.1.1 GasCell() [1/2]

```
GasCell::GasCell (
    const Math::Vector< double > & U,
    double a,
    double b,
    const double A,
    Chemistry::SolidGasReaction * QR )
```

Construct a new Cell object.

Parameters

<i>U</i>	the vector of conserved variables.
<i>a</i>	the left limit of the cell.
<i>b</i>	the right limit of the cell.
<i>A</i>	the area of the cell.
<i>QR</i>	a reference to the quchemical reaction.

7.17.1.2 GasCell() [2/2]

```
GasCell::GasCell (
    FILE * file,
    Chemistry::SolidGasReaction * QR )
```

Construct a new Cell object from file.

Parameters

<i>file</i>	
-------------	--

7.17.2 Member Function Documentation

7.17.2.1 read_from_file()

```
void GasCell::read_from_file (
    FILE * file )
```

Reads all cell values from file and calls [update\(\)](#)

Parameters

<i>file</i>	
-------------	--

7.17.2.2 to_string()

```
std::string GasCell::to_string ( ) const
```

Returns a string representation of the [GasCell](#).

Returns

std::string

7.17.3 Member Data Documentation

7.17.3.1 c

```
double GasCell::c
```

Gas sound speed.

Returns

double

7.17.3.2 P

```
double GasCell::P
```

Gas pressure.

Returns

double

7.17.3.3 rho

```
double GasCell::rho
```

Gas density.

Returns

double

7.17.3.4 T

```
double GasCell::T
```

Gas temperature.

Returns

double

7.17.3.5 v

```
double GasCell::v
```

Gas speed.

Returns

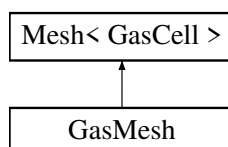
double

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

- [Mesh/Cell.hpp](#)
- [Mesh/Cell.cpp](#)

7.18 GasMesh Class Reference

Inheritance diagram for GasMesh:



Public Member Functions

- **GasMesh** (**GasCell** ***first_cell**=nullptr, **GasCell** ***last_cell**=nullptr, const **Chemistry::SolidGasReaction** &**QR**=**Chemistry::SolidGasReaction**(), std::function< double(const double **x**)> **A_func**=[],(double **x**){return 0;}, const double **detail_subdivide_threshold**=0.01, const double **detail_merge_threshold**=0.0005, const double **max_length_factor**=1./20, const double **min_length_factor**=1./10000, const double **boundary_cell_max_length_factor**=1./10000)
Construct a new Gas **Mesh** object.
- std::vector< double > **c** () const
Returns gas sound speed of all cells in the mesh.
- std::vector< double > **rho** () const
Returns the density of all cells in the mesh.
- std::vector< double > **v** () const
Returns gas speed of all cells in the mesh.
- std::vector< double > **T** () const
Returns the temperature of all cells in the mesh.
- std::vector< double > **P** () const
Returns the pressure of all cells in the mesh.
- std::vector< double > **M** () const
Returns the Mach number of all cells in the mesh.

Additional Inherited Members

7.18.1 Constructor & Destructor Documentation

7.18.1.1 GasMesh()

```
GasMesh::GasMesh (
    GasCell * first_cell = nullptr,
    GasCell * last_cell = nullptr,
    const Chemistry::SolidGasReaction & QR = Chemistry::SolidGasReaction(),
    std::function< double(const double x)> A_func = [] (double x){return 0;},
    const double detail_subdivide_threshold = 0.01,
    const double detail_merge_threshold = 0.0005,
    const double max_length_factor = 1./20,
    const double min_length_factor = 1./10000,
    const double boundary_cell_max_length_factor = 1./1000 )
```

Construct a new Gas **Mesh** object.

Parameters

<i>first_cell</i>	pointer to the first cell.
<i>last_cell</i>	pointer to the last cell.
<i>QR</i>	chemical reaction.
<i>A_func</i>	area function.
<i>detail_subdivide_threshold</i>	determines when cells are subdivided.
<i>detail_merge_threshold</i>	determines when cells are merged.
<i>max_length_factor</i>	limits the maximum length of a created cell.
<i>min_length_factor</i>	limits the minimum length of a created cell.
<i>boundary_cell_max_length_factor</i>	limits the maximum length of cells near the boundary.

7.18.2 Member Function Documentation

7.18.2.1 c()

```
std::vector< double > GasMesh::c ( ) const
```

Returns gas sound speed of all cells in the mesh.

Returns

```
std::vector<double>
```

7.18.2.2 M()

```
std::vector< double > GasMesh::M ( ) const
```

Returns the Mach number of all cells in the mesh.

Returns

```
std::vector<double>
```

7.18.2.3 P()

```
std::vector< double > GasMesh::P ( ) const
```

Returns the pressure of all cells in the mesh.

Returns

```
std::vector<double>
```

7.18.2.4 rho()

```
std::vector< double > GasMesh::rho ( ) const
```

Returns the density of all cells in the mesh.

Returns

```
std::vector<double>
```

7.18.2.5 T()

```
std::vector< double > GasMesh::T ( ) const
```

Returns the temperature of all cells in the mesh.

Returns

```
std::vector<double>
```

7.18.2.6 v()

```
std::vector< double > GasMesh::v ( ) const
```

Returns gas speed of all cells in the mesh.

Returns

```
std::vector<double>
```

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

- Mesh/[Mesh.hpp](#)
- Mesh/[Mesh.cpp](#)

7.19 CPGF::Plot2d::Graphic Class Reference

Public Member Functions

- void **add** ([GraphicObject](#) *object, [Axis](#) *Y, [Axis](#) *X)
- [Scene2d](#) **render_to_scene** () const
- **Graphic** (const bool show_legend=false)

Public Attributes

- std::vector< std::tuple< [GraphicObject](#) *, [Axis](#) *, [Axis](#) * > > **graphic_objects**
- bool **show_legend**
- LegendPosition **legend_position**
- std::vector< [Axis](#) * > **axes**

Static Public Attributes

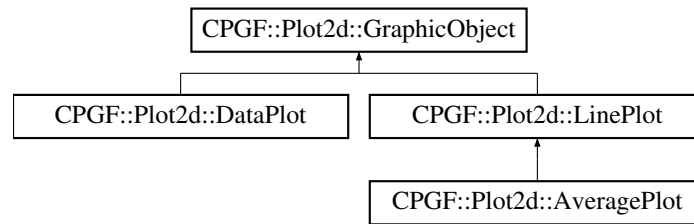
- constexpr static const double **LEGEND_MARGIN** = 0.25
- constexpr static const double **LEGEND_VERTICAL_DISPLACEMENT_PER_LINE** = 0.25
- constexpr static const double **CHARACTER_WIDTH** = 0.1

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

- CPGF/Plot2d/[Graphic.hpp](#)
- CPGF/Plot2d/[Graphic.cpp](#)

7.20 CPGF::Plot2d::GraphicObject Class Reference

Inheritance diagram for CPGF::Plot2d::GraphicObject:



Public Member Functions

- virtual double **x_min** () const =0
- virtual double **x_max** () const =0
- virtual double **y_min** () const =0
- virtual double **y_max** () const =0
- virtual **Objects2d::Object2d miniature** (const [AffineSpace::Point2d](#) &pos) const =0
- virtual **Scene2d render_to_scene** (std::function< [AffineSpace::Point2d](#)(const [AffineSpace::Point2d](#) &P)> transform, const double x_min, const double x_max, const double y_min, const double y_max) const =0

Public Attributes

- std::string **legend**

Static Public Attributes

- constexpr static const double **MINIATURE_HALF_WIDTH** = 0.5

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

- CPGF/Plot2d/GraphicObject.hpp

7.21 Mesh< Cell >::Iterator Class Reference

[Iterator](#) object to loop through all the cells of the mesh.

```
#include <Mesh.hpp>
```


Public Member Functions

- **Iterator** ()
Construct a new [Iterator](#) object that points to nullptr.
- **Iterator** (Cell *[cell](#))
Construct a new [Iterator](#) object that points to cell.
- **Iterator** (const [Iterator](#) &J)
Copy constructor.
- **Iterator & operator=** (const [Iterator](#) &J)
Assignment operator.
- **Iterator operator+** (const int i)
Returns the cell placed i positions to the right.
- **Iterator operator-** (const int i)
Returns the cell placed i positions to the left.
- **Iterator operator++** (int)
Advance one cell to the right in the mesh.
- **Iterator & operator++** ()
Advance one cell to the right in the mesh.
- **Iterator operator--** (int)
Advance one cell to the left in the mesh.
- **Iterator & operator--** ()
Advance one cell to the left in the mesh.
- **Cell & operator*** ()
Dereference operator returns the cell it points to.
- **Cell * operator->** ()
Dereference operator returns the cell it points to.
- **operator Cell *** () const
Implicit conversion into cell pointer.
- **bool operator==** (const [Iterator](#) &J)
Returns true if both iterators point to the same cell and false otherwise.
- **bool operator!=** (const [Iterator](#) &J)
Returns false if both iterators point to the same cell and true otherwise.

Public Attributes

- **Cell * [cell](#)**
Pointer to the current cell.

7.21.1 Detailed Description

```
template<class Cell>
class Mesh< Cell >::Iterator
```

[Iterator](#) object to loop through all the cells of the mesh.

7.21.2 Constructor & Destructor Documentation

7.21.2.1 `Iterator()` [1/2]

```
template<class Cell >
Mesh< Cell >::Iterator::Iterator (
    Cell * cell )
```

Construct a new `Iterator` object that points to *cell*.

Parameters

<i>cell</i>	
-------------	--

7.21.2.2 `Iterator()` [2/2]

```
template<class Cell >
Mesh< Cell >::Iterator::Iterator (
    const Iterator & J )
```

Copy constructor.

Parameters

<i>J</i>	
----------	--

7.21.3 Member Function Documentation

7.21.3.1 `operator Cell *()`

```
template<class Cell >
Mesh< Cell >::Iterator::operator Cell *
```

Implicit conversion into cell pointer.

Returns

`Cell*`

7.21.3.2 `operator!=(())`

```
template<class Cell >
bool Mesh< Cell >::Iterator::operator!= (
    const Iterator & J )
```

Returns false if both iterators point to the same cell and true otherwise.

Parameters

<i>I</i>	
<i>J</i>	

Returns

true
false

7.21.3.3 operator*()

```
template<class Cell >  
Cell & Mesh< Cell >::Iterator::operator*
```

Dereference operator returns the cell it points to.

Returns

Cell&

7.21.3.4 operator+()

```
template<class Cell >  
Mesh< Cell >::Iterator Mesh< Cell >::Iterator::operator+ (  
    const int i )
```

Returns the cell placed i positions to the right.

Parameters

<i>i</i>	
----------	--

Returns

Iterator&

7.21.3.5 operator++() [1/2]

```
template<class Cell >  
Mesh< Cell >::Iterator & Mesh< Cell >::Iterator::operator++
```

Advance one cell to the right in the mesh.

Returns

[Iterator](#)&

7.21.3.6 operator++() [2/2]

```
template<class Cell >
Mesh< Cell >::Iterator Mesh< Cell >::Iterator::operator++ (
    int )
```

Advance one cell to the right in the mesh.

Returns

[Iterator](#)&

7.21.3.7 operator-()

```
template<class Cell >
Mesh< Cell >::Iterator Mesh< Cell >::Iterator::operator- (
    const int i )
```

Returns the cell placed i positions to the left.

Parameters

<i>i</i>	
----------	--

Returns

[Iterator](#)&

7.21.3.8 operator--() [1/2]

```
template<class Cell >
Mesh< Cell >::Iterator & Mesh< Cell >::Iterator::operator--
```

Advance one cell to the left in the mesh.

Returns

[Iterator](#)&

7.21.3.9 operator--() [2/2]

```
template<class Cell >
Mesh< Cell >::Iterator Mesh< Cell >::Iterator::operator-- (
    int )
```

Advance one cell to the left in the mesh.

Returns

Iterator&

7.21.3.10 operator->()

```
template<class Cell >
Cell * Mesh< Cell >::Iterator::operator->
```

Dereference operator returns the cell it points to.

Returns

Cell*

7.21.3.11 operator=()

```
template<class Cell >
Mesh< Cell >::Iterator & Mesh< Cell >::Iterator::operator= (
    const Iterator & J )
```

Assignment operator.

Parameters

<i>J</i>	
----------	--

Returns

Iterator&

7.21.3.12 operator==()

```
template<class Cell >
bool Mesh< Cell >::Iterator::operator== (
    const Iterator & J )
```

Returns true if both iterators point to the same cell and false otherwise.

Parameters

<i>I</i>	
<i>J</i>	

Returns

true
false

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

- Mesh/[Mesh.hpp](#)
- Mesh/Mesh.cpp

7.22 Utilities::FileArray< T >::Iterator Class Reference

Public Member Functions

- **Iterator** ([FileArray](#) *file_array, const unsigned long i=0)
- bool **operator==** (const [Iterator](#) &J)
- bool **operator!=** (const [Iterator](#) &J)
- **operator T** () const
- [Iterator](#) & **operator=** (const T &val)
- [Iterator](#) & **operator++** ()
- [Iterator](#) **operator++** (int)
- [Iterator](#) & **operator--** ()
- [Iterator](#) **operator--** (int)
- [Iterator](#) & **operator+** (const unsigned long j)
- [Iterator](#) & **operator-** (const unsigned long j)

Public Attributes

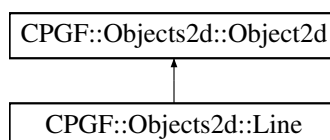
- [FileArray](#) * **file_array**
- unsigned long **i**

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

- Utilities/FileArray.hpp
- Utilities/FileArray.cpp

7.23 CPGF::Objects2d::Line Class Reference

Inheritance diagram for CPGF::Objects2d::Line:



Public Member Functions

- **Line** (const [AffineSpace::Point2d](#) &A, const [AffineSpace::Point2d](#) &B, const [Color](#) &color=[Color::BLACK](#), const double opacity=1, const double line_width=[LineWidth::SEMITHICK](#), const std::vector< double > &dash_pattern=[DashPatterns::SOLID](#))
- **Line** (const std::vector< [AffineSpace::Point2d](#) > &points, const [Color](#) &color=[Color::BLACK](#), const double opacity=1, const double line_width=[LineWidth::SEMITHICK](#), const std::vector< double > &dash_↵ pattern=[DashPatterns::SOLID](#))
- [AffineSpace::Point2d](#) & **start** ()
- [AffineSpace::Point2d](#) & **end** ()

Static Protected Member Functions

- static [Object2d](#) **builder** (const std::vector< [AffineSpace::Point2d](#) > &points, const [Color](#) &color, const double opacity, const double line_width, const std::vector< double > &dash_pattern)

Additional Inherited Members

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

- CPGF/Objects2d/BasicGeometries.hpp
- CPGF/Objects2d/BasicGeometries.cpp

7.24 Math::Interpolation::LinearInterpolation Class Reference

Public Member Functions

- **LinearInterpolation** (const std::vector< double > &Y, const std::vector< double > &X)
- **LinearInterpolation** (const double *Y, const double *X, const unsigned int N)
- **LinearInterpolation** (const [LinearInterpolation](#) &l)
- [LinearInterpolation](#) & **operator=** (const [LinearInterpolation](#) &l)
- double **operator()** (const double x) const

Protected Attributes

- unsigned int **N**
- double * **x_part**
- double * **b**
- double * **c**

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

- Math/Interpolation.hpp
- Math/Interpolation.cpp

7.25 Math::AlgebraicSolvers::LinearSystemSolver Class Reference

Solves linear systems of the form $Ax=b$ through LU decomposition with partial pivoting.

```
#include <AlgebraicSolvers.hpp>
```

Public Member Functions

- `Vector< double > solve` (const `Vector< double >` &b) const
Returns the solution of the system $Ax=b$.
- `LinearSystemSolver` (const `Matrix< double >` &A)
Construct a new Linear System Solver object for the coefficient matrix A. It computes its LU decomposition with partial pivoting.

Protected Attributes

- `Matrix< double > L`
The lower diagonal matrix of the LU decomposition of A ($PA=LU$).
- `Matrix< double > U`
The upper diagonal matrix of the LU decomposition of A ($PA=LU$).
- `std::vector< unsigned int > P`
The permutation vector.

7.25.1 Detailed Description

Solves linear systems of the form $Ax=b$ through LU decomposition with partial pivoting.

For efficiency reasons, the coefficient matrix must be supplied first, then the LU decomposition is done and, afterwards, we may solve any system of the form $Ax=b$ without having to recompute the LU decomposition of A.

7.25.2 Constructor & Destructor Documentation

7.25.2.1 LinearSystemSolver()

```
LinearSystemSolver::LinearSystemSolver (
    const Matrix< double > & A ) [explicit]
```

Construct a new Linear System Solver object for the coefficient matrix A. It computes its LU decomposition with partial pivoting.

Parameters

A	
---	--

7.25.3 Member Function Documentation

7.25.3.1 solve()

```
Vector< double > LinearSystemSolver::solve (
    const Vector< double > & b ) const
```

Returns the solution of the system $Ax=b$.

Parameters

b	the independent vector.
-----	-------------------------

Returns

Vector<double>

7.25.4 Member Data Documentation

7.25.4.1 P

```
std::vector<unsigned int> Math::AlgebraicSolvers::LinearSystemSolver::P [protected]
```

The permutation vector.

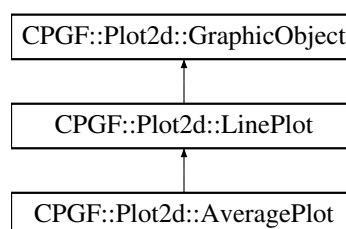
It is used to keep track of the permutations done when applying Gaussian. The i -th component of the permutation vector stores which row of the matrix A is now the i -th row of LU .

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

- [Math/AlgebraicSolvers.hpp](#)
- [Math/AlgebraicSolvers.cpp](#)

7.26 CPGF::Plot2d::LinePlot Class Reference

Inheritance diagram for CPGF::Plot2d::LinePlot:



Public Member Functions

- **LinePlot** (const std::vector< double > &Y, const std::vector< double > &X, const [Color](#) &color=[Color::BLUE](#), const double line_width=[LineWidth::THIN](#), const double opacity=1, const std::vector< double > &dash_↵ pattern=[DashPatterns::SOLID](#), const std::string &legend="")
- **LinePlot** (const std::vector< double > &Y, const std::vector< double > &X, std::function< [Color](#)(unsigned int)> color, std::function< double(unsigned int)> line_width, std::function< double(unsigned int)> opacity, std::function< std::vector< double >(unsigned int)> dash_pattern, const std::string &legend="")
- **LinePlot** (const std::vector< double > &Y, const std::vector< double > &X, std::function< [Color](#)([AffineSpace::Point2d](#) &)> color, std::function< double([AffineSpace::Point2d](#) &)> line_width, std::function< double([AffineSpace::Point2d](#) &)> opacity, std::function< std::vector< double >([AffineSpace::Point2d](#) &)> dash_pattern, const std::string &legend="")
- **LinePlot** (const std::vector< [AffineSpace::Point2d](#) > &data, const [Color](#) &color=[Color::BLUE](#), const double line_width=[LineWidth::THIN](#), const double opacity=1, const std::vector< double > &dash_pattern=[Dash↵ Patterns::SOLID](#), const std::string &legend="")
- **LinePlot** (const std::vector< [AffineSpace::Point2d](#) > &data, std::function< [Color](#)(unsigned int)> color, std::↵ function< double(unsigned int)> line_width, std::function< double(unsigned int)> opacity, std::function< std::vector< double >(unsigned int)> dash_pattern, const std::string &legend="")
- **LinePlot** (const std::vector< [AffineSpace::Point2d](#) > &data, std::function< [Color](#)([AffineSpace::Point2d](#) &)> color, std::function< double([AffineSpace::Point2d](#) &)> line_width, std::function< double([AffineSpace::Point2d](#) &)> opacity, std::function< std::vector< double >([AffineSpace::Point2d](#) &)> dash_pattern, const std::string &legend="")
- double [x_min](#) () const override
- double [x_max](#) () const override
- double [y_min](#) () const override
- double [y_max](#) () const override
- [Objects2d::Object2d](#) miniature (const [AffineSpace::Point2d](#) &pos) const override
- [Scene2d](#) render_to_scene (std::function< [AffineSpace::Point2d](#)(const [AffineSpace::Point2d](#) &P)> transform, const double x_min, const double x_max, const double y_min, const double y_max) const override

Public Attributes

- std::vector< [AffineSpace::Point2d](#) > **points**
- std::function< [Color](#)(unsigned int)> **color**
- std::function< double(unsigned int)> **line_width**
- std::function< double(unsigned int)> **opacity**
- std::function< std::vector< double >(unsigned int)> **dash_pattern**

Protected Attributes

- bool **const_parameters**

Additional Inherited Members

7.26.1 Member Function Documentation

7.26.1.1 miniature()

```
Object2d LinePlot::miniature (
    const AffineSpace::Point2d & pos ) const [override], [virtual]
```

Implements [CPGF::Plot2d::GraphicObject](#).

7.26.1.2 render_to_scene()

```
Scene2d LinePlot::render_to_scene (
    std::function< AffineSpace::Point2d(const AffineSpace::Point2d &P)> transform,
    const double x_min,
    const double x_max,
    const double y_min,
    const double y_max ) const [override], [virtual]
```

Implements [CPGF::Plot2d::GraphicObject](#).

7.26.1.3 x_max()

```
double LinePlot::x_max ( ) const [override], [virtual]
```

Implements [CPGF::Plot2d::GraphicObject](#).

7.26.1.4 x_min()

```
double LinePlot::x_min ( ) const [override], [virtual]
```

Implements [CPGF::Plot2d::GraphicObject](#).

7.26.1.5 y_max()

```
double LinePlot::y_max ( ) const [override], [virtual]
```

Implements [CPGF::Plot2d::GraphicObject](#).

7.26.1.6 y_min()

```
double LinePlot::y_min ( ) const [override], [virtual]
```

Implements [CPGF::Plot2d::GraphicObject](#).

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

- CPGF/Plot2d/LinePlot.hpp
- CPGF/Plot2d/LinePlot.cpp

7.27 CPGF::LineWidth Class Reference

This class provides a handful of useful predefined constants to express line width.

```
#include <PGFConf.hpp>
```

Static Public Attributes

- static const double **ULTRA_THIN** = 0.025
- static const double **VERY_THIN** = 0.05
- static const double **THIN** = 0.1
- static const double **SEMITHICK** = 0.2
- static const double **THICK** = 0.4
- static const double **VERY_THICK** = 0.8
- static const double **ULTRA_THICK** = 1.2

7.27.1 Detailed Description

This class provides a handful of useful predefined constants to express line width.

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

- CPGF/PGFBasics/PGFConf.hpp
- CPGF/PGFBasics/PGFConf.cpp

7.28 Math::Matrix< K > Class Template Reference

Public Member Functions

- **Matrix** (unsigned int **m**=0, unsigned int **n**=0)
- **Matrix** (unsigned int **m**, unsigned int **n**, std::vector< K > elements)
- **Matrix** (const **Matrix**< K > &B)
- **Matrix** & **operator=** (const **Matrix**< K > &B)
- **operator Matrix**< std::complex< K > > () const
- **Matrix**< K > & **operator+=** (const K &alpha)
- **Matrix**< K > & **operator-=** (const K &alpha)
- **Matrix**< K > & **operator*=** (const K &alpha)
- **Matrix**< K > & **operator/=** (const K &alpha)
- **Vector**< K > & **operator()** (const unsigned int i, const std::string &j)
- **Vector**< K > **operator()** (const unsigned int i, const std::string &j) const
- **Vector**< K > **operator()** (const std::string &i, const unsigned int j) const
- K & **operator()** (const unsigned int i, const unsigned int j)
- K **operator()** (const unsigned int i, const unsigned int j) const
- unsigned int **m** () const
Horizontal dimension.
- unsigned int **n** () const
Vertical dimension.
- bool **is_square** () const
Returns true if m==n.
- void **LU** (**Matrix**< K > &L, **Matrix**< K > &U, std::vector< unsigned int > &P) const
- K **det** () const
Calculates determinant.
- K **tr** () const
Calculates trace.
- std::vector< std::complex< double > > **eigenvalues** () const
- std::vector< **Vector**< std::complex< double > > > **eigenvectors** (std::vector< std::complex< double > > &**eigenvalues**, const bool calculate_eigenvalues=true) const
Returns eigenvectors.
- std::string **to_string** () const

Static Public Member Functions

- static **Matrix**< K > **zero** (const unsigned int **m**, const unsigned int **n**)
- static **Matrix**< K > **zero** (const unsigned int **n**)
- static **Matrix**< K > **identity** (const unsigned int **n**)

Protected Attributes

- **Vector**< K > * **vec**
- unsigned int **_m**

Friends

- [Matrix](#)< K > **operator+** (const [Matrix](#)< K > &A, const [Matrix](#)< K > &B)
- [Matrix](#)< K > **operator+** (const [Matrix](#)< K > &A, const K &alpha)
- [Matrix](#)< K > **operator+** (const K &alpha, const [Matrix](#)< K > &A)
- [Matrix](#)< K > **operator-** (const [Matrix](#)< K > &A, const [Matrix](#)< K > &B)
- [Matrix](#)< K > **operator-** (const [Matrix](#)< K > &A, const K &alpha)
- [Matrix](#)< K > **operator-** (const K &alpha, const [Matrix](#)< K > &A)
- [Matrix](#)< K > **operator*** (const [Matrix](#)< K > &A, const K &alpha)
- [Matrix](#)< K > **operator*** (const K &alpha, const [Matrix](#)< K > &A)
- [Matrix](#)< K > **operator/** (const [Matrix](#)< K > &A, const K &alpha)
- [Matrix](#)< K > **operator|** (const [Matrix](#)< K > &A, const [Matrix](#)< K > &B)
- [Vector](#)< K > **operator|** (const [Matrix](#)< K > &A, const [Vector](#)< K > &v)
- [Vector](#)< K > **operator|** (const [Vector](#)< K > &v, const [Matrix](#)< K > &A)

7.28.1 Member Function Documentation

7.28.1.1 det()

```
template<typename K >
K Matrix::det
```

Calculates determinant.

Returns

double

7.28.1.2 eigenvalues()

```
template<typename K >
std::vector< std::complex< double > > Matrix::eigenvalues
```

Returns

std::vector<std::complex<double>>

7.28.1.3 eigenvectors()

```
template<typename K >
std::vector< Vector< std::complex< double > > > Matrix::eigenvectors (
    std::vector< std::complex< double > > & eigenvalues,
    const bool calculate_eigenvalues = true ) const
```

Returns eigenvectors.

May also return eigenvalues through argument.

Parameters

<i>calculate_eigenvalues</i>	
<i>eigenvalues</i>	

Returns

`std::vector<Vector<std::complex<double>>>`

7.28.1.4 is_square()

```
template<typename K >
bool Matrix::is_square
```

Returns true if $m==n$.

Returns

true
false

7.28.1.5 m()

```
template<typename K >
unsigned int Matrix::m
```

Horizontal dimension.

Returns

unsigned int

7.28.1.6 n()

```
template<typename K >
unsigned int Matrix::n
```

Vertical dimension.

Returns

unsigned int

7.28.1.7 operator() [1/3]

```
template<typename K >
Vector< K > Matrix::operator() (
    const std::string & i,
    const unsigned int j ) const
```

Returns column j.

7.28.1.8 operator() [2/3]

```
template<typename K >
Vector< K > & Matrix::operator() (
    const unsigned int i,
    const std::string & j )
```

Returns row i.

7.28.1.9 operator() [3/3]

```
template<typename K >
Vector< K > Matrix::operator() (
    const unsigned int i,
    const std::string & j ) const
```

Returns row i.

7.28.1.10 tr()

```
template<typename K >
K Matrix::tr
```

Calculates trace.

Returns

double

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

- Math/Matrix.hpp
- Math/Matrix.cpp

7.29 Mesh< Cell > Class Template Reference

This object represents a collection of cells. The number of cells used can be changed. Methods for adaptive refinement are included.

```
#include <Mesh.hpp>
```

Classes

- class [Iterator](#)
Iterator object to loop through all the cells of the mesh.

Public Member Functions

- unsigned int [N_cells](#) () const
Returns total number of cells.
- [Mesh](#) (Cell *[first_cell](#)=nullptr, Cell *[last_cell](#)=nullptr, const [Chemistry::SolidGasReaction](#) &[QR](#)=[Chemistry::SolidGasReaction](#)(), std::function< double(const double [x](#))> [A_func](#)=[](double [x](#)){return 0;}, const double [detail_subdivide_threshold](#)=0.↔01, const double [detail_merge_threshold](#)=0.001, const double [max_length_factor](#)=1./50, const double [min_length_factor](#)=1./2000, const double [boundary_cell_max_length_factor](#)=1./1000)
Construct a new [Mesh](#) object.
- [Mesh](#) (const [Mesh](#) &mesh)
Performs a deep copy of mesh .
- [Mesh](#) ([Mesh](#) &&mesh)
Move constructor.
- [Mesh](#) & [operator=](#) (const [Mesh](#) &mesh)
Assignment operator.
- ~[Mesh](#) ()
Destroy the [Mesh](#) object.
- void [free](#) ()
Frees memory associated to the mesh by deleting all cells.
- [Mesh::Iterator](#) [begin](#) () const
Returns an iterator to the first cell.
- [Mesh::Iterator](#) [rbegin](#) () const
Returns an iterator to the last cell.
- [Mesh::Iterator](#) [end](#) () const
Returns an iterator to a virtual cell after the last one (or before the first one).
- Cell * [subdivide_at](#) (Cell *[C](#))
Subdivides cell C. Returns a pointer to the new right cell.
- Cell * [merge_cells](#) (Cell *[L](#), Cell *[R](#))
Merges two cells into one. Returns a pointer to the new cell.
- void [calculate_variable_ranges](#) ()
Calculates the maximum and minimum value of the norm of the vector of conserved quantities throughout the mesh. Then, subtracts the maximum value and the minimum value, saving the result in the member ranges.
- double [detail](#) (Cell *[L](#), Cell *[R](#)) const
This is function to estimate how different are the values stored in two neighbouring cells.
- void [optimize_mesh](#) ()
Loops through the mesh and subdivides and merges cells according to the thresholds.
- std::vector< double > [x](#) () const
Returns the average position of all cells in the mesh.
- std::vector< double > [x_partition](#) () const
Returns the partition of the X axis given by the boundaries of the cells.
- std::vector< double > [A](#) () const
Returns the area of all cells in the mesh.
- std::vector< [Math::Vector](#)< double > > [U](#) () const
Returns the vector of conserved variables of all cells in the mesh.
- void [read_from_file](#) (FILE *file)
Writes the mesh to file.
- void [write_to_file](#) (FILE *file) const
Reads the mesh from file.

Public Attributes

- Cell * **first_cell**
Pointer to the first cell of the mesh.
- Cell * **last_cell**
Pointer to the last cell of the mesh.
- [Chemistry::SolidGasReaction](#) **QR**
Chemical reaction object.
- double **detail_subdivide_threshold**
If a cell detail is bigger than this number, we divide the cell in two.
- double **detail_merge_threshold**
If the details of two neighbouring cells are lower than this number, they are merged.
- double **max_length_factor**
Determines the maximum size of a cell. A cell cannot be greater than max_length_factor times the length of the computational domain.
- double **min_length_factor**
Determines the minimum size of a cell. A cell cannot be smaller than min_length_factor times the length of the computational domain.
- double **boundary_cell_max_length_factor**
The max length factor for cells closer than 4 cells to the boundary.
- std::function< double(const double x)> **A_func**
Function used to compute the area of the newly created cells when using adaptive refinement.

Protected Attributes

- double **ranges**
The difference between the maximum and the minimum norm of the vector of preserved quantities is stored here.

7.29.1 Detailed Description

```
template<class Cell>
class Mesh< Cell >
```

This object represents a collection of cells. The number of cells used can be changed. Methods for adaptive refinement are included.

7.29.2 Constructor & Destructor Documentation

7.29.2.1 Mesh() [1/3]

```
template<class Cell >
Mesh< Cell >::Mesh (
    Cell * first_cell = nullptr,
    Cell * last_cell = nullptr,
    const Chemistry::SolidGasReaction & QR = Chemistry::SolidGasReaction(),
    std::function< double(const double x)> A_func = [] (double x){return 0;},
    const double detail_subdivide_threshold = 0.01,
    const double detail_merge_threshold = 0.001,
    const double max_length_factor = 1./50,
    const double min_length_factor = 1./2000,
    const double boundary_cell_max_length_factor = 1./1000 )
```

Construct a new [Mesh](#) object.

Parameters

<i>first_cell</i>	pointer to the first cell.
<i>last_cell</i>	pointer to the last cell.
<i>QR</i>	chemical reaction.
<i>A_func</i>	area function.
<i>detail_subdivide_threshold</i>	determines when cells are subdivided.
<i>detail_merge_threshold</i>	determines when cells are merged.
<i>max_length_factor</i>	limits the maximum length of a created cell.
<i>min_length_factor</i>	limits the minimum length of a created cell.
<i>boundary_cell_max_length_factor</i>	limits the maxium length of cells near the boundary.

7.29.2.2 Mesh() [2/3]

```
template<class Cell >
Mesh< Cell >::Mesh (
    const Mesh< Cell > & mesh )
```

Perfomrs a deep copy of *mesh* .

Parameters

<i>mesh</i>	
-------------	--

7.29.2.3 Mesh() [3/3]

```
template<class Cell >
Mesh< Cell >::Mesh (
    Mesh< Cell > && mesh )
```

Move constructor.

Parameters

<i>mesh</i>	
-------------	--

7.29.3 Member Function Documentation**7.29.3.1 A()**

```
template<class Cell >
std::vector< double > Mesh< Cell >::A
```

Returns the area of all cells in the mesh.

Returns

`std::vector<double>`

7.29.3.2 begin()

```
template<class Cell >
Mesh< Cell >::Iterator Mesh< Cell >::begin
```

Returns an iterator to the first cell.

Returns

`Mesh::Iterator`

7.29.3.3 detail()

```
template<class Cell >
double Mesh< Cell >::detail (
    Cell * L,
    Cell * R ) const
```

This is function to estimate how different are the values stored in two neighbouring cells.

Parameters

<i>L</i>	
<i>R</i>	

Returns

`double`

7.29.3.4 end()

```
template<class Cell >
Mesh< Cell >::Iterator Mesh< Cell >::end
```

Returns an iterator to a virtual cell after the last one (or before the first one).

Returns

`Mesh::Iterator`

7.29.3.5 merge_cells()

```
template<class Cell >
Cell * Mesh< Cell >::merge_cells (
    Cell * L,
    Cell * R )
```

Merges two cells into one. Returns a pointer to the new cell.

Parameters

<i>C1</i>	
<i>C2</i>	

Returns

Mesh&

7.29.3.6 N_cells()

```
template<class Cell >
unsigned int Mesh< Cell >::N_cells
```

Returns total number of cells.

Returns

unsigned int

7.29.3.7 operator=()

```
template<class Cell >
Mesh< Cell > & Mesh< Cell >::operator= (
    const Mesh< Cell > & mesh )
```

Assignment operator.

Parameters

<i>mesh</i>	
-------------	--

Returns

Mesh&

7.29.3.8 optimize_mesh()

```
template<class Cell >
void Mesh< Cell >::optimize_mesh
```

Loops through the mesh and subdivides and merges cells according to the thresholds.

Returns

Mesh&

7.29.3.9 rbegin()

```
template<class Cell >
Mesh< Cell >::Iterator Mesh< Cell >::rbegin
```

Returns an iterator to the last cell.

Returns

Mesh::Iterator

7.29.3.10 read_from_file()

```
template<class Cell >
void Mesh< Cell >::read_from_file (
    FILE * file )
```

Writes the mesh to file.

Parameters

<i>file</i>	
-------------	--

7.29.3.11 subdivide_at()

```
template<class Cell >
Cell * Mesh< Cell >::subdivide_at (
    Cell * C )
```

Subdivides cell C. Returns a pointer to the new right cell.

Parameters

<i>C</i>	
----------	--

7.29.3.12 U()

```
template<class Cell >
std::vector< Vector< double > > Mesh< Cell >::U
```

Returns the vector of conserved variables of all cells in the mesh.

Returns

`std::vector<Math::Vector<double>>`

7.29.3.13 write_to_file()

```
template<class Cell >
void Mesh< Cell >::write_to_file (
    FILE * file ) const
```

Reads the mesh from file.

Parameters

<i>file</i>	
-------------	--

7.29.3.14 x()

```
template<class Cell >
std::vector< double > Mesh< Cell >::x
```

Returns the average position of all cells in the mesh.

Returns

`std::vector<double>`

7.29.3.15 x_partition()

```
template<class Cell >
std::vector< double > Mesh< Cell >::x_partition
```

Returns the partition of the X axis given by the boundaries of the cells.

Returns

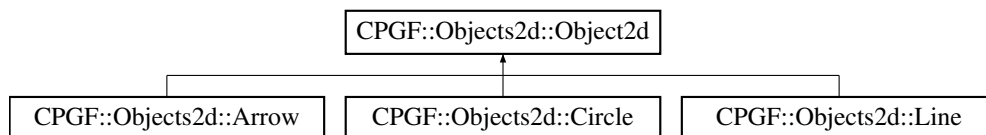
std::vector<double>

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

- [Mesh/Mesh.hpp](#)
- [Mesh/Mesh.cpp](#)

7.30 CPGF::Objects2d::Object2d Class Reference

Inheritance diagram for CPGF::Objects2d::Object2d:



Public Member Functions

- **Object2d** (const [Basics::Path2d](#) &path)
- **Object2d** (const std::vector< [Basics::Path2d](#) > &paths)
- unsigned int **size** () const
- **Object2d** & **translate** (const [AffineSpace::Vector2d](#) &v)
- **Object2d** & **rotate_with_respect_to** (const [AffineSpace::Point2d](#) &Q, const double theta)
- **Object2d** & **scale_with_respect_to** (const [AffineSpace::Point2d](#) &Q, const [AffineSpace::Vector2d](#) &s)
- **Object2d** & **operator+=** (const **Object2d** &B)
- **Object2d** & **operator+=** (const [Basics::Path2d](#) &path)
- std::string **render_to_string** () const

Public Attributes

- std::vector< [Basics::Path2d](#) > **paths**

Friends

- **Object2d operator+** (const **Object2d** &A, const **Object2d** &B)

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

- [CPGF/Objects2d/Object2d.hpp](#)
- [CPGF/Objects2d/Object2d.cpp](#)

7.31 CPGF::Basics::Path2d Class Reference

Public Member Functions

- **Path2d** ([SimpleStroke2d](#) &stroke, const [PGFConf](#) &conf=[PGFConf](#)())
- **Path2d** (const std::vector< [SimpleStroke2d](#) * > &strokes, const [PGFConf](#) &conf=[PGFConf](#)())
- **Path2d** (const [Path2d](#) &path)
- [Path2d](#) & **operator=** (const [Path2d](#) &path)
- [Path2d](#) & **translate** (const [AffineSpace::Vector2d](#) &v)
- [Path2d](#) & **rotate_with_respect_to** (const [AffineSpace::Point2d](#) &Q, const double theta)
- [Path2d](#) & **scale_with_respect_to** (const [AffineSpace::Point2d](#) &Q, const [AffineSpace::Vector2d](#) &v)
- [AffineSpace::Point2d](#) & **start** ()
- [AffineSpace::Point2d](#) **start** () const
- [AffineSpace::Point2d](#) & **end** ()
- [AffineSpace::Point2d](#) **end** () const
- double **length** () const
- double **area** () const
- [Path2d](#) & **operator+=** ([SimpleStroke2d](#) &stroke)
- [Path2d](#) & **add_stroke** ([SimpleStroke2d](#) &stroke)
- unsigned int **size** () const
- std::string **render_to_string** () const

Public Attributes

- std::vector< [SimpleStroke2d](#) * > **strokes**
- [PGFConf](#) **conf**

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

- CPGF/PGFBasics/Path2d.hpp
- CPGF/PGFBasics/Path2d.cpp

7.32 CPGF::PGFConf Class Reference

This object is used to store all information needed to draw a path.

```
#include <PGFConf.hpp>
```

Public Member Functions

- **PGFConf** (const [DrawType](#) draw_type=DrawType::DRAW, const [Color](#) &color=[Color::BLACK](#), const double [opacity](#)=1, const double [line_width](#)=LineWidth::SEMITHICK, const std::vector< double > &[dash_pattern](#)=std::vector< double >(), const double [dash_phase](#)=0, const [LineCap](#) line_cap=[LineCap::BUTT](#), const [LineJoin](#) line_join=[LineJoin::BEVEL](#))

Public Attributes

- [DrawType](#) **draw_type**
- [LineCap](#) **line_cap**
- [LineJoin](#) **line_join**
- [Color](#) **color**
- `std::vector< double >` **dash_pattern**
This double array is used to decide whether the line drawn is continuous or discontinuous (i.e. it has gaps).
- `double` **dash_phase**
- `double` **opacity**
This determines how transparent the path drawn is. It must be a real number between zero and one. One means it is fully opaque and zero represents full transparency.
- `double` **line_width**
This determines the thickness of the line drawn. For default predefined values, you can use the class [LineWidth](#).

7.32.1 Detailed Description

This object is used to store all information needed to draw a path.

7.32.2 Member Data Documentation

7.32.2.1 dash_pattern

```
std::vector<double> CPGF::PGFConf::dash_pattern
```

This double array is used to decide whether the line drawn is continuous or discontinuous (i.e. it has gaps).

Todo Explain this better.

7.32.2.2 dash_phase

```
double CPGF::PGFConf::dash_phase
```

Todo Explain this.

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

- CPGF/PGFBasics/PGFConf.hpp
- CPGF/PGFBasics/PGFConf.cpp

7.33 CPGF::AffineSpace::Point2d Class Reference

A point of a 2D affine space.

```
#include <Point2d.hpp>
```

Public Member Functions

- [Point2d](#) & [operator+=](#) (const [Vector2d](#) &v)
Traslates the point by the vector v.
- double [angle_with_respect_to](#) (const [Point2d](#) &Q)
Calculates the angle the point Q has with respect to P.
- [Point2d](#) & [rotate_with_respect_to](#) (const [Point2d](#) &Q, const double theta)
*Rotates the point (*this) an angle theta around the point Q.*
- [Point2d](#) & [rotate_to_with_respect_to](#) (const [Point2d](#) &Q, const double theta)
*Rotates the point (*this) to a fixed angle theta around the point Q.*
- [Point2d](#) & [scale_with_respect_to](#) (const [Point2d](#) &Q, const [Vector2d](#) &s)
*The position of *this with respect to the point Q is scaled according to the components of the vector s.*
- std::string [to_string](#) () const
Retruns a string representation of the point.
- [Point2d](#) ()
Returns the origin of coordinates.
- [Point2d](#) (double x, double y)
Retruns the point (x,y).

Public Attributes

- double x
The x coordinate of the point.
- double y
The y coordinate of the point.

Friends

- [Point2d](#) [operator+](#) (const [Point2d](#) &P, const [Vector2d](#) &v)
Function that adds a vector v to the point P.
- [Vector2d](#) [operator-](#) (const [Point2d](#) &P, const [Point2d](#) &Q)
Function that returns the vector that joins two points.
- bool [operator==](#) (const [Point2d](#) &P, const [Point2d](#) &Q)
Returns whether P and Q are closer than 1e-10.

7.33.1 Detailed Description

A point of a 2D affine space.

As usual, a vector may be added to a point and two points may be "substracted" to obtain a vector.

7.33.2 Constructor & Destructor Documentation

7.33.2.1 Point2d()

```
Point2d::Point2d (
    double x,
    double y )
```

Retruns the point (x,y).

Parameters

<i>x</i>	
<i>y</i>	

7.33.3 Member Function Documentation

7.33.3.1 angle_with_respect_to()

```
double Point2d::angle_with_respect_to (
    const Point2d & Q )
```

Calculates the angle the point Q has with respect to P.

Returns the angle between the horizontal line that passes through P and the line that joins P and Q.

Parameters

<i>Q</i>	a 2D point.
----------	-------------

Returns

double. A real number between 0 and 2π .

7.33.3.2 operator+=()

```
Point2d & Point2d::operator+= (
    const Vector2d & v )
```

Traslates the point by the vector v.

Equivalent to `*this = *this + v;`

Parameters

<i>v</i>	a 2D vector.
----------	--------------

Returns

[Point2d](#)&

7.33.3.3 rotate_to_with_respect_to()

```
Point2d & Point2d::rotate_to_with_respect_to (
    const Point2d & Q,
    const double theta )
```

Rotates the point (*this) to a fixed angle theta around the point Q.

Using Q as the center of rotation, the line that joins P and Q is rotated around Q until the angle between that line and the horizontal line that passes through Q is exactly theta. As a consequence, the position of P changes, although the distance PQ is preserved.

Parameters

<i>Q</i>	a 2D point.
<i>theta</i>	a double. A real number between 0 and 2π .

Returns

[Point2d](#)&

7.33.3.4 rotate_with_respect_to()

```
Point2d & Point2d::rotate_with_respect_to (
    const Point2d & Q,
    const double theta )
```

Rotates the point (*this) an angle theta around the point Q.

Using Q as the center of rotation, the line that joins P and Q is rotates around Q an angle theta. As a consequence, the position of P changes, although the distance PQ is preserved.

Parameters

<i>Q</i>	a 2D point.
<i>theta</i>	a double. A real number between 0 and 2π .

Returns

[Point2d](#)&**7.33.3.5 scale_with_respect_to()**

```
Point2d & Point2d::scale_with_respect_to (
    const Point2d & Q,
    const Vector2d & s )
```

The position of *this with respect to the point Q is scaled according to the components of the vector s.

First, we express the point P through its coordinates with respect to the point Q. Then, those coordinates are multiplied componentwise by the vector s. Finally, the new P is expressed through its coordinates with respect to the origin.

Parameters

<i>Q</i>	
<i>s</i>	

Returns

[Point2d](#)&**7.33.3.6 to_string()**

```
std::string Point2d::to_string ( ) const
```

Retruns a string representation of the point.

Returns

std::string

7.33.4 Friends And Related Function Documentation**7.33.4.1 operator+**

```
Point2d operator+ (
    const Point2d & P,
    const Vector2d & v ) [friend]
```

Function that adds a vector v to the point P.

Returns the position of the end of the vector v when its start is placed at the point P.

Parameters

P	a 2D point.
v	a 2D vector.

Returns

Point2D

7.33.4.2 operator-

```
Vector2d operator- (
    const Point2d & P,
    const Point2d & Q ) [friend]
```

Function that returns the vector that joins two points.

Returns the vector whose start is at point P and whose end is at point Q.

Parameters

P	a 2D point.
Q	a 2D point.

Returns

Vector2d

7.33.4.3 operator==

```
bool operator== (
    const Point2d & P,
    const Point2d & Q ) [friend]
```

Returns whether P and Q are closer than 1e-10.

Parameters

P	
Q	

Returns

true

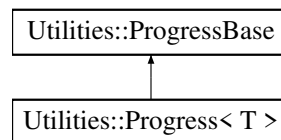
false

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

- CPGF/AffineSpace2d/Point2d.hpp
- CPGF/AffineSpace2d/Point2d.cpp

7.34 Utilities::Progress< T > Class Template Reference

Inheritance diagram for Utilities::Progress< T >:



Public Member Functions

- **Progress** (const std::string &name="", ProgressEstimation estimation=ProgressEstimation::LINEAR, const T initial=T(), const T objective=T())
- **Progress** ([Progress](#)< T > &progress)
- [Progress](#)< T > & **operator=** (const [Progress](#)< T > &progress)
- std::string [report](#) (const unsigned int level) const override
- [Progress](#)< T > & **operator=** (const T &val)
- [Progress](#)< T > & **operator+=** (const T &val)
- [Progress](#)< T > & **operator++** ()
- [Progress](#)< T > & **operator-=** (const T &val)
- [Progress](#)< T > & **operator--** ()
- [Progress](#)< T > & **operator*=** (const T &val)
- [Progress](#)< T > & **operator/=** (const T &val)
- [Progress](#)< T > & **operator%=>** (const T &val)
- **operator std::atomic< T > &** ()
- **operator T** () const

Protected Attributes

- T **initial**
- T **objective**
- std::atomic< T > **current**

Additional Inherited Members

7.34.1 Member Function Documentation

7.34.1.1 report()

```
template<class T >
std::string Progress::report (
    const unsigned int level ) const [override], [virtual]
```

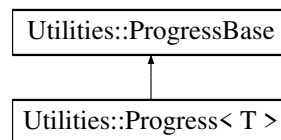
Reimplemented from [Utilities::ProgressBase](#).

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

- Utilities/Progress.hpp
- Utilities/Progress.cpp

7.35 Utilities::ProgressBase Class Reference

Inheritance diagram for Utilities::ProgressBase:



Public Member Functions

- void **start** ()
- void **pause** ()
- void **resume** ()
- void **finish** ()
- virtual std::string **report** (const unsigned int level) const
- **ProgressBase** (const std::string &name="", const ProgressEstimation estimation=ProgressEstimation::↔ LINEAR)
- **ProgressBase** (const [ProgressBase](#) &progress)
- [ProgressBase](#) & **operator=** (const [ProgressBase](#) &progress)
- [ProgressBase](#) & **operator[]** (std::string name)
- void **add_child** ([ProgressBase](#) &progress)
- void **eliminate_child** ([ProgressBase](#) &progress)
- void **update_to_terminal** (unsigned int period=250)

Public Attributes

- ProgressStatus **status**

Protected Attributes

- ProgressEstimation **estimation**
- std::string **name**
- std::chrono::system_clock::time_point **t_start**
- std::chrono::system_clock::time_point **t_paused**
- std::chrono::system_clock::time_point **t_finish**
- std::chrono::system_clock::duration **inactive_time**
- std::map< std::string, [ProgressBase](#) * > **children**
- std::mutex **M**
- std::future< void > **task**

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

- Utilities/Progress.hpp
- Utilities/Progress.cpp

7.36 Math::Rational< K > Class Template Reference

Public Member Functions

- **Rational** (const K [num](#))
- **Rational** (const K [num](#), const K [den](#))
- std::string **to_string** () const

Public Attributes

- K [num](#)
- K [den](#)

Friends

- [Rational](#)< K > & **operator+** (const [Rational](#)< K > &p, const [Rational](#)< K > &q)
- [Rational](#)< K > & **operator-** (const [Rational](#)< K > &p, const [Rational](#)< K > &q)
- [Rational](#)< K > & **operator*** (const [Rational](#)< K > &p, const [Rational](#)< K > &q)
- [Rational](#)< K > & **operator/** (const [Rational](#)< K > &p, const [Rational](#)< K > &q)
- bool **operator==** (const [Rational](#)< K > &p, const [Rational](#)< K > &q)
- bool **operator!=** (const [Rational](#)< K > &p, const [Rational](#)< K > &q)
- bool **operator<** (const [Rational](#)< K > &p, const [Rational](#)< K > &q)
- bool **operator<=** (const [Rational](#)< K > &p, const [Rational](#)< K > &q)
- bool **operator>** (const [Rational](#)< K > &p, const [Rational](#)< K > &q)
- bool **operator>=** (const [Rational](#)< K > &p, const [Rational](#)< K > &q)

7.36.1 Member Data Documentation

7.36.1.1 den

```
template<typename K >
K Math::Rational< K >::den
```

Denominator.

7.36.1.2 num

```
template<typename K >
K Math::Rational< K >::num
```

Numerator.

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

- Math/Rational.hpp
- Math/Rational.cpp

7.37 CPGF::Scene2d Class Reference

Public Member Functions

- **Scene2d** (const std::vector< [Objects2d::Object2d](#) * > &objects=std::vector< [Objects2d::Object2d](#) * >(), const std::vector< [Text](#) * > &texts=std::vector< [Text](#) * >())
- [Scene2d](#) & **add** ([Objects2d::Object2d](#) &object)
- [Scene2d](#) & **add** ([Text](#) &texts)
- [Scene2d](#) friend **operator+** (const [Scene2d](#) &S1, const [Scene2d](#) &S2)
- [Scene2d](#) & **operator+=** (const [Scene2d](#) &S2)
- [Scene2d](#) & **operator+=** ([Objects2d::Object2d](#) &Obj)
- [Scene2d](#) & **operator+=** ([Text](#) &text)
- std::string **render_to_string** () const
- void **render** (const std::string &filename, const unsigned int density=100) const

Public Attributes

- std::vector< [Objects2d::Object2d](#) * > **objects**
- std::vector< [Text](#) * > **texts**

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

- CPGF/Scene2d.hpp
- CPGF/Scene2d.cpp

7.38 CPGF::Plot2d::Shapes Class Reference

Static Public Member Functions

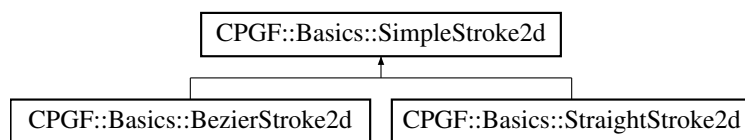
- static [Objects2d::Object2d](#) **Circle** (const [AffineSpace::Point2d](#) &pos, const [Color](#) &color, const double opacity, const double size)
- static [Objects2d::Object2d](#) **Square** (const [AffineSpace::Point2d](#) &pos, const [Color](#) &color, const double opacity, const double size)

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

- CPGF/Plot2d/DataPlot.hpp
- CPGF/Plot2d/DataPlot.cpp

7.39 CPGF::Basics::SimpleStroke2d Class Reference

Inheritance diagram for CPGF::Basics::SimpleStroke2d:



Public Member Functions

- virtual [SimpleStroke2d](#) * **clone** () const =0
- virtual [AffineSpace::Point2d](#) & **start** ()=0
- virtual [AffineSpace::Point2d](#) **start** () const =0
- virtual [AffineSpace::Point2d](#) & **end** ()=0
- virtual [AffineSpace::Point2d](#) **end** () const =0
- virtual [SimpleStroke2d](#) & **translate** (const [AffineSpace::Vector2d](#) &v)=0
- virtual [SimpleStroke2d](#) & **rotate_with_respect_to** (const [AffineSpace::Point2d](#) &Q, const double theta)=0
- virtual [SimpleStroke2d](#) & **scale_with_respect_to** (const [AffineSpace::Point2d](#) &Q, const [AffineSpace::Vector2d](#) &s)=0
- virtual double **length** () const =0
- virtual double **area** () const =0
- virtual std::vector< [AffineSpace::Point2d](#) > **operator/** (const [SimpleStroke2d](#) &B)=0
- virtual std::string **render_to_string** () const =0

7.39.1 Member Function Documentation

7.39.1.1 render_to_string()

```
virtual std::string CPGF::Basics::SimpleStroke2d::render_to_string ( ) const [pure virtual]
```

Parameters

<i>alpha</i>	a number between zero and one.
--------------	--------------------------------

Returns

[AffineSpace::Point2d](#)

Implemented in [CPGF::Basics::StraightStroke2d](#), and [CPGF::Basics::BezierStroke2d](#).

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

- CPGF/PGFBasics/Strokes2d.hpp

7.40 Simulation Class Reference

This object is used to create some initial conditions and let the domain evolve in time.

```
#include <Simulation.hpp>
```

Public Member Functions

- [Simulation](#) (const std::string &name, const [Chemistry::SolidGasReaction](#) &QR, const double a, const double b, const unsigned int N_cells_solid, const unsigned int N_cells_gas, const bool adaptive_refinement_solid, const bool adaptive_refinement_gas, const unsigned int N_tasks, const double CFL, const unsigned int N_saves, const [SolidBoundaryConditions](#) &solid_BC, const [GasBoundaryConditions](#) &gas_BC, const double x_q, std::function< double(double x)> v, std::function< double(double x)> P, std::function< double(double x)> T, std::function< double(double x)> A, std::function< void(const [GasCell](#) &A, const [GasCell](#) &B, [Math::Vector](#)< double > *F, double *S_max)> convection_solver, std::function< void([SolidMesh](#) &mesh, double &dt, const double CFL, const [SolidBoundaryConditions](#) BC, const double t)> diffusion_solver, std::function< double(std::function< double(double x)> f, const double a, const double b)> integrator=[]) (std::function< double(double x)> f, const double a, const double b) {return [Math::Integrators::Gauss_Konrad_G7_K15](#)(f, a, b);}, std::function< double(const [GasCell](#) &C, const double t)> external_forces=[]) (const [GasCell](#) &C, const double t){return 0.;}, const unsigned int adaptive_refinement_period=1, const double detail_subdivide_threshold=0.01, const double detail_merge_threshold=0.001, const double max_length_factor=1./50, const double min_length_factor=1./2000, const double boundary_cell_max_length_factor=1./1000)

Constructs a [Simulation](#) object from initial conditions. It is assumed that the simulation is of type BOTH.

- [Simulation](#) (const std::string &name, const [Chemistry::SolidGasReaction](#) &QR, const double a, const double b, const unsigned int N_cells, const bool adaptive_refinement, const unsigned int N_tasks, const double CFL, const unsigned int N_saves, const [GasBoundaryConditions](#) &BC, std::function< double(double x)> rho, std::function< double(double x)> v, std::function< double(double x)> P, std::function< double(double x)> A, std::function< void(const [GasCell](#) &A, const [GasCell](#) &B, [Math::Vector](#)< double > *F, double *S_max)> convection_solver, std::function< double(std::function< double(double x)> f, const double a, const double b)> integrator=[]) (std::function< double(double x)> f, const double a, const double b) {return [Math::Integrators::Gauss_Konrad_G7_K15](#)(f, a, b);}, std::function< double(const [GasCell](#) &C, const double t)> external_forces=[]) (const [GasCell](#) &C, const double t){return 0.;}, const unsigned int adaptive_refinement_period=1, const double detail_subdivide_threshold=0.01, const double detail_merge_threshold=0.001, const double max_length_factor=1./50, const double min_length_factor=1./2000, const double boundary_cell_max_length_factor=1./1000)

Constructs a [Simulation](#) object from initial conditions. It is assumed that the simulation is of type GAS.

- **Simulation** (const std::string &name, const **Chemistry::SolidGasReaction** &QR, const double a, const double b, const unsigned int N_cells, const bool adaptive_refinement, const unsigned int N_tasks, const double CFL, const unsigned int N_saves, const **SolidBoundaryConditions** &BC, std::function< double(double x)> T, std::function< double(double x)> A, std::function< void(**SolidMesh** &mesh, double &dt, const double CFL, const **SolidBoundaryConditions** BC, const double t)> diffusion_solver, std::function< double(std::function< double(double x)> f, const double a, const double b)> integrator=[](std::function< double(double x)> f, const double a, const double b) {return Math::Integrators::Gauss_Konrad_G7_K15(f, a, b);}, const unsigned int adaptive_refinement_period=1, const double detail_subdivide_threshold=0.01, const double detail_merge↔_threshold=0.001, const double max_length_factor=1./50, const double min_length_factor=1./2000, const double boundary_cell_max_length_factor=1./1000)

Constructs a **Simulation** object from initial conditions. It is assumed that the simulation is of type SOLID.

- **Simulation** (const std::string &file)
Construct a new **Simulation** object from a file.
- void **update** (double dt=std::numeric_limits< double >::max())
Compute a new time step.
- void **simulate_until** (const double t)
Runs the simulation until time t has been reached. The last computed time is exactly t.
- void **write_to_file** (const std::string &file) const
Writes the simulation into a file.
- double **x_q** (const double t) const
Returns the position of the combustion at a specific time t.
- double **v_q** (const double t) const
Returns the burning rate at a specific time t.
- std::function< double(const double x)> **A** (const double t) const
Returns the function A(x) for a certain time t.
- std::function< double(const double x)> **rho** (const double t) const
Returns the function rho(x) for a certain time t.
- std::function< double(const double x)> **c** (const double t) const
Returns the function c(x) for a certain time t.
- std::function< double(const double x)> **v** (const double t) const
Returns the function v(x) for a certain time t.
- std::function< double(const double x)> **T** (const double t) const
Returns the function T(x) for a certain time t.
- std::function< double(const double x)> **P** (const double t) const
Returns the function P(x) for a certain time t.
- std::function< double(const double x)> **M** (const double t) const
Returns the function M(x) for a certain time t.
- **CPGF::Plot2d::Graphic** * **mesh_plot** () const
Returns a graphic where the cell boundaries are drawn as a function of time.

Public Attributes

- **GasMesh** **instant_gas_mesh**
Gas **Mesh** at the current time.
- **SolidMesh** **instant_solid_mesh**
Solid **Mesh** at the current time.
- double **instant_v_q**
The burning rate at the current time.
- double **instant_x_q**
The position of the combustion front at the current time.
- double **instant_t**

- Current time.*
- unsigned int **refine**

The value of this variable varies from 0 to the `adaptive_refinement_period` - 1 and is used to determine when to refine the mesh. Its value is incremented once per iteration.
- std::string **name**

The name of the simulation.
- SimulationType **simulation_type**

Stores which type of simulation we have to solve.
- Utilities::FileArray< GasMesh > * **gas_mesh**

An array of gas meshes for each time instant t. It is stored in a file.
- Utilities::FileArray< SolidMesh > * **solid_mesh**

An array of solid meshes for each time instant t. It is stored in a file.
- std::vector< double > **x_q_array**

An array which contains the position of the combustion front for each time t.
- std::vector< double > **v_q_array**

An array which contains the burning rate for each time t.
- std::vector< double > **t_array**

An array of all computed times.
- GasBoundaryConditions **gas_BC**

Gas boundary conditions.
- SolidBoundaryConditions **solid_BC**

Solid boundary conditions.
- double **CFL**

Courant-Friedrichs-Lewy number.
- unsigned int **N_saves**

Determines how many times during the simulation the current meshes and values are saved to the file.
- bool **adaptive_refinement_solid**

Determines whether to use adaptive refinement for the solid.
- bool **adaptive_refinement_gas**

Determines whether to use adaptive refinement for the gas.
- unsigned int **adaptive_refinement_period**

Controls how often the mesh is refined. In particular, it represents how many iterations we wait after the mesh has been refined to refine it again.
- unsigned int **N_tasks**

How many (CPU) tasks to use in order to execute the simulation.
- std::function< double(const GasCell &C, const double t)> **external_forces**

Function that computes the external forces acting on each gas cell.
- std::function< void(const GasCell &A, const GasCell &B, Math::Vector< double > *F, double *S_max)> **convection_solver**

Function that returns the numerical flux at the interface between gas cells A and B.
- std::function< void(SolidMesh &mesh, double &dt, const double CFL, const SolidBoundaryConditions BC, const double t)> **diffusion_solver**

Function that solves the solid part.
- Utilities::Progress< double > **progress**

Variable used to store the progress of the simulation.

Protected Attributes

- double **a**
Coefficient that accompanies the term T_{n-1} in the energy equation of the last solid cell.
- double **b**
Coefficient that accompanies the term T_n in the energy equation of the last solid cell.
- double **d**
Independent term of the energy equation of the last solid cell.
- double **rho_g**
Density of the first gas cell.
- double **v_g**
Gas speed of the first gas cell.

7.40.1 Detailed Description

This object is used to create some initial conditions and let the domain evolve in time.

7.40.2 Constructor & Destructor Documentation

7.40.2.1 Simulation() [1/4]

```
Simulation::Simulation (
    const std::string & name,
    const Chemistry::SolidGasReaction & QR,
    const double a,
    const double b,
    const unsigned int N_cells_solid,
    const unsigned int N_cells_gas,
    const bool adaptive_refinement_solid,
    const bool adaptive_refinement_gas,
    const unsigned int N_tasks,
    const double CFL,
    const unsigned int N_saves,
    const SolidBoundaryConditions & solid_BC,
    const GasBoundaryConditions & gas_BC,
    const double x_q,
    std::function< double(double x)> v,
    std::function< double(double x)> P,
    std::function< double(double x)> T,
    std::function< double(double x)> A,
    std::function< void(const GasCell &A, const GasCell &B, Math::Vector< double >
    *F, double *S_max)> convection_solver,
    std::function< void(SolidMesh &mesh, double &dt, const double CFL, const SolidBoundaryConditions
    BC, const double t)> diffusion_solver,
    std::function< double(std::function< double(double x)> f, const double a, const
    double b)> integrator = [] (std::function< double(double x)> f, const double a, const double b) {return Math<
    ::Integrators::Gauss_Konrad_G7_K15(f, a, b);},
    std::function< double(const GasCell &C, const double t)> external_forces = [] (const GasCell &C,
    const unsigned int adaptive_refinement_period = 1,
```

```

const double detail_subdivide_threshold = 0.01,
const double detail_merge_threshold = 0.001,
const double max_length_factor = 1./50,
const double min_length_factor = 1./2000,
const double boundary_cell_max_length_factor = 1./1000 )

```

Constructs a [Simulation](#) object from initial conditions. It is assumed that the simulation is of type BOTH.

Parameters

<i>name</i>	the name of the simulation.
<i>QR</i>	the quchemical reaction.
<i>a</i>	the left limit of the mesh.
<i>b</i>	the right limit of the mesh.
<i>N_cells_solid</i>	the initial number of cells for the solid mesh.
<i>N_cells_gas</i>	the initial number of cells for the gas mesh.
<i>adaptive_refinement_solid</i>	whether to use adaptive refinement for the solid mesh.
<i>adaptive_refinement_gas</i>	whether to use adaptive refinement for the gas mesh.
<i>N_tasks</i>	the number of CPU tasks to use in order to compute the simulation.
<i>CFL</i>	the Courant-Friedrichs-Lewy number.
<i>N_saves</i>	how many times throughout the simulation is data stored in a file.
<i>solid_BC</i>	solid boundary conditions for the simulation.
<i>gas_BC</i>	gas boundary conditions for the simulation.
<i>x_q</i>	the initial position of the solid-gas interface.
<i>v</i>	the initial gas speed as a function of space.
<i>P</i>	the initial gas pressure as a function of space
<i>T</i>	the initial temperature as a function of space.
<i>A</i>	the conduct area as a function of space.
<i>convection_solver</i>	the solver to use in order to calculate intercell fluxes.
<i>diffusion_solver</i>	the solver to use in order to solve the solid.
<i>integrator</i>	the function to use in order to integrate the initial conditions.
<i>external_forces</i>	a function that computes the external forces acting on a gas cell for a specific moment in time.
<i>adaptive_refinement_period</i>	determines how often the mesh is refined.
<i>detail_subdivide_threshold</i>	if a cell detail is bigger than this number, we divide the cell in two.
<i>detail_merge_threshold</i>	if the details of two neighbouring cells are lower than this number, they are merged.
<i>max_length_factor</i>	a cell cannot be bigger than <code>max_length_factor * total length of the mesh</code> .
<i>min_length_factor</i>	a cell cannot be small than <code>min_length_factor * total length of the mesh</code> .
<i>boundary_cell_max_length_factor</i>	<code>max_length_factor</code> for cells near the boundary.

7.40.2.2 Simulation() [2/4]

```

Simulation::Simulation (
    const std::string & name,
    const Chemistry::SolidGasReaction & QR,
    const double a,
    const double b,

```

```

    const unsigned int N_cells,
    const bool adaptive_refinement,
    const unsigned int N_tasks,
    const double CFL,
    const unsigned int N_saves,
    const GasBoundaryConditions & BC,
    std::function< double(double x)> rho,
    std::function< double(double x)> v,
    std::function< double(double x)> P,
    std::function< double(double x)> A,
    std::function< void(const GasCell &A, const GasCell &B, Math::Vector< double >
    *F, double *S_max)> convection_solver,
    std::function< double(std::function< double(double x)> f, const double a, const
double b)> integrator = [] (std::function< double(double x)> f, const double a, const double b) {return Math←
::Integrators::Gauss_Konrad_G7_K15(f, a, b);},
    std::function< double(const GasCell &C, const double t)> external_forces = [] (const GasCell &C,
    const unsigned int adaptive_refinement_period = 1,
    const double detail_subdivide_threshold = 0.01,
    const double detail_merge_threshold = 0.001,
    const double max_length_factor = 1./50,
    const double min_length_factor = 1./2000,
    const double boundary_cell_max_length_factor = 1./1000 )

```

Constructs a [Simulation](#) object from initial conditions. It is assumed that the simulation is of type GAS.

Parameters

<i>name</i>	the name of the simulation.
<i>QR</i>	the quchemical reaction object that stores the gas constants.
<i>a</i>	the left limit of the mesh.
<i>b</i>	the right limit of the mesh.
<i>N_cells</i>	the initial number of cells.
<i>adaptive_refinement</i>	whether to use adaptive refinement of the mesh.
<i>N_tasks</i>	the number of CPU tasks to use in order to compute the simulation.
<i>CFL</i>	the Courant-Friedrichs-Lewy number.
<i>N_saves</i>	how many times throughout the simulation is data stored in a file.
<i>BC</i>	boundary conditions for the simulation.
<i>rho</i>	the initial gas density as a function of space.
<i>v</i>	the initial gas speed as a function of space.
<i>P</i>	the initial pressure as a function of space.
<i>A</i>	the conduct area as a function of space.
<i>convection_solver</i>	the solver to use in order to solve convection.
<i>integrator</i>	the function to use in order to integrate the initial conditions.
<i>external_forces</i>	a function that computes the external forces acting on a cell for a specific moment in time.
<i>adaptive_refinement_period</i>	determines how often the mesh is refined.
<i>detail_subdivide_threshold</i>	if a cell detail is bigger than this number, we divide the cell in two.
<i>detail_merge_threshold</i>	if the details of two neighbouring cells are lower than this number, they are merged.
<i>max_length_factor</i>	a cell cannot be bigger than <code>max_length_factor * total length of the mesh</code> .
<i>min_length_factor</i>	a cell cannot be small than <code>min_length_factor * total length of the mesh</code> .
<i>boundary_cell_max_length_factor</i>	<code>max_length_factor</code> for cells near the boundary.

7.40.2.3 Simulation() [3/4]

```
Simulation::Simulation (
    const std::string & name,
    const Chemistry::SolidGasReaction & QR,
    const double a,
    const double b,
    const unsigned int N_cells,
    const bool adaptive_refinement,
    const unsigned int N_tasks,
    const double CFL,
    const unsigned int N_saves,
    const SolidBoundaryConditions & BC,
    std::function< double(double x)> T,
    std::function< double(double x)> A,
    std::function< void(SolidMesh &mesh, double &dt, const double CFL, const SolidBoundaryConditions
BC, const double t)> diffusion_solver,
    std::function< double(std::function< double(double x)> f, const double a, const
double b)> integrator = [] (std::function<double(double x)> f, const double a, const double b) {return MathC
::Integrators::Gauss_Konrad_G7_K15(f, a, b);},
    const unsigned int adaptive_refinement_period = 1,
    const double detail_subdivide_threshold = 0.01,
    const double detail_merge_threshold = 0.001,
    const double max_length_factor = 1./50,
    const double min_length_factor = 1./2000,
    const double boundary_cell_max_length_factor = 1./1000 )
```

Constructs a [Simulation](#) object from initial conditions. It is assumed that the simulation is of type SOLID.

Parameters

<i>name</i>	the name of the simulation.
<i>QR</i>	the quchemical reaction.
<i>a</i>	the left limit of the mesh.
<i>b</i>	the right limit of the mesh.
<i>N_cells</i>	the initial number of cells.
<i>adaptive_refinement</i>	whether to use adaptive refinement of the mesh.
<i>N_tasks</i>	the number of CPU tasks to use in order to compute the simulation.
<i>CFL</i>	the Courant-Friedrichs-Lewy number.
<i>N_saves</i>	how many times throughout the simulation is data stored in a file.
<i>BC</i>	boundary conditions for the simulation.
<i>T</i>	the initial temperature as a function of space.
<i>A</i>	the conduct area as a function of space.
<i>diffusion_solver</i>	the solver to use in order to solve diffusion.
<i>integrator</i>	the function to use in order to integrate the initial conditions.
<i>adaptive_refinement_period</i>	determines how often the mesh is refined.
<i>detail_subdivide_threshold</i>	if a cell detail is bigger than this number, we divide the cell in two.
<i>detail_merge_threshold</i>	if the details of two neighbouring cells are lower than this number, they are merged.
<i>max_length_factor</i>	a cell cannot be bigger than max_length_factor * total length of the mesh.
<i>min_length_factor</i>	a cell cannot be small than min_length_factor * total length of the mesh.
<i>boundary_cell_max_length_factor</i>	max_length_factor for cells near the boundary.

7.40.2.4 Simulation() [4/4]

```
Simulation::Simulation (
    const std::string & file ) [explicit]
```

Construct a new [Simulation](#) object from a file.

Parameters

<i>file</i>	the name of the file where the simulation is stored.
-------------	--

7.40.3 Member Function Documentation

7.40.3.1 A()

```
std::function< double(const double x)> Simulation::A (
    const double t ) const
```

Returns the function $A(x)$ for a certain time t .

A temporal and spatial linear interpolation is used.

Parameters

<i>t</i>	
----------	--

Returns

`std::function<double(const double x)>`

7.40.3.2 c()

```
std::function< double(const double x)> Simulation::c (
    const double t ) const
```

Returns the function $c(x)$ for a certain time t .

A temporal and spatial linear interpolation is used.

Parameters

t	
-----	--

Returns

`std::function<double(const double x)>`

7.40.3.3 M()

```
std::function< double(const double x)> Simulation::M (
    const double t ) const
```

Returns the function $M(x)$ for a certain time t .

A temporal and spatial linear interpolation is used.

Parameters

t	
-----	--

Returns

`std::function<double(const double x)>`

7.40.3.4 mesh_plot()

```
Graphic * Simulation::mesh_plot ( ) const
```

Returns a graphic where the cell boundaries are drawn as a function of time.

Returns

`CPGF::Plot2d::Graphic`

7.40.3.5 P()

```
std::function< double(const double x)> Simulation::P (
    const double t ) const
```

Returns the function $P(x)$ for a certain time t .

A temporal and spatial linear interpolation is used.

Parameters

t	
-----	--

Returns

`std::function<double(const double x)>`

7.40.3.6 rho()

```
std::function< double(const double x)> Simulation::rho (  
    const double t ) const
```

Returns the function $\rho(x)$ for a certain time t .

A temporal and spatial linear interpolation is used.

Parameters

t	
-----	--

Returns

`std::function<double(const double x)>`

7.40.3.7 simulate_until()

```
void Simulation::simulate_until (  
    const double t )
```

Runs the simulation until time t has been reached. The last computed time is exactly t .

Parameters

t	
-----	--

7.40.3.8 T()

```
std::function< double(const double x)> Simulation::T (  
    const double t ) const
```

Returns the function $T(x)$ for a certain time t .

A temporal and spatial linear interpolation is used.

Parameters

t	
-----	--

Returns

`std::function<double(const double x)>`

7.40.3.9 update()

```
void Simulation::update (
    double dt = std::numeric_limits<double>::max() )
```

Compute a new time step.

Parameters

dt	the maximum time step allowed.
------	--------------------------------

7.40.3.10 v()

```
std::function< double(const double x)> Simulation::v (
    const double t ) const
```

Returns the function $v(x)$ for a certain time t .

A temporal and spatial linear interpolation is used.

Parameters

t	
-----	--

Returns

`std::function<double(const double x)>`

7.40.3.11 v_q()

```
double Simulation::v_q (
    const double t ) const
```

Returns the burning rate at a specific time t .

It does a linear interpolation using `t_array` and `v_q_array`.

Parameters

<i>t</i>	
----------	--

Returns

double

7.40.3.12 write_to_file()

```
void Simulation::write_to_file (
    const std::string & file ) const
```

Writes the simulation into a file.

Parameters

<i>file</i>	the name of the file to write the simulation to.
-------------	--

7.40.3.13 x_q()

```
double Simulation::x_q (
    const double t ) const
```

Returns the position of the combustion at a specific time *t*.

It does a linear interpolation using *t_array* and *x_q_array*.

Parameters

<i>t</i>	
----------	--

Returns

double

7.40.4 Member Data Documentation

7.40.4.1 adaptive_refinement_period

```
unsigned int Simulation::adaptive_refinement_period
```

Controls how often the mesh is refined. In particular, it represents how many iterations we wait after the mesh has been refined to refine it again.

Setting this parameter to one means the mesh is refined in every iteration.

7.40.4.2 external_forces

```
std::function<double(const GasCell& C, const double t)> Simulation::external_forces
```

Function that computes the external forces acting on each gas cell.

Returns

```
std::function<double(const Cell& C, const double t)>
```

7.40.4.3 N_saves

```
unsigned int Simulation::N_saves
```

Determines how many times during the simulation the current meshes and values are saved to the file.

The number of actual saves is `N_saves + 2`, because the first and last state are saved by default. The saves are evenly spread throughout the simulation time.

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

- Simulation.hpp
- Simulation.cpp

7.41 SolidBoundaryConditions Class Reference

Object used to store left and right boundary conditions of the solid.

```
#include <SolidSolvers.hpp>
```

Public Member Functions

- [SolidBoundaryConditions](#) (const SolidBoundaryConditionsType [left](#)=SolidBoundaryConditionsType::FIXED_TEMPERATURE, const SolidBoundaryConditionsType [right](#)=SolidBoundaryConditionsType::FIXED_TEMPERATURE, std::function< double(double T, double t)> [left_condition](#)=[](double T, double t){return 0;}, std::function< double(double T, double t)> [right_condition](#)=[](double T, double t){return 0;})

Construct a new Solid Boundary Conditions object.

Public Attributes

- SolidBoundaryConditionsType **left**
Boundary condition at the left side of the mesh.
- SolidBoundaryConditionsType **right**
Boundary condition at the right side of the mesh.
- std::function< double(double T, double t)> **left_condition**
Function of temperature and time that returns the value for the left boundary condition.
- std::function< double(double T, double t)> **right_condition**
Function of temperature and time that provides the value for the right boundary condition.

7.41.1 Detailed Description

Object used to store left and right boundary conditions of the solid.

7.41.2 Constructor & Destructor Documentation

7.41.2.1 SolidBoundaryConditions()

```
SolidBoundaryConditions::SolidBoundaryConditions (
    const SolidBoundaryConditionsType left = SolidBoundaryConditionsType::FIXED_TEMPERATURE,
    const SolidBoundaryConditionsType right = SolidBoundaryConditionsType::FIXED_TEMPERATURE,
    std::function< double(double T, double t)> left_condition = [](double T, double t){return 0;},
    std::function< double(double T, double t)> right_condition = [](double T, double t){return 0;}
)
```

Construct a new Solid Boundary Conditions object.

Parameters

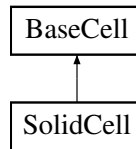
<i>left</i>	type of the left boundary condition.
<i>right</i>	type of the right boundary condition.
<i>left_condition</i>	function used to provide values for the left boundary condition.
<i>right_condition</i>	function used to provide values for the right boundary condition.

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

- Solvers/Solid/SolidSolvers.hpp
- Solvers/Solid/SolidSolvers.cpp

7.42 SolidCell Class Reference

Inheritance diagram for SolidCell:



Public Member Functions

- **SolidCell** (const [Math::Vector](#)< double > &[U](#), double [a](#), double [b](#), const double [A](#), [Chemistry::SolidGasReaction](#) *[QR](#))
Construct a new Solid Cell object.
- **SolidCell** (FILE *file, [Chemistry::SolidGasReaction](#) *[QR](#))
Construct a new Cell object from file.
- void **update** ()
Updates the value of the temperature from the value of U.
- void **read_from_file** (FILE *file)
Reads the cell values from file and calls [update\(\)](#)
- std::string **to_string** () const
Returns a string representation of the [SolidCell](#).

Public Attributes

- [SolidCell](#) * **right_neighbour**
Right neighbour of the cell.
- [SolidCell](#) * **left_neighbour**
Left neighbour of the cell.
- double **T**
Solid temperature.

7.42.1 Constructor & Destructor Documentation

7.42.1.1 SolidCell() [1/2]

```

SolidCell::SolidCell (
    const Math::Vector< double > & U,
    double a,
    double b,
    const double A,
    Chemistry::SolidGasReaction * QR )

```

Construct a new Solid Cell object.

Parameters

U	the vector of preserved variables.
a	the left limit of the cell.
b	the right limit of the cell.
A	the area of the cell.
QR	a pointer to the chemical reaction.

7.42.1.2 SolidCell() [2/2]

```
SolidCell::SolidCell (
    FILE * file,
    Chemistry::SolidGasReaction * QR )
```

Construct a new Cell object from file.

Parameters

<i>file</i>	
-------------	--

7.42.2 Member Function Documentation

7.42.2.1 read_from_file()

```
void SolidCell::read_from_file (
    FILE * file )
```

Reads the cell values from file and calls [update\(\)](#)

Parameters

<i>file</i>	
-------------	--

7.42.2.2 to_string()

```
std::string SolidCell::to_string ( ) const
```

Returns a string representation of the [SolidCell](#).

Returns

std::string

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

- Mesh/[Cell.hpp](#)
- Mesh/[Cell.cpp](#)

7.43 Chemistry::SolidGasReaction Class Reference

This object is used to store all properties of a solid reactant and a gas product. It is also used to compute the speed of the combustion front.

```
#include <Reaction.hpp>
```

Public Member Functions

- double **v_q** (const double P) const
Computes the burning rate at the given conditions.
- **SolidGasReaction** (const double rho_s=0, const double k_s=0, const double cV_s=0, const double cV_g=0, const double R_g=0, const double a=0, const double P_ref=0, const double n=0, const double delta_H=0)
Construct a new Solid Gas Reaction object.
- **SolidGasReaction** (FILE *file)
Construct a new Solid Gas Reaction object reading it from a file.
- void **read_from_file** (FILE *file)
Reads the object from file.
- void **write_to_file** (FILE *file) const
Writes the object to file.
- std::string **to_string** () const
Returns a string representation of the object.

Public Attributes

- double **rho_s**
Solid density [kg/m³].
- double **k_s**
Thermal conductivity of solid [W/(m·K)].
- double **cV_s**
Specific heat capacity at constant volume of solid [J/(kg·K)].
- double **alpha**
Thermal diffusivity of solid [m²/s].
- double **cV_g**
Specific heat capacity at constant volume of gas [J/(kg·K)].
- double **R_g**
Gas constant of gas [J/(kg·K)].
- double **gamma**
Adiabatic compression coefficient of gas [adimensional].
- double **a**
The prepotential factor of Vieille's law [m/s].
- double **P_ref**
The reference pressure for Vieille's law [Pa].
- double **n**
The exponent of Vieille's law [adimensional].
- double **delta_H**
The enthalpy increase of the chemical reaction [J/kg].

7.43.1 Detailed Description

This object is used to store all properties of a solid reactant and a gas product. It is also used to compute the speed of the combustion front.

7.43.2 Constructor & Destructor Documentation

7.43.2.1 SolidGasReaction() [1/2]

```
SolidGasReaction::SolidGasReaction (
    const double rho_s = 0,
    const double k_s = 0,
    const double cV_s = 0,
    const double cV_g = 0,
    const double R_g = 0,
    const double a = 0,
    const double P_ref = 0,
    const double n = 0,
    const double delta_H = 0 )
```

Construct a new Solid Gas Reaction object.

Parameters

<i>rho_s</i>	the solid density [kg/m ³].
<i>k_s</i>	the solid thermal conductivity [W/(m·K)].
<i>cV_s</i>	the solid heat capacity at constant volume [J/(kg·K)].
<i>cV_g</i>	the gas heat capacity at constant volume [J/(kg·K)].
<i>R_g</i>	the gas constant [J/(kg·K)].
<i>a</i>	the prepotential factor of Vieille's law [m/s].
<i>P_ref</i>	the reference pressure for Vieille's law [Pa].
<i>n</i>	the exponent of Vieille's law [adimensional].
<i>delta_H</i>	the enthalpy increase of the chemical reaction [J/kg].

7.43.2.2 SolidGasReaction() [2/2]

```
SolidGasReaction::SolidGasReaction (
    FILE * file ) [explicit]
```

Construct a new Solid Gas Reaction object reading it from a file.

Parameters

<i>file</i>	
-------------	--

7.43.3 Member Function Documentation

7.43.3.1 read_from_file()

```
void SolidGasReaction::read_from_file (
    FILE * file )
```

Reads the object from file.

Parameters

<i>file</i>	
-------------	--

7.43.3.2 to_string()

```
std::string SolidGasReaction::to_string ( ) const
```

Returns a string representation of the object.

Returns

std::string

7.43.3.3 v_q()

```
double SolidGasReaction::v_q (
    const double P ) const
```

Computes the burning rate at the given conditions.

Parameters

<i>P</i>	the pressure of the gas.
----------	--------------------------

Returns

double

7.43.3.4 write_to_file()

```
void SolidGasReaction::write_to_file (
    FILE * file ) const
```

Writes the object to file.

Parameters

<i>file</i>	
-------------	--

7.43.4 Member Data Documentation

7.43.4.1 gamma

```
double Chemistry::SolidGasReaction::gamma
```

Adiabatic compression coefficient of gas [adimensional].

Returns

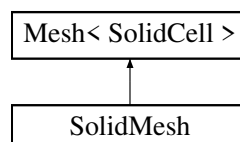
double

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

- Chemistry/[Reaction.hpp](#)
- Chemistry/[Reaction.cpp](#)

7.44 SolidMesh Class Reference

Inheritance diagram for SolidMesh:



Public Member Functions

- [SolidMesh](#) ([SolidCell](#) **first_cell*=nullptr, [SolidCell](#) **last_cell*=nullptr, const [Chemistry::SolidGasReaction](#) &*QR*=[Chemistry::SolidGasReaction](#)(), std::function< double(const double *x*)> *A_func*=[](double *x*){return 0;}, const double *detail_subdivide_threshold*=0.01, const double *detail_merge_threshold*=0.0005, const double *max_length_factor*=1./20, const double *min_length_factor*=1./2000, const double *boundary_cell_max_length_factor*=1./1000) const
Construct a new Solid *Mesh* object.
- std::vector< double > [T](#) () const
Returns the temperature of all cells in the mesh.

Additional Inherited Members

7.44.1 Constructor & Destructor Documentation

7.44.1.1 SolidMesh()

```
SolidMesh::SolidMesh (
    SolidCell * first_cell = nullptr,
    SolidCell * last_cell = nullptr,
    const Chemistry::SolidGasReaction & QR = Chemistry::SolidGasReaction(),
    std::function< double(const double x)> A_func = [] (double x){return 0;},
    const double detail_subdivide_threshold = 0.01,
    const double detail_merge_threshold = 0.0005,
    const double max_length_factor = 1./20,
    const double min_length_factor = 1./2000,
    const double boundary_cell_max_length_factor = 1./1000 )
```

Construct a new Solid [Mesh](#) object.

Parameters

<i>first_cell</i>	pointer to the first cell.
<i>last_cell</i>	pointer to the last cell.
<i>QR</i>	chemical reaction.
<i>A_func</i>	area function.
<i>detail_subdivide_threshold</i>	determines when cells are subdivided.
<i>detail_merge_threshold</i>	determines when cells are merged.
<i>max_length_factor</i>	limits the maximum length of a created cell.
<i>min_length_factor</i>	limits the minimum length of a created cell.
<i>boundary_cell_max_length_factor</i>	limits the maximum length of cells near the boundary.

7.44.2 Member Function Documentation

7.44.2.1 T()

```
std::vector< double > SolidMesh::T ( ) const
```

Returns the temperature of all cells in the mesh.

Returns

`std::vector<double>`

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

- [Mesh/Mesh.hpp](#)
- [Mesh/Mesh.cpp](#)

7.45 Solvers::Rocket::SteadySolver Class Reference

This object can be used to compute chamber and exit values for a specific rocket geometry.

```
#include <RocketSolver.hpp>
```

Public Member Functions

- **SteadySolver** (const double **T_c**, const double **M_catm**, const double **n**, const double **gamma**, const double **R**, const double **A_c**)
Construct a new Steady Solver object.
- void **solve_for_exit_area** (const double **A_e**)
Determines all chamber and exit values.
- void **calculate_optimum_parameters** ()
Calculates the chamber and exit values that provide the maximum thrust, that is, when the exit Mach number is set to one.
- std::string **to_string** () const
Returns a string representation of the object.

Public Attributes

- double **A_c**
Chamber area.
- double **A_e**
Exit area.
- double **M_catm**
Chamber Mach number when chamber pressure is the atmospheric pressure.
- double **gamma**
Gas adiabatic expansion coefficient.
- double **R**
Gas constant.
- double **n**
Vieille's power law exponent.
- double **T_c**
Chamber temperature. Also, the burning temperature of the propellant.
- double **v_c**
Chamber gas speed.
- double **M_c**
Chamber Mach number.
- double **P_c**
Chamber pressure.
- double **rho_c**
Chamber gas density.
- double **v_e**
Exit speed.
- double **M_e**
Exit Mach number.
- double **P_e**
Exit pressure.

- double **T_e**
Exit temperature.
- double **rho_e**
Exit density.
- double **m_dot**
Mass flux.
- double **thrust**
Thrust generated by the rocket.

Protected Attributes

- double **G1**
 $(\gamma - 1) / 2$
- double **G2**
 $\gamma / (\gamma - 1)$
- double **G3**
 $(\gamma + 1) / 2$
- double **G4**
 $(\gamma + 1) / (2 * (\gamma - 1))$

Static Protected Attributes

- constexpr static double **P_atm** = 101325
The atmospheric pressure.

7.45.1 Detailed Description

This object can be used to compute chamber and exit values for a specific rocket geometry.

7.45.2 Constructor & Destructor Documentation

7.45.2.1 SteadySolver()

```
SteadySolver::SteadySolver (
    const double T_c,
    const double M_catm,
    const double n,
    const double gamma,
    const double R,
    const double A_c )
```

Construct a new Steady Solver object.

Parameters

T_c	chamber temperature, also known as propellant burning temperature.
M_{catm}	chamber Mach number when the chamber pressure is the atmospheric pressure.
n	Vieille's power law exponent.
γ	gas adiabatic expansion coefficient.
R	gas constant.
A_c	chamber area.

7.45.3 Member Function Documentation

7.45.3.1 solve_for_exit_area()

```
void SteadySolver::solve_for_exit_area (
    const double A_e )
```

Determines all chamber and exit values.

Parameters

$A_{\leftrightarrow e}$	exit area.
-------------------------	------------

7.45.3.2 to_string()

```
std::string SteadySolver::to_string ( ) const
```

Returns a string representation of the object.

Returns

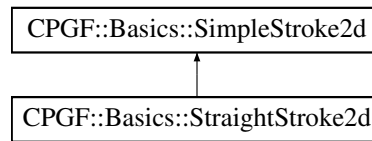
std::string

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

- Solvers/Rocket/RocketSolver.hpp
- Solvers/Rocket/RocketSolver.cpp

7.46 CPGF::Basics::StraightStroke2d Class Reference

Inheritance diagram for CPGF::Basics::StraightStroke2d:



Public Member Functions

- **StraightStroke2d** (const std::vector< [AffineSpace::Point2d](#) > points)
- [StraightStroke2d](#) * **clone** () const override
- [StraightStroke2d](#) & **operator+=** (const [AffineSpace::Point2d](#) &P)
- [StraightStroke2d](#) & **add_point** (const [AffineSpace::Point2d](#) &P)
- unsigned int **size** () const
- [AffineSpace::Point2d](#) & **start** () override
- [AffineSpace::Point2d](#) **start** () const override
- [AffineSpace::Point2d](#) & **end** () override
- [AffineSpace::Point2d](#) **end** () const override
- [StraightStroke2d](#) & **translate** (const [AffineSpace::Vector2d](#) &v) override
- [StraightStroke2d](#) & **rotate_with_respect_to** (const [AffineSpace::Point2d](#) &Q, const double theta) override
- [StraightStroke2d](#) & **scale_with_respect_to** (const [AffineSpace::Point2d](#) &Q, const [AffineSpace::Vector2d](#) &s) override
- double **length** () const override
- double **area** () const override
- std::vector< [AffineSpace::Point2d](#) > **operator/** (const [SimpleStroke2d](#) &B) override
- std::vector< [AffineSpace::Point2d](#) > **operator/** (const [StraightStroke2d](#) &B)
- std::string **render_to_string** () const override

Public Attributes

- std::vector< [AffineSpace::Point2d](#) > **points**

7.46.1 Member Function Documentation

7.46.1.1 area()

```
double StraightStroke2d::area ( ) const [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.46.1.2 clone()

```
StraightStroke2d * StraightStroke2d::clone ( ) const [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.46.1.3 end() [1/2]

```
Point2d StraightStroke2d::end ( ) const [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.46.1.4 end() [2/2]

```
Point2d & StraightStroke2d::end ( ) [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.46.1.5 length()

```
double StraightStroke2d::length ( ) const [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.46.1.6 operator/()

```
std::vector< Point2d > StraightStroke2d::operator/ (
    const SimpleStroke2d & B ) [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.46.1.7 render_to_string()

```
std::string StraightStroke2d::render_to_string ( ) const [override], [virtual]
```

Parameters

<i>alpha</i>	a number between zero and one.
--------------	--------------------------------

Returns

[AffineSpace::Point2d](#)

Implements [CPGF::Basics::SimpleStroke2d](#).

7.46.1.8 rotate_with_respect_to()

```
StraightStroke2d & StraightStroke2d::rotate_with_respect_to (
    const AffineSpace::Point2d & Q,
    const double theta ) [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.46.1.9 scale_with_respect_to()

```
StraightStroke2d & StraightStroke2d::scale_with_respect_to (
    const AffineSpace::Point2d & Q,
    const AffineSpace::Vector2d & s ) [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.46.1.10 start() [1/2]

```
Point2d StraightStroke2d::start ( ) const [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.46.1.11 start() [2/2]

```
Point2d & StraightStroke2d::start ( ) [override], [virtual]
```

Implements [CPGF::Basics::SimpleStroke2d](#).

7.46.1.12 translate()

```

StraightStroke2d & StraightStroke2d::translate (
    const AffineSpace::Vector2d & v ) [override], [virtual]

```

Implements [CPGF::Basics::SimpleStroke2d](#).

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

- CPGF/PGFBasics/Strokes2d.hpp
- CPGF/PGFBasics/Strokes2d.cpp

7.47 CPGF::Text Class Reference

```
#include <Text.hpp>
```

Public Member Functions

- **Text** (const [AffineSpace::Point2d](#) &pos=[AffineSpace::Point2d](#)(0, 0), const std::string &text="", const [Color](#) &color=[Color::BLACK](#), const [TextAlignment](#) text_alignment=[TextAlignment::CENTER](#), const double rot=0)
- std::string **render_to_string** () const

Public Attributes

- std::string **text**
The text that is going to be printed.
- [AffineSpace::Point2d](#) **pos**
The text position.
- double **rot**
The text rotation in degrees.
- [TextAlignment](#) **text_alignment**
The text alignment form.
- [Color](#) **color**
The text color.

7.47.1 Detailed Description

Warning

[Text](#) cannot be scaled or rotated, only moved.

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

- CPGF/Text/Text.hpp
- CPGF/Text/Text.cpp

7.48 Math::Vector< K > Class Template Reference

Generic [Vector](#) over the field K with any number of components.

```
#include <Vector.hpp>
```

Public Member Functions

- **Vector** ()
Creates a [Vector](#) of length zero. A [Vector](#) of length zero can be safely added or subtracted to another [Vector](#) of any length.
- **Vector** (const unsigned int N)
Creates a [Vector](#) of length N .
- **Vector** (const [Vector](#)< K > & w)
Copy constructor. Creates a deep copy of w .
- **Vector** ([Vector](#)< K > && w)
Move constructor. Copies the pointer components and N .
- **Vector** (const K *[components](#), const unsigned int N)
Creates a [Vector](#) of length N whose components are copied from the array components.
- **Vector** (std::initializer_list< K > list)
Allows to create a [Vector](#) from a initializer_list. The dimension of the [Vector](#) is automatically deduced.
- **~Vector** ()
Frees the contents of components.
- **Vector**< K > & **operator=** (const K & α)
Equivalent to $v[i] = \alpha$ for all i .
- **Vector**< K > & **operator=** (const [Vector](#)< K > & w)
Assignment operator.
- **Vector**< K > & **operator=** ([Vector](#)< K > && w)
Move-assignment operator.
- **Vector**< K > **operator+** ()
Returns the vector unaltered.
- **Vector**< K > **operator-** ()
Returns minus the vector.
- **Vector**< K > & **operator+=** (const K & α)
Equivalent to $v[i] += \alpha$ for all i .
- **Vector**< K > & **operator+=** (const [Vector](#)< K > & w)
If this has length zero, than w is copied into this. If this and w have the same dimension, this is equivalent to $this->operator[] (i) += w[i]$ for all i . Otherwise, an exception is raised.
- **Vector**< K > & **operator-=** (const K & α)
Equivalent to $v[i] -= \alpha$ for all i .
- **Vector**< K > & **operator-=** (const [Vector](#)< K > & w)
If this has length zero, than $-w$ is copied into this. If this and w have the same dimension, this is equivalent to $this->operator[] (i) -= w[i]$ for all i . Otherwise, an exception is raised.
- **Vector**< K > & **operator*=** (const K & α)
*Equivalent to $v[i] *= \alpha$ for all i .*
- **Vector**< K > & **operator*=** (const [Vector](#)< K > & w)
*If this and w have the same dimension, this is equivalent to $this->operator[] (i) *= w[i]$ for all i . Otherwise, an exception is raised.*
- **Vector**< K > & **operator/=** (const K & α)
Equivalent to $v[i] /= \alpha$ for all i .
- **Vector**< K > & **operator/=** (const [Vector](#)< K > & w)

If this and w have the same dimension, this is equivalent to this- \rightarrow `operator[]`(i) $\neq w[i]$ for all i . Otherwise, an exception is raised.

- `Vector< K > operator+ () const`
Equivalent to $v[i] = +v[i]$ for all i .
- `Vector< K > operator- () const`
Equivalent to $v[i] = -v[i]$ for all i .
- `K & operator[] (const int i)`
Indexing operator that allows to use array syntax for accessing the components of a `Vector`. Checks for out of index errors. Allows for negative indexes.
- `K operator[] (const int i) const`
Version of the `operator[]` for const objects.
- `Vector< K > slice (int first, int last) const`
Slicing operator: equivalent to Python `v[first: last]`. Returns a subvector of this whose first component is `v[first]` and whose last component is `v[last - 1]`.
- `Vector< K > reverse () const`
Returns the `Vector` reversed: it starts with the last component and it finishes with the first.
- `unsigned int size () const`
Returns the length (or dimension) of the `Vector`.
- `double norm_2 () const`
Norm 2 (euclidean norm) of the `Vector`.
- `double norm_1 () const`
Norm 1 (taxi-cab norm) of the `Vector`.
- `double norm_inf () const`
Norm infinity (maximum norm) of the `Vector`.
- `double norm_p (const double p) const`
Norm p of the `Vector`.
- `std::string to_string () const`
Returns a string representation of the `Vector`. It will look similar to `(1, 0, -3, 7.3)`.
- `void read_from_file (FILE *file)`
Reads the `Vector` from a binary file. It first reads N , then deallocates and reallocates components and finally reads components from the file.
- `void write_to_file (FILE *file) const`
Writes the `Vector` to a binary file. It first writes N and then components.

Public Attributes

- `K * components`
Array where the components of the `Vector` are stored.
- `unsigned int N`
Dimension (also called length) of the `Vector`.

Friends

- `Vector< K > operator+ (const Vector< K > &v, const Vector< K > &w)`
Componentwise addition operator.
- `Vector< K > operator+ (const Vector< K > &v, const K &alpha)`
Componentwise `Vector` + scalar addition operator.
- `Vector< K > operator+ (const K &alpha, const Vector< K > &w)`
Component wise scalar + `Vector` addition operator.
- `Vector< K > operator- (const Vector< K > &v, const Vector< K > &w)`

- Componentwise subtraction operator.*

 - `Vector< K > operator-` (const `Vector< K >` &v, const K &alpha)

Componentwise `Vector` - scalar subtraction operator.

 - `Vector< K > operator-` (const K &alpha, const `Vector< K >` &w)

Componentwise scalar - `Vector` subtraction operator.

 - `Vector< K > operator*` (const `Vector< K >` &v, const `Vector< K >` &w)

Component wise multiplication operator.

 - `Vector< K > operator*` (const K &alpha, const `Vector< K >` &v)

Component wise `Vector` - scalar multiplication operator.

 - `Vector< K > operator*` (const `Vector< K >` &v, const K &alpha)

Component wise scalar - `Vector` multiplication operator.

 - `Vector< K > operator/` (const `Vector< K >` &v, const `Vector< K >` &w)

Componentwise division operator.

 - `Vector< K > operator/` (const `Vector< K >` &v, const K &alpha)

Componentwise `Vector` - scalar division operator.

 - `Vector< K > operator/` (const K &alpha, const `Vector< K >` &w)

Componentwise scalar - `Vector` division operator.

 - `bool operator` (const `Vector< K >` &v, const `Vector< K >` &w)

Returns true if $v[i] < w[i]$ for all i and false otherwise.

 - `bool operator` (const `Vector< K >` &v, const K &alpha)

Returns true if $v[i] < \alpha$ for all i and false otherwise.

 - `bool operator` (const K &alpha, const `Vector< K >` &v)

Returns true if $\alpha < v[i]$ for all i and false otherwise.

 - `bool operator` (const `Vector< K >` &v, const `Vector< K >` &w)

Returns true if $v[i] \leq w[i]$ for all i and false otherwise.

 - `bool operator` (const `Vector< K >` &v, const K &alpha)

Returns true if $v[i] \leq \alpha$ for all i and false otherwise.

 - `bool operator` (const K &alpha, const `Vector< K >` &v)

Returns true if $\alpha \leq v[i]$ for all i and false otherwise.

 - `bool operator>` (const `Vector< K >` &v, const `Vector< K >` &w)

Returns true if $v[i] > w[i]$ for all i and false otherwise.

 - `bool operator>` (const `Vector< K >` &v, const K &alpha)

Returns true if $v[i] > \alpha$ for all i and false otherwise.

 - `bool operator>` (const K &alpha, const `Vector< K >` &v)

Returns true if $\alpha > v[i]$ for all i and false otherwise.

 - `bool operator>=` (const `Vector< K >` &v, const `Vector< K >` &w)

Returns true if $v[i] \geq w[i]$ for all i and false otherwise.

 - `bool operator>=` (const `Vector< K >` &v, const K &alpha)

Returns true if $v[i] \geq \alpha$ for all i and false otherwise.

 - `bool operator>=` (const K &alpha, const `Vector< K >` &v)

Returns true if $\alpha \geq v[i]$ for all i and false otherwise.

 - `bool operator==` (const `Vector< K >` &v, const `Vector< K >` &w)

If v and w have a different number of componentes, false is returned. If v and w have the same number of componenets, then true is returned if $v[i] == w[i]$ for all i and false is returned otherwise.

 - `bool operator==` (const `Vector< K >` &v, const K &alpha)

Returns true if $v[i] == \alpha$ for all i and false otherwise.

 - `bool operator==` (const K &alpha, const `Vector< K >` &v)

Returns true if $\alpha == v[i]$ for all i and false otherwise.

 - `bool operator!=` (const `Vector< K >` &v, const `Vector< K >` &w)

Equivalent to $!(v == w)$.

 - `bool operator!=` (const `Vector< K >` &v, const K &alpha)

- Returns true if $v[i] \neq \alpha$ for all i and false otherwise.*

 - `bool operator!= (const K &alpha, const Vector< K > &v)`

Returns true if $\alpha \neq v[i]$ for all i and false otherwise.

 - `Vector< K > operator& (const Vector< K > &v, const Vector< K > &w)`

Concatenation operator.

 - `Vector< K > operator& (const Vector< K > &v, const K &w)`

Concatenation operator.

 - `Vector< K > operator& (const K &v, const Vector< K > &w)`

Concatenation operator.

 - `K operator| (const Vector< K > &v, const Vector< K > &w)`

Scalar product operator.

 - `K vector_product_2d (const Vector< K > &v, const Vector< K > &w)`

Returns the Vector product of two 2d vectors v and w .

 - `Vector< K > vector_product_3d (const Vector< K > &v, const Vector< K > &w)`

Returns the Vector product of two 3d vectors v and w .

 - `std::ostream & operator (std::ostream &os, const Vector< K > &v)`

Allows to print a Vector v to screen with `std::cout << v`.

 - `K min (const Vector< K > &v)`

Returns the minimum element of the vector.

 - `K max (const Vector< K > &v)`

Returns the maximal element of the vector.

 - `K sum (const Vector< K > &v)`

Returns the sum of all the components of the vector.

 - `K multiply (const Vector< K > &v)`

Returns the product of all the components of the vector.

 - `Vector< K > cos (const Vector< K > &v)`

Equivalent to $\cos(v[i])$ for all i .

 - `Vector< K > sin (const Vector< K > &v)`

Equivalent to $\sin(v[i])$ for all i .

 - `Vector< K > tan (const Vector< K > &v)`

Equivalent to $\tan(v[i])$ for all i .

 - `Vector< K > acos (const Vector< K > &v)`

Equivalent to $\arccos(v[i])$ for all i .

 - `Vector< K > asin (const Vector< K > &v)`

Equivalent to $\arcsin(v[i])$ for all i .

 - `Vector< K > atan (const Vector< K > &v)`

Equivalent to $\arctan(v[i])$ for all i .

 - `Vector< K > atan2 (const K v, const Vector< K > &w)`

Equivalent to $\arctan2(v, w[i])$ for all i .

 - `Vector< K > atan2 (const Vector< K > &v, const K w)`

Equivalent to $\arctan2(v[i], w)$ for all i .

 - `Vector< K > atan2 (const Vector< K > &v, const Vector< K > &w)`

Equivalent to $\arctan2(v[i], w[i])$ for all i .

 - `Vector< K > cosh (const Vector< K > &v)`

Equivalent to $\cosh(v[i])$ for all i .

 - `Vector< K > sinh (const Vector< K > &v)`

Equivalent to $\sinh(v[i])$ for all i .

 - `Vector< K > tanh (const Vector< K > &v)`

Equivalent to $\tanh(v[i])$ for all i .

 - `Vector< K > acosh (const Vector< K > &v)`

Equivalent to $\operatorname{acosh}(v[i])$ for all i .

- `Vector< K > asinh` (const `Vector< K >` &v)
Equivalent to `asinh(v[i])` for all *i*.
- `Vector< K > atanh` (const `Vector< K >` &v)
Equivalent to `atanh(v[i])` for all *i*.
- `Vector< K > exp` (const `Vector< K >` &v)
Equivalent to `exp(v[i])` for all *i*.
- `Vector< K > frexp` (const `Vector< K >` &v, `Vector< int >` *exp)
Equivalent to `frexp(v[i],exp[i])` for all *i*.
- `Vector< K > ldexp` (const `Vector< K >` &v, const int exp)
Equivalent to `ldexp(v[i],exp)` for all *i*.
- `Vector< K > ldexp` (const `Vector< K >` &v, const `Vector< int >` &exp)
Equivalent to `ldexp(v[i],exp[i])` for all *i*.
- `Vector< K > log` (const `Vector< K >` &v)
Equivalent to `log(v[i])` for all *i*.
- `Vector< K > log10` (const `Vector< K >` &v)
Equivalent to `log10(v[i])` for all *i*.
- `Vector< K > modf` (const `Vector< K >` &v, `Vector< K >` *intpart)
Equivalent to `modf(v[i],intpart[i])` for all *i*.
- `Vector< K > exp2` (const `Vector< K >` &v)
Equivalent to `exp2(v[i])` for all *i*.
- `Vector< K > expm1` (const `Vector< K >` &v)
Equivalent to `expm1(v[i])` for all *i*.
- `Vector< K > ilogb` (const `Vector< K >` &v)
Equivalent to `ilogb(v[i])` for all *i*.
- `Vector< K > log1p` (const `Vector< K >` &v)
Equivalent to `log1p(v[i])` for all *i*.
- `Vector< K > log2` (const `Vector< K >` &v)
Equivalent to `log2(v[i])` for all *i*.
- `Vector< K > logb` (const `Vector< K >` &v)
Equivalent to `logb(v[i])` for all *i*.
- `Vector< K > scalbn` (const `Vector< K >` &v, const int n)
Equivalent to `sclbn(v[i],n)` for all *i*.
- `Vector< K > scalbn` (const `Vector< K >` &v, const `Vector< int >` &n)
Equivalent to `sclbn(v[i],n[i])` for all *i*.
- `Vector< K > scalbln` (const `Vector< K >` &v, const long int n)
Equivalent to `sclbln(v[i],n[i])` for all *i*.
- `Vector< K > scalbln` (const `Vector< K >` &v, const `Vector< long int >` &n)
Equivalent to `sclbln(v[i],n[i])` for all *i*.
- `Vector< K > pow` (const K v, const `Vector< K >` &exponent)
Equivalent to `pow(v,exponent[i])` for all *i*.
- `Vector< K > pow` (const `Vector< K >` &v, const K exponent)
Equivalent to `pow(v[i],exponent)` for all *i*.
- `Vector< K > pow` (const `Vector< K >` &v, const `Vector< K >` &exponent)
Equivalent to `sclbln(v[i],exponent[i])` for all *i*.
- `Vector< K > sqrt` (const `Vector< K >` &v)
Equivalent to `sqrt(v[i])` for all *i*.
- `Vector< K > cbrt` (const `Vector< K >` &v)
Equivalent to `cbrt(v[i])` for all *i*.
- `Vector< K > hypot` (const K x, const `Vector< K >` &y)
Equivalent to `hypot(x,y[i])` for all *i*.
- `Vector< K > hypot` (const `Vector< K >` &x, const K y)

- Equivalent to `hypot(x[i],y)` for all i .*

 - `Vector< K > hypot` (const `Vector< K >` &x, const `Vector< K >` &y)
- Equivalent to `hypot(x[i],y[i])` for all i .*

 - `Vector< K > erf` (const `Vector< K >` &v)
- Equivalent to `erf(v[i])` for all i .*

 - `Vector< K > erfc` (const `Vector< K >` &v)
- Equivalent to `erfc(v[i])` for all i .*

 - `Vector< K > tgamma` (const `Vector< K >` &v)
- Equivalent to `tgamma(v[i])` for all i .*

 - `Vector< K > lgamma` (const `Vector< K >` &v)
- Equivalent to `lgamma(v[i])` for all i .*

 - `Vector< K > ceil` (const `Vector< K >` &v)
- Equivalent to `ceil(v[i])` for all i .*

 - `Vector< K > floor` (const `Vector< K >` &v)
- Equivalent to `floor(v[i])` for all i .*

 - `Vector< K > fmod` (const K numer, const `Vector< K >` &denom)
- Equivalent to `fmod(numer,denom[i])` for all i .*

 - `Vector< K > fmod` (const `Vector< K >` &numer, const K denom)
- Equivalent to `fmod(numer[i],denom)` for all i .*

 - `Vector< K > fmod` (const `Vector< K >` &numer, const `Vector< K >` &denom)
- Equivalent to `fmod(numer[i],denom[i])` for all i .*

 - `Vector< K > trunc` (const `Vector< K >` &v)
- Equivalent to `trunc(v[i])` for all i .*

 - `Vector< K > round` (const `Vector< K >` &v)
- Equivalent to `round(v[i])` for all i .*

 - `Vector< long int > lround` (const `Vector< K >` &v)
- Equivalent to `lround(v[i])` for all i .*

 - `Vector< long long int > llround` (const `Vector< K >` &v)
- Equivalent to `llround(v[i])` for all i .*

 - `Vector< K > rint` (const `Vector< K >` &v)
- Equivalent to `rint(v[i])` for all i .*

 - `Vector< long int > lrint` (const `Vector< K >` &v)
- Equivalent to `lrint(v[i])` for all i .*

 - `Vector< long long int > llrint` (const `Vector< K >` &v)
- Equivalent to `llrint(v[i])` for all i .*

 - `Vector< K > nearbyint` (const `Vector< K >` &v)
- Equivalent to `nearbyint(v[i])` for all i .*

 - `Vector< K > remainder` (const K numer, const `Vector< K >` &denom)
- Equivalent to `remainder(numer,denom[i])` for all i .*

 - `Vector< K > remainder` (const `Vector< K >` &numer, const K denom)
- Equivalent to `remainder(numer[i],denom)` for all i .*

 - `Vector< K > remainder` (const `Vector< K >` &numer, const `Vector< K >` &denom)
- Equivalent to `remainder(numer[i],denom[i])` for all i .*

 - `Vector< K > remquo` (const K numer, const `Vector< K >` &denom, `Vector< int >` *quot)
- Equivalent to `remquo(numer,denom[i],quot[i])` for all i .*

 - `Vector< K > remquo` (const `Vector< K >` &numer, const K denom, `Vector< int >` *quot)
- Equivalent to `remquo(numer[i],denom,quot[i])` for all i .*

 - `Vector< K > remquo` (const `Vector< K >` &numer, const `Vector< K >` &denom, `Vector< int >` *quot)
- Equivalent to `remquo(numer[i],denom[i],quot[i])` for all i .*

 - `Vector< K > copysign` (const `Vector< K >` &x, const K y)
- Equivalent to `copysign(x[i],y)` for all i .*

- `Vector< K > copysign` (const `Vector< K > &x`, const `Vector< K > &y`)
Equivalent to `copysign(x[i],y[i])` for all i .
- `Vector< K > nan` (const unsigned int `N`, const char *`tagp`)
Equivalent to `nan(tagp)` for all i .
- `Vector< K > nextafter` (const `Vector< K > &x`, const `Vector< K > &y`)
Equivalent to `nextafter(x[i],y[i])` for all i .
- `Vector< K > fdim` (const `K x`, const `Vector< K > &y`)
Equivalent to `fdim(x,y[i])` for all i .
- `Vector< K > fdim` (const `Vector< K > &x`, `K y`)
Equivalent to `fdim(x[i],y)` for all i .
- `Vector< K > fdim` (const `Vector< K > &x`, const `Vector< K > &y`)
Equivalent to `fdim(x[i],y[i])` for all i .
- `Vector< double > fabs` (const `Vector< K > &v`)
Equivalent to `fabs(v[i])` for all i .
- `Vector< double > abs` (const `Vector< K > &v`)
Equivalent to `abs(v[i])` for all i .
- `Vector< K > fma` (const `K x`, const `Vector< K > &y`, const `Vector< K > &z`)
Equivalent to `fma(x,y[i],z[i])` for all i .
- `Vector< K > fma` (const `Vector< K > &x`, const `K y`, const `Vector< K > &z`)
Equivalent to `fma(x[i],y,z[i])` for all i .
- `Vector< K > fma` (const `Vector< K > &x`, const `Vector< K > &y`, const `K z`)
Equivalent to `fma(x[i],y[i],z)` for all i .
- `Vector< K > fma` (const `K x`, const `K y`, const `Vector< K > &z`)
Equivalent to `fma(x,y,z[i])` for all i .
- `Vector< K > fma` (const `K x`, const `Vector< K > &y`, const `K z`)
Equivalent to `fma(x,y[i],z)` for all i .
- `Vector< K > fma` (const `Vector< K > &x`, const `K y`, const `K z`)
Equivalent to `fma(x[i],y,z)` for all i .
- `Vector< K > fma` (const `Vector< K > &x`, const `Vector< K > &y`, const `Vector< K > &z`)
Equivalent to `fma(x[i],y[i],z[i])` for all i .
- `Vector< int > fpclassify` (const `Vector< K > &v`)
Equivalent to `fpclassify(v[i])` for all i .
- `bool isfinite` (const `Vector< K > &v`)
Returns true if all components of v are finite and false otherwise.
- `bool isinf` (const `Vector< K > &v`)
Returns true if at least one components of v is infinite and false otherwise.
- `bool isnan` (const `Vector< K > &v`)
Returns true if at least one components of v is nan and false otherwise.
- `bool isnormal` (const `Vector< K > &v`)
Returns true if all components of v are normal and false otherwise.
- `Vector< K > conj` (const `Vector< K > &v`)
Equivalent to `conj(v[i])` for all i .

7.48.1 Detailed Description

```
template<typename K>
class Math::Vector< K >
```

Generic [Vector](#) over the field K with any number of components.

It is implemented through generic programming with a template typename K . The components of the [Vector](#) are stored in an array which is dynamically assigned.

It implements the following [Vector](#) componentwise operations: $+$, $-$, $*$, $/$. Boolean operators $<$, \leq , $>$, \geq , $==$, $!=$ apply the corresponding operator component by component and "and" the results together. In other words, two vectors v and w satisfy $v < w = \text{true}$ if and only if $a[i] < b[i]$ for all i . Otherwise, $a < b = \text{false}$. In addition, all previously mentioned operators allow a Vector-scalar or a scalar-Vector version, that is: $v + 5$ or $5 + v$, for example. In that case, the scalar operator is applied to all components of the [Vector](#), in other words: $v + 5$ is equivalent to $v[i] + 5$ for all i . In the same way, $v < 5 = \text{true}$ is the same as $v[i] < 5 = \text{true}$ for all i .

Moreover, a scalar product operator $|$ is provided. Note that C++ operator precedence for $|$ is low; so it is recommend to always write parenthesis that englobe the scalar product, obtaining a mixture of Dirac's and mathematicians' notation for the scalar product: *scalar product of v and w* $= (v|w)$. The object also counts with a concatenation operator $\&$: $v \& w$ places the components of v followed by the components of w in a new [Vector](#).

The index operator $[]$ can be used to access the [Vector](#) components. Bear in mind that indexes start with zero, following the convention of the C programming language.

Template Parameters

K	Any mathematical field. Normally <i>double</i> or <i>float</i> .
-----	--

7.48.2 Constructor & Destructor Documentation

7.48.2.1 [Vector\(\)](#) [1/5]

```
template<typename K >
Vector::Vector (
    const unsigned int N ) [explicit]
```

Creates a [Vector](#) of length N .

Parameters

N	Length of the Vector .
-----	--

7.48.2.2 [Vector\(\)](#) [2/5]

```
template<typename K >
```

```
Vector::Vector (
    const Vector< K > & w )
```

Copy constructor. Creates a deep copy of *w*.

Parameters

<i>w</i>	
----------	--

7.48.2.3 Vector() [3/5]

```
template<typename K >
Vector::Vector (
    Vector< K > && w )
```

Move constructor. Copies the pointer *components* and *N*.

Warning

It leaves *w* in a corrupted state.

Parameters

<i>w</i>	
----------	--

7.48.2.4 Vector() [4/5]

```
template<typename K >
Vector::Vector (
    const K * components,
    const unsigned int N )
```

Creates a [Vector](#) of length *N* whose components are copied from the array *components*.

Parameters

<i>components</i>	An array of numbers.
<i>N</i>	The length of the array.

Warning

May cause undefined behaviour if the array supplied has not been properly constructed.

7.48.2.5 `Vector()` [5/5]

```
template<typename K >
Vector::Vector (
    std::initializer_list< K > list )
```

Allows to create a [Vector](#) from a `initializer_list`. The dimension of the [Vector](#) is automatically deduced.

Parameters

<i>list</i>	
-------------	--

7.48.3 Member Function Documentation

7.48.3.1 `norm_1()`

```
template<typename K >
double Vector::norm_1
```

Norm 1 (taxi-cab norm) of the [Vector](#).

Returns

K

7.48.3.2 `norm_2()`

```
template<typename K >
double Vector::norm_2
```

Norm 2 (euclidean norm) of the [Vector](#).

Returns

K

7.48.3.3 `norm_inf()`

```
template<typename K >
double Vector::norm_inf
```

Norm infinity (maximum norm) of the [Vector](#).

Returns

K

7.48.3.4 norm_p()

```
template<typename K >
double Vector::norm_p (
    const double p ) const
```

Norm p of the [Vector](#).

Parameters

p	
-----	--

Returns

K

7.48.3.5 operator*=() [1/2]

```
template<typename K >
Vector< K > & Vector::operator*= (
    const K & alpha )
```

Equivalent to $v[i] *= \alpha$ for all i .

Parameters

α	
----------	--

Returns

Vector<K>&

7.48.3.6 operator*=() [2/2]

```
template<typename K >
Vector< K > & Vector::operator*= (
    const Vector< K > & w )
```

If *this* and *w* have the same dimension, this is equivalent to $this->operator[] (i) *= w[i]$ for all i . Otherwise, an exception is raised.

Exceptions

<i>An</i>	exception is thrown if <i>this</i> and <i>w</i> have a different number of components.
-----------	--

Parameters

<i>w</i>	
----------	--

Returns

Vector<K>&

7.48.3.7 operator+() [1/2]

```
template<typename K >
Vector< K > Vector::operator+
```

Returns the vector unaltered.

Returns

Vector<K>

7.48.3.8 operator+() [2/2]

```
template<typename K >
Vector< K > Vector::operator+
```

Equivalent to $v[i] = +v[i]$ for all i .

Returns

Vector<K>

7.48.3.9 operator+=() [1/2]

```
template<typename K >
Vector< K > & Vector::operator+= (
    const K & alpha )
```

Equivalent to $v[i] += \alpha$ for all i .

Parameters

<i>alpha</i>	
--------------	--

Returns

Vector<K>&

7.48.3.10 operator+=() [2/2]

```
template<typename K >
Vector< K > & Vector::operator+= (
    const Vector< K > & w )
```

If *this* has length zero, then *w* is copied into *this* . If *this* and *w* have the same dimension, this is equivalent to *this*->operator[](*i*) += *w*[*i*] for all *i* . Otherwise, an exception is raised.

Exceptions

An	exception is thrown if <i>this</i> and <i>w</i> have a different number of components and the dimension of <i>this</i> is non-zero.
----	---

Parameters

<i>w</i>	
----------	--

Returns

Vector<K>&

7.48.3.11 operator-() [1/2]

```
template<typename K >
Vector< K > Vector::operator-
```

Returns minus the vector.

Returns

Vector<K>

7.48.3.12 operator-() [2/2]

```
template<typename K >
Vector< K > Vector::operator-
```

Equivalent to $v[i] = -v[i]$ for all *i* .

Returns

Vector<K>

7.48.3.13 operator-=() [1/2]

```
template<typename K >
Vector< K > & Vector::operator-= (
    const K & alpha )
```

Equivalent to $v[i] -= \alpha$ for all i .

Parameters

<i>alpha</i>	
--------------	--

Returns

Vector<K>&

7.48.3.14 operator-=() [2/2]

```
template<typename K >
Vector< K > & Vector::operator-= (
    const Vector< K > & w )
```

If *this* has length zero, then $-w$ is copied into *this*. If *this* and *w* have the same dimension, this is equivalent to $this->operator[] (i) -= w[i]$ for all i . Otherwise, an exception is raised.

Exceptions

<i>An</i>	exception is thrown if <i>this</i> and <i>w</i> have a different number of components and the dimension of <i>this</i> is non-zero.
-----------	---

Parameters

<i>w</i>	
----------	--

Returns

Vector<K>&

7.48.3.15 operator/=() [1/2]

```
template<typename K >
Vector< K > & Vector::operator/= (
    const K & alpha )
```

Equivalent to $v[i] /= \alpha$ for all i .

Parameters

<i>alpha</i>	
--------------	--

Returns

Vector<K>&

7.48.3.16 operator/=() [2/2]

```
template<typename K >
Vector< K > & Vector::operator/= (
    const Vector< K > & w )
```

If *this* and *w* have the same dimension, this is equivalent to *this->operator[]*(*i*) /= *w*[*i*] for all *i* . Otherwise, an exception is raised.

Exceptions

An	exception is thrown if <i>this</i> and <i>w</i> have a different number of components.
----	--

Parameters

<i>w</i>	
----------	--

Returns

Vector<K>&

7.48.3.17 operator=() [1/3]

```
template<typename K >
Vector< K > & Vector::operator= (
    const K & alpha )
```

Equivalent to *v*[*i*] = *alpha* for all *i*.

Parameters

<i>alpha</i>	
--------------	--

Returns

Vector<K>&

7.48.3.18 operator=() [2/3]

```
template<typename K >
Vector< K > & Vector::operator= (
    const Vector< K > & w )
```

Assignment operator.

If needed, the *components* array is freed and dynamically reassigned to match the dimension of *w*. Afterwards, *N* and the components of the Vector *w* are copied to the current object.

Parameters

<i>w</i>	
----------	--

Returns

Vector<K>&

7.48.3.19 operator=() [3/3]

```
template<typename K >
Vector< K > & Vector::operator= (
    Vector< K > && w )
```

Move-assignment operator.

Copies *components* and *N* instead of creating a new array. Faster than normal assignment operator.

Warning

It leaves *w* in a corrupted state.

Parameters

<i>w</i>	
----------	--

Returns

Vector<K>&

7.48.3.20 operator[]() [1/2]

```
template<typename K >
K & Vector::operator[] (
    const int i )
```

Indexing operator that allows to use array syntax for accessing the components of a [Vector](#). Checks for out of index errors. Allows for negative indexes.

Exceptions

<i>Throws</i>	an exception if the index supplied is greater than N .
---------------	--

Parameters

i	
-----	--

Returns

K&

7.48.3.21 operator[]() [2/2]

```
template<typename K >
K Vector::operator[] (
    const int i ) const
```

Version of the operator[] for const objects.

Parameters

i	
-----	--

Returns

K

7.48.3.22 read_from_file()

```
template<typename K >
void Vector::read_from_file (
    FILE * file )
```

Reads the [Vector](#) from a binary file. It first reads N , then deallocates and reallocates *componentes* and finally reads *components* from the file.

Parameters

<i>file</i>	
-------------	--

7.48.3.23 reverse()

```
template<typename K >
Vector< K > Vector::reverse
```

Returns the [Vector](#) reversed: it starts with the last component and it finishes with the first.

Returns

Vector<K>

7.48.3.24 size()

```
template<typename K >
unsigned int Vector::size
```

Returns the length (or dimension) of the [Vector](#).

Returns

unsigned int

7.48.3.25 slice()

```
template<typename K >
Vector< K > Vector::slice (
    int first,
    int last ) const
```

Slicing operator: equivalent to Python `v[first: last]`. Returns a subvector of *this* whose first component is `v[first]` and whose last component is `v[last - 1]`.

Parameters

<i>first</i>	The first component of the slice will be <code>v[first]</code> .
<i>last</i>	The last component of the slice will be <code>v[last - 1]</code> .

Exceptions

<i>Throws</i>	an exception if <i>last</i> < <i>first</i> .
---------------	--

Returns

Vector<K>

7.48.3.26 to_string()

```
template<typename K >
std::string Vector::to_string
```

Returns a string representation of the [Vector](#). It will look similar to (1, 0, -3, 7.3).

Returns

std::string

7.48.3.27 write_to_file()

```
template<typename K >
void Vector::write_to_file (
    FILE * file ) const
```

Writes the [Vector](#) to a binary file. It first writes *N* and than *components* .

Parameters

<i>file</i>	
-------------	--

7.48.4 Friends And Related Function Documentation

7.48.4.1 abs

```
template<typename K >
Vector< double > abs (
    const Vector< K > & v ) [friend]
```

Equivalent to *abs(v[i])* for all *i* .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.2 acos

```
template<typename K >
Vector< K > acos (
    const Vector< K > & v ) [friend]
```

Equivalent to $\text{acos}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.3 acosh

```
template<typename K >
Vector< K > acosh (
    const Vector< K > & v ) [friend]
```

Equivalent to $\text{acosh}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.4 asin

```
template<typename K >
Vector< K > asin (
    const Vector< K > & v ) [friend]
```

Equivalent to $\text{asin}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.5 asinh

```
template<typename K >
Vector< K > asinh (
    const Vector< K > & v ) [friend]
```

Equivalent to $\text{asinh}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.6 atan

```
template<typename K >
Vector< K > atan (
    const Vector< K > & v ) [friend]
```

Equivalent to $\text{atan}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.7 atan2 [1/3]

```
template<typename K >
Vector< K > atan2 (
    const K v,
    const Vector< K > & w ) [friend]
```

Equivalent to $\text{atan2}(v, w[i])$ for all i .

Parameters

v	
w	

Returns

Vector<K>

7.48.4.8 atan2 [2/3]

```
template<typename K >
Vector< K > atan2 (
    const Vector< K > & v,
    const K w ) [friend]
```

Equivalent to $\text{atan2}(v[i], w)$ for all i .

Parameters

v	
w	

Returns

Vector<K>

7.48.4.9 atan2 [3/3]

```
template<typename K >
Vector< K > atan2 (
```



```
const Vector< K > & v,
const Vector< K > & w ) [friend]
```

Equivalent to $\text{atan2}(v[i], w[i])$ for all i .

Exceptions

<i>Throws</i>	an exception if v and w have different lengths.
---------------	---

Parameters

v	
w	

Returns

Vector<K>

7.48.4.10 atanh

```
template<typename K >
Vector< K > atanh (
    const Vector< K > & v ) [friend]
```

Equivalent to $\text{atanh}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.11 cbrt

```
template<typename K >
Vector< K > cbrt (
    const Vector< K > & v ) [friend]
```

Equivalent to $\text{cbrt}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.12 ceil

```
template<typename K >
Vector< K > ceil (
    const Vector< K > & v ) [friend]
```

Equivalent to *ceil*(*v*[*i*]) for all *i*.

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

7.48.4.13 conj

```
template<typename K >
Vector< K > conj (
    const Vector< K > & v ) [friend]
```

Equivalent to *conj*(*v*[*i*]) for all *i*.

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

7.48.4.14 copysign [1/2]

```
template<typename K >
Vector< K > copysign (
    const Vector< K > & x,
    const K y ) [friend]
```

Equivalent to *copysign*(*x*[*i*],*y*) for all *i*.

Parameters

x	
y	

Returns

Vector<K>

7.48.4.15 copysign [2/2]

```
template<typename K >
Vector< K > copysign (
    const Vector< K > & x,
    const Vector< K > & y ) [friend]
```

Equivalent to $\text{copysign}(x[i], y[i])$ for all i .

Exceptions

<i>Throws</i>	an exception if x and y have different lengths.
---------------	---

Parameters

x	
y	

Returns

Vector<K>

7.48.4.16 cos

```
template<typename K >
Vector< K > cos (
    const Vector< K > & v ) [friend]
```

Equivalent to $\cos(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.17 cosh

```
template<typename K >
Vector< K > cosh (
    const Vector< K > & v ) [friend]
```

Equivalent to $\cosh(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.18 erf

```
template<typename K >
Vector< K > erf (
    const Vector< K > & v ) [friend]
```

Equivalent to $\operatorname{erf}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.19 erfc

```
template<typename K >
Vector< K > erfc (
    const Vector< K > & v ) [friend]
```

Equivalent to $\operatorname{erfc}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.20 exp

```
template<typename K >
Vector< K > exp (
    const Vector< K > & v ) [friend]
```

Equivalent to $\exp(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.21 exp2

```
template<typename K >
Vector< K > exp2 (
    const Vector< K > & v ) [friend]
```

Equivalent to $\exp2(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.22 expm1

```
template<typename K >
Vector< K > expm1 (
    const Vector< K > & v ) [friend]
```

Equivalent to $\text{expm1}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.23 fabs

```
template<typename K >
Vector< double > fabs (
    const Vector< K > & v ) [friend]
```

Equivalent to $\text{fabs}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.24 fdim [1/3]

```
template<typename K >
Vector< K > fdim (
    const K x,
    const Vector< K > & y ) [friend]
```

Equivalent to $\text{fdim}(x,y[i])$ for all i .

Parameters

x	
y	

Returns

Vector<K>

7.48.4.25 fdim [2/3]

```
template<typename K >
Vector< K > fdim (
    const Vector< K > & x,
    const Vector< K > & y ) [friend]
```

Equivalent to $fdim(x[i],y[i])$ for all i .

Exceptions

<i>Throws</i>	an exception if x and y have different lengths.
---------------	---

Parameters

x	
y	

Returns

Vector<K>

7.48.4.26 fdim [3/3]

```
template<typename K >
Vector< K > fdim (
    const Vector< K > & x,
    K y ) [friend]
```

Equivalent to $fdim(x[i],y)$ for all i .

Parameters

x	
y	

Returns

Vector<K>

7.48.4.27 floor

```
template<typename K >
Vector< K > floor (
    const Vector< K > & v ) [friend]
```

Equivalent to $\text{floor}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.28 fma [1/7]

```
template<typename K >
Vector< K > fma (
    const K x,
    const K y,
    const Vector< K > & z ) [friend]
```

Equivalent to $\text{fma}(x,y,z[i])$ for all i .

Parameters

x	
y	
z	

Returns

Vector<K>

7.48.4.29 fma [2/7]

```
template<typename K >
Vector< K > fma (
    const K x,
    const Vector< K > & y,
    const K z ) [friend]
```

Equivalent to $\text{fma}(x,y[i],z)$ for all i .

Parameters

x	
y	
z	

Returns

Vector<K>

7.48.4.30 fma [3/7]

```
template<typename K >
Vector< K > fma (
    const K x,
    const Vector< K > & y,
    const Vector< K > & z ) [friend]
```

Equivalent to $fma(x,y[i],z[i])$ for all i .

Exceptions

Throws	an exception if the length of y does not equal the length of z .
---------------	--

Parameters

x	
y	
z	

Returns

Vector<K>

7.48.4.31 fma [4/7]

```
template<typename K >
Vector< K > fma (
    const Vector< K > & x,
    const K y,
    const K z ) [friend]
```

Equivalent to $fma(x[i],y,z)$ for all i .

Parameters

x	
y	
z	

Returns

Vector<K>

7.48.4.32 fma [5/7]

```
template<typename K >
Vector< K > fma (
    const Vector< K > & x,
    const K y,
    const Vector< K > & z ) [friend]
```

Equivalent to $fma(x[i],y,z[i])$ for all i .

Exceptions

Throws	an exception if the length of x does not equal the length of z .
---------------	--

Parameters

x	
y	
z	

Returns

Vector<K>

7.48.4.33 fma [6/7]

```
template<typename K >
Vector< K > fma (
    const Vector< K > & x,
    const Vector< K > & y,
    const K z ) [friend]
```

Equivalent to $fma(x[i],y[i],z)$ for all i .

Exceptions

<i>Throws</i>	an exception if the length of x does not equal the length of y .
---------------	--

Parameters

x	
y	
z	

Returns

Vector<K>

7.48.4.34 fma [7/7]

```
template<typename K >
Vector< K > fma (
    const Vector< K > & x,
    const Vector< K > & y,
    const Vector< K > & z ) [friend]
```

Equivalent to $fma(x[i],y[i],z[i])$ for all i .

Exceptions

<i>Throws</i>	and exception if the lengths of x , y and z are not equal.
---------------	--

Parameters

x	
y	
z	

Returns

Vector<K>

7.48.4.35 fmod [1/3]

```
template<typename K >
Vector< K > fmod (
    const K numer,
    const Vector< K > & denom ) [friend]
```

Equivalent to $fmod(numer,denom[i])$ for all i .

Parameters

<i>numer</i>	
<i>denom</i>	

Returns

Vector<K>

7.48.4.36 fmod [2/3]

```
template<typename K >
Vector< K > fmod (
    const Vector< K > & numer,
    const K denom ) [friend]
```

Equivalent to *fmod(numer[i],denom)* for all *i*.

Parameters

<i>numer</i>	
<i>denom</i>	

Returns

Vector<K>

7.48.4.37 fmod [3/3]

```
template<typename K >
Vector< K > fmod (
    const Vector< K > & numer,
    const Vector< K > & denom ) [friend]
```

Equivalent to *fmod(numer[i],denom[i])* for all *i*.

Exceptions

<i>Throws</i>	an expception if <i>v</i> and <i>exponent</i> have different lenghts.
---------------	---

Parameters

<i>numer</i>	
<i>denom</i>	

Returns

Vector<K>

7.48.4.38 fpclassify

```
template<typename K >
Vector< int > fpclassify (
    const Vector< K > & v ) [friend]
```

Equivalent to *fpclassify(v[i])* for all *i*.

Parameters

<i>v</i>	
----------	--

Returns

Vector<int>

7.48.4.39 frexp

```
template<typename K >
Vector< K > frexp (
    const Vector< K > & v,
    Vector< int > * exp ) [friend]
```

Equivalent to *frexp(v[i],exp[i])* for all *i*.

Parameters

<i>v</i>	
<i>exp</i>	

Returns

Vector<K>

7.48.4.40 hypot [1/3]

```
template<typename K >
Vector< K > hypot (
```

```
const K x,  
const Vector< K > & y ) [friend]
```

Equivalent to *hypot*(*x*,*y*[*i*]) for all *i* .

Parameters

x	
y	

Returns

Vector<K>

7.48.4.41 hypot [2/3]

```
template<typename K >
Vector< K > hypot (
    const Vector< K > & x,
    const K y ) [friend]
```

Equivalent to $\text{hypot}(x[i],y)$ for all i .

Parameters

x	
y	

Returns

Vector<K>

7.48.4.42 hypot [3/3]

```
template<typename K >
Vector< K > hypot (
    const Vector< K > & x,
    const Vector< K > & y ) [friend]
```

Equivalent to $\text{hypot}(x[i],y[i])$ for all i .

Exceptions

Throws	an exception if v and $exponent$ have different lengths.
---------------	--

Parameters

x	
y	

Returns

Vector<K>

7.48.4.43 ilogb

```
template<typename K >
Vector< K > ilogb (
    const Vector< K > & v ) [friend]
```

Equivalent to *ilogb(v[i])* for all *i*.

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

7.48.4.44 isfinite

```
template<typename K >
bool isfinite (
    const Vector< K > & v ) [friend]
```

Returns *true* if all components of *v* are finite and *false* otherwise.

Parameters

<i>v</i>	
----------	--

Returns

true

false

7.48.4.45 isinf

```
template<typename K >
bool isinf (
    const Vector< K > & v ) [friend]
```

Returns *true* if at least one components of *v* is infinite and *false* otherwise.

Parameters

<i>v</i>	
----------	--

Returns

true

false

7.48.4.46 isnan

```
template<typename K >
bool isnan (
    const Vector< K > & v ) [friend]
```

Returns *true* if at least one components of *v* is nan and *false* otherwise.

Parameters

<i>v</i>	
----------	--

Returns

true

false

7.48.4.47 isnormal

```
template<typename K >
bool isnormal (
    const Vector< K > & v ) [friend]
```

Returns *true* if all components of *v* are normal and *false* otherwise.

Parameters

<i>v</i>	
----------	--

Returns

true

false

7.48.4.48 Idexp [1/2]

```
template<typename K >
Vector< K > ldexp (
    const Vector< K > & v,
    const int exp ) [friend]
```

Equivalent to $ldexp(v[i], exp)$ for all i .

Parameters

v	
exp	

Returns

Vector<K>

7.48.4.49 Idexp [2/2]

```
template<typename K >
Vector< K > ldexp (
    const Vector< K > & v,
    const Vector< int > & exp ) [friend]
```

Equivalent to $ldexp(v[i], exp[i])$ for all i .

Parameters

v	
exp	

Returns

Vector<K>

7.48.4.50 lgamma

```
template<typename K >
Vector< K > lgamma (
    const Vector< K > & v ) [friend]
```

Equivalent to $lgamma(v[i])$ for all i .

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

7.48.4.51 llrint

```
template<typename K >
Vector< long long int > llrint (
    const Vector< K > & v ) [friend]
```

Equivalent to *llrint(v[i])* for all *i*.

Parameters

<i>v</i>	
----------	--

Returns

Vector<long long int>

7.48.4.52 llround

```
template<typename K >
Vector< long long int > llround (
    const Vector< K > & v ) [friend]
```

Equivalent to *llround(v[i])* for all *i*.

Parameters

<i>v</i>	
----------	--

Returns

Vector<long long int>

7.48.4.53 log

```
template<typename K >
Vector< K > log (
    const Vector< K > & v ) [friend]
```

Equivalent to $\log(v[i])$ for all i .

Exceptions

Throws	an exception if v and exp have different lengths.
---------------	---

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.54 log10

```
template<typename K >
Vector< K > log10 (
    const Vector< K > & v ) [friend]
```

Equivalent to $\log_{10}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.55 log1p

```
template<typename K >
Vector< K > log1p (
    const Vector< K > & v ) [friend]
```

Equivalent to $\log_{1p}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.56 log2

```
template<typename K >
Vector< K > log2 (
    const Vector< K > & v ) [friend]
```

Equivalent to $\log_2(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.57 logb

```
template<typename K >
Vector< K > logb (
    const Vector< K > & v ) [friend]
```

Equivalent to $\log_b(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.58 lrint

```
template<typename K >
Vector< long int > lrint (
    const Vector< K > & v ) [friend]
```

Equivalent to *lrint(v[i])* for all *i*.

Parameters

<i>v</i>	
----------	--

Returns

Vector<long int>

7.48.4.59 lround

```
template<typename K >
Vector< long int > lround (
    const Vector< K > & v ) [friend]
```

Equivalent to *lround(v[i])* for all *i*.

Parameters

<i>v</i>	
----------	--

Returns

Vector<long int>

7.48.4.60 max

```
template<typename K >
K max (
    const Vector< K > & v ) [friend]
```

Returns the maximal element of the vector.

Parameters

<i>v</i>	
----------	--

Returns

K

7.48.4.61 min

```
template<typename K >
K min (
    const Vector< K > & v ) [friend]
```

Returns the minimum element of the vector.

Parameters

<i>v</i>	
----------	--

Returns

K

7.48.4.62 modf

```
template<typename K >
Vector< K > modf (
    const Vector< K > & v,
    Vector< K > * intpart ) [friend]
```

Equivalent to *modf(v[i],intpart[i])* for all *i*.

Parameters

<i>v</i>	
<i>intpart</i>	

Returns

Vector<K>

7.48.4.63 multiply

```
template<typename K >
K multiply (
    const Vector< K > & v ) [friend]
```

Returns the product of all the components of the vector.

Parameters

<i>v</i>	
----------	--

Returns

K

7.48.4.64 nan

```
template<typename K >
Vector< K > nan (
    const unsigned int N,
    const char * tagp ) [friend]
```

Equivalent to *nan(tagp)* for all *i*.

Parameters

<i>N</i>	
<i>tagp</i>	

Returns

Vector<K>

7.48.4.65 nearbyint

```
template<typename K >
Vector< K > nearbyint (
    const Vector< K > & v ) [friend]
```

Equivalent to *nearbyint(v[i])* for all *i*.

Parameters

<i>v</i>	
----------	--

Returns

Vector<K>

7.48.4.66 nextafter

```
template<typename K >
Vector< K > nextafter (
    const Vector< K > & x,
    const Vector< K > & y ) [friend]
```

Equivalent to *nextafter*(*x*[*i*],*y*[*i*]) for all *i* .

Exceptions

<i>Throws</i>	an exception if <i>x</i> and <i>y</i> have different lengths.
---------------	---

Parameters

<i>x</i>	
<i>y</i>	

Returns

Vector<K>

7.48.4.67 operator [1/7]

```
template<typename K >
bool operator (
    const K & alpha,
    const Vector< K > & v ) [friend]
```

Returns *true* if *alpha* < *v*[*i*] for all *i* and *false* otherwise.

Parameters

<i>alpha</i>	
<i>v</i>	

Returns

true

false

7.48.4.68 operator [2/7]

```
template<typename K >
bool operator (
```

```
const K & alpha,
const Vector< K > & v ) [friend]
```

Returns *true* if *alpha* $\leq v[i]$ for all *i* and *false* otherwise.

Parameters

<i>alpha</i>	
<i>v</i>	

Returns

true

false

7.48.4.69 operator [3/7]

```
template<typename K >
bool operator (
    const Vector< K > & v,
    const K & alpha ) [friend]
```

Returns *true* if $v[i] < alpha$ for all *i* and *false* otherwise.

Parameters

<i>v</i>	
<i>alpha</i>	

Returns

true

false

7.48.4.70 operator [4/7]

```
template<typename K >
bool operator (
    const Vector< K > & v,
    const K & alpha ) [friend]
```

Returns *true* if $v[i] \leq alpha$ for all *i* and *false* otherwise.

Parameters

<i>v</i>	
<i>alpha</i>	

Returns

true
false

7.48.4.71 operator [5/7]

```
template<typename K >
bool operator (
    const Vector< K > & v,
    const Vector< K > & w ) [friend]
```

Returns true if $v[i] < w[i]$ for all i and false otherwise.

Exceptions

<i>An</i>	exception is raised if v and w have a different number of components.
-----------	---

Parameters

v	
w	

Returns

true
false

7.48.4.72 operator [6/7]

```
template<typename K >
bool operator (
    const Vector< K > & v,
    const Vector< K > & w ) [friend]
```

Returns *true* if $v[i] \leq w[i]$ for all i and *false* otherwise.

Exceptions

<i>An</i>	exception is raised if v and w have a different number of components.
-----------	---

Parameters

v	
w	

Returns

true
false

7.48.4.73 operator [7/7]

```
template<typename K >
std::ostream & operator (
    std::ostream & os,
    const Vector< K > & v ) [friend]
```

Allows to print a [Vector](#) *v* to screen with `std::cout << v`.

Parameters

<i>os</i>	
<i>v</i>	

Returns

std::ostream&

7.48.4.74 operator"!=" [1/3]

```
template<typename K >
bool operator!= (
    const K & alpha,
    const Vector< K > & v ) [friend]
```

Returns *true* if *alpha* *!=* *v[i]* for all *i* and *false* otherwise.

Parameters

<i>alpha</i>	
<i>v</i>	

Returns

true
false

7.48.4.75 operator"!= [2/3]

```
template<typename K >
bool operator!= (
    const Vector< K > & v,
    const K & alpha ) [friend]
```

Returns *true* if $v[i] \neq \alpha$ for all i and *false* otherwise.

Parameters

<i>v</i>	
<i>alpha</i>	

Returns

true
false

7.48.4.76 operator"!= [3/3]

```
template<typename K >
bool operator!= (
    const Vector< K > & v,
    const Vector< K > & w ) [friend]
```

Equivalent to $!(v == w)$.

Parameters

<i>v</i>	
<i>w</i>	

Returns

true
false

7.48.4.77 operator& [1/3]

```
template<typename K >
Vector< K > operator& (
    const K & v,
    const Vector< K > & w ) [friend]
```

Concatenation operator.

Creates a new **Vector** whose components are the scalar v followed by the components of w .

Parameters

v	
w	

Returns

Vector<K>

7.48.4.78 operator& [2/3]

```
template<typename K >
Vector< K > operator& (
    const Vector< K > & v,
    const K & w ) [friend]
```

Concatenation operator.

Creates a new [Vector](#) whose components are the components of v followed by the scalar w .

Parameters

v	
w	

Returns

Vector<K>

7.48.4.79 operator& [3/3]

```
template<typename K >
Vector< K > operator& (
    const Vector< K > & v,
    const Vector< K > & w ) [friend]
```

Concatenation operator.

Creates a new [Vector](#) whose components are the components of v followed by the components of w .

Parameters

v	
w	

Returns

Vector<K>

7.48.4.80 operator* [1/3]

```
template<typename K >
Vector< K > operator* (
    const K & alpha,
    const Vector< K > & v ) [friend]
```

Component wise Vector - scalar multiplication operator.

$v * \alpha$ is equivalent to $v[i] * \alpha$ for all i .

Parameters

<i>alpha</i>	
<i>v</i>	

Returns

Vector<K>

7.48.4.81 operator* [2/3]

```
template<typename K >
Vector< K > operator* (
    const Vector< K > & v,
    const K & alpha ) [friend]
```

Component wise scalar - Vector multiplication operator.

$\alpha * v$ is equivalent to $\alpha * v[i]$ for all i .

Parameters

<i>v</i>	
<i>alpha</i>	

Returns

Vector<K>

7.48.4.82 operator* [3/3]

```
template<typename K >
Vector< K > operator* (
    const Vector< K > & v,
    const Vector< K > & w ) [friend]
```

Component wise multiplication operator.

$v * w$ is equivalent to $v[i] * w[i]$ for all i .

Exceptions

<i>An</i>	exception is thrown if v and w have a different number of components.
-----------	---

Parameters

v	
w	

Returns

Vector<K>

7.48.4.83 operator+ [1/3]

```
template<typename K >
Vector< K > operator+ (
    const K & alpha,
    const Vector< K > & w ) [friend]
```

Component wise scalar + Vector addition operator.

$alpha + v$ is equivalent to $alpha + v[i]$ for all i .

Parameters

$alpha$	
w	

Returns

Vector<K>

7.48.4.84 operator+ [2/3]

```
template<typename K >
Vector< K > operator+ (
    const Vector< K > & v,
    const K & alpha ) [friend]
```

Componentwise **Vector** + scalar addition operator.

$v + \alpha$ is equivalent to $v[i] + \alpha$ for all i .

Parameters

<i>v</i>	
<i>alpha</i>	

Returns

Vector<K>

7.48.4.85 operator+ [3/3]

```
template<typename K >
Vector< K > operator+ (
    const Vector< K > & v,
    const Vector< K > & w ) [friend]
```

Componentwise addition operator.

$v + w$ is equivalent to $v[i] + w[i]$ for all i . If any of the vectors has length zero, then the other one is returned.

Exceptions

<i>A</i>	runtime exception is thrown if both vectors have different number of components and their lengths are both non-zero.
----------	--

Parameters

<i>v</i>	
<i>w</i>	

Returns

Vector<K>

7.48.4.86 operator- [1/3]

```
template<typename K >
Vector< K > operator- (
    const K & alpha,
    const Vector< K > & w ) [friend]
```

Componentwise scalar - **Vector** subtraction operator.

alpha - *v* is equivalent to *alpha* - *v[i]* for all *i* .

Parameters

<i>alpha</i>	
<i>w</i>	

Returns

Vector<K>

7.48.4.87 operator- [2/3]

```
template<typename K >
Vector< K > operator- (
    const Vector< K > & v,
    const K & alpha ) [friend]
```

Componentwise **Vector** - scalar subtraction operator.

v - *alpha* is equivalent to *v[i]* - *alpha* for all *i* .

Parameters

<i>v</i>	
<i>alpha</i>	

Returns

Vector<K>

7.48.4.88 operator- [3/3]

```
template<typename K >
Vector< K > operator- (
    const Vector< K > & v,
    const Vector< K > & w ) [friend]
```

Componentwise subtraction operator.

$v - w$ is equivalent to $v[i] - w[i]$ for all i . If v has null length, then $-w$ is returned. If w has length zero, then v is returned.

Exceptions

<i>A</i>	runtime exception is thrown if both vectors have different number of components and their lengths are both non-zero.
----------	--

Parameters

<i>v</i>	
<i>w</i>	

Returns

Vector<K>

7.48.4.89 operator/ [1/3]

```
template<typename K >
Vector< K > operator/ (
    const K & alpha,
    const Vector< K > & w ) [friend]
```

Componentwise scalar - Vector division operator.

α / v is equivalent to $\alpha / v[i]$ for all i .

Parameters

<i>alpha</i>	
<i>w</i>	

Returns

Vector<K>

7.48.4.90 operator/ [2/3]

```
template<typename K >
Vector< K > operator/ (
    const Vector< K > & v,
    const K & alpha ) [friend]
```

Componentwise Vector - scalar division operator.

v / α is equivalent to $v[i] / \alpha$ for all i .

Parameters

v	
α	

Returns

Vector<K>

7.48.4.91 operator/ [3/3]

```
template<typename K >
Vector< K > operator/ (
    const Vector< K > & v,
    const Vector< K > & w ) [friend]
```

Componentwise division operator.

v / w is equivalent to $v[i] / w[i]$ for all i .

Exceptions

<i>An</i>	exception is thrown if v and w have a different number of components.
-----------	---

Parameters

v	
w	

Returns

Vector<K>

7.48.4.92 operator== [1/3]

```
template<typename K >
bool operator== (
    const K & alpha,
    const Vector< K > & v ) [friend]
```

Returns *true* if $\alpha == v[i]$ for all i and *false* otherwise.

Parameters

α	
v	

Returns

true
false

7.48.4.93 operator== [2/3]

```
template<typename K >
bool operator== (
    const Vector< K > & v,
    const K & alpha ) [friend]
```

Returns *true* if $v[i] == \alpha$ for all i and *false* otherwise.

Parameters

v	
α	

Returns

true
false

7.48.4.94 operator== [3/3]

```
template<typename K >
bool operator== (
    const Vector< K > & v,
    const Vector< K > & w ) [friend]
```

If v and w have a different number of componentes, *false* is returned. If v and w have the same number of componenets, then *true* is returned if $v[i] == w[i]$ for all i and *false* is returned otherwise.

Parameters

v	
w	

Returns

true
false

7.48.4.95 operator> [1/3]

```
template<typename K >
bool operator> (
    const K & alpha,
    const Vector< K > & v ) [friend]
```

Returns *true* if *alpha* > *v[i]* for all *i* and *false* otherwise.

Parameters

<i>alpha</i>	
<i>v</i>	

Returns

true

false

7.48.4.96 operator> [2/3]

```
template<typename K >
bool operator> (
    const Vector< K > & v,
    const K & alpha ) [friend]
```

Returns *true* if *v[i]* > *alpha* for all *i* and *false* otherwise.

Parameters

<i>v</i>	
<i>alpha</i>	

Returns

true

false

7.48.4.97 operator> [3/3]

```
template<typename K >
bool operator> (
    const Vector< K > & v,
    const Vector< K > & w ) [friend]
```

Returns *true* if *v[i]* > *w[i]* for all *i* and *false* otherwise.

Exceptions

<i>An</i>	exception is raised if v and w have a different number of components.
-----------	---

Parameters

v	
w	

Returns

true

false

7.48.4.98 operator>= [1/3]

```
template<typename K >
bool operator>= (
    const K & alpha,
    const Vector< K > & v ) [friend]
```

Returns *true* if $\alpha \geq v[i]$ for all i and *false* otherwise.

Parameters

α	
v	

Returns

true

false

7.48.4.99 operator>= [2/3]

```
template<typename K >
bool operator>= (
    const Vector< K > & v,
    const K & alpha ) [friend]
```

Returns *true* if $v[i] \geq \alpha$ for all i and *false* otherwise.

Parameters

v	
α	

Returns

true
false

7.48.4.100 operator>= [3/3]

```
template<typename K >
bool operator>= (
    const Vector< K > & v,
    const Vector< K > & w ) [friend]
```

Returns *true* if $v[i] \geq w[i]$ for all i and *false* otherwise.

Exceptions

<i>An</i>	exception is raised if v and w have a different number of components.
-----------	---

Parameters

v	
w	

Returns

true
false

7.48.4.101 operator" |

```
template<typename K >
K operator| (
    const Vector< K > & v,
    const Vector< K > & w ) [friend]
```

Scalar product operator.

Performs the following operation:

$$(v|w) = \sum_{i=0}^{N-1} v_i^* w_i$$

which is the canonical scalar product of \mathbb{K}^N .

Parameters

v	
w	

Returns

K

7.48.4.102 pow [1/3]

```
template<typename K >
Vector< K > pow (
    const K v,
    const Vector< K > & exponent ) [friend]
```

Equivalent to $\text{pow}(v, \text{exponent}[i])$ for all i .

Parameters

v	
exponent	

Returns

Vector<K>

7.48.4.103 pow [2/3]

```
template<typename K >
Vector< K > pow (
    const Vector< K > & v,
    const K exponent ) [friend]
```

Equivalent to $\text{pow}(v[i], \text{exponent})$ for all i .

Parameters

v	
exponent	

Returns

Vector<K>

7.48.4.104 pow [3/3]

```
template<typename K >
Vector< K > pow (
```

```
const Vector< K > & v,
const Vector< K > & exponent ) [friend]
```

Equivalent to *sclbln*(*v*[*i*],*exponent*[*i*]) for all *i* .

Exceptions

<i>Throws</i>	an expception if <i>v</i> and <i>exponent</i> have different lenghts.
---------------	---

Parameters

<i>v</i>	
<i>exponent</i>	

Returns

Vector<K>

7.48.4.105 remainder [1/3]

```
template<typename K >
Vector< K > remainder (
    const K numer,
    const Vector< K > & denom ) [friend]
```

Equivalent to *remainder*(*numer*,*denom*[*i*]) for all *i* .

Parameters

<i>numer</i>	
<i>denom</i>	

Returns

Vector<K>

7.48.4.106 remainder [2/3]

```
template<typename K >
Vector< K > remainder (
    const Vector< K > & numer,
    const K denom ) [friend]
```

Equivalent to *remainder*(*numer*[*i*],*denom*) for all *i* .

Parameters

<i>numer</i>	
<i>denom</i>	

Returns

Vector<K>

7.48.4.107 remainder [3/3]

```
template<typename K >
Vector< K > remainder (
    const Vector< K > & numer,
    const Vector< K > & denom ) [friend]
```

Equivalent to *remainder(numer[i],denom[i])* for all *i* .

Exceptions

<i>Throws</i>	an exception if <i>numer</i> and <i>denom</i> have different lengths.
---------------	---

Parameters

<i>numer</i>	
<i>denom</i>	

Returns

Vector<K>

7.48.4.108 remquo [1/3]

```
template<typename K >
Vector< K > remquo (
    const K numer,
    const Vector< K > & denom,
    Vector< int > * quot ) [friend]
```

Equivalent to *remquo(numer,denom[i],quot[i])* for all *i* .

Parameters

<i>numer</i>	
<i>denom</i>	
<i>quot</i>	

Returns

Vector<K>

7.48.4.109 remquo [2/3]

```
template<typename K >
Vector< K > remquo (
    const Vector< K > & numer,
    const K denom,
    Vector< int > * quot ) [friend]
```

Equivalent to *remquo(numer[i],denom,quot[i])* for all *i*.

Parameters

<i>numer</i>	
<i>denom</i>	
<i>quot</i>	

Returns

Vector<K>

7.48.4.110 remquo [3/3]

```
template<typename K >
Vector< K > remquo (
    const Vector< K > & numer,
    const Vector< K > & denom,
    Vector< int > * quot ) [friend]
```

Equivalent to *remquo(numer[i],denom[i],quot[i])* for all *i*.

Exceptions

<i>Throws</i>	an exception if <i>numer</i> and <i>denom</i> have different lengths.
---------------	---

Parameters

<i>numer</i>	
<i>denom</i>	
<i>quot</i>	

Returns

Vector<K>

7.48.4.111 rint

```
template<typename K >
Vector< K > rint (
    const Vector< K > & v ) [friend]
```

Equivalent to $\text{rint}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.112 round

```
template<typename K >
Vector< K > round (
    const Vector< K > & v ) [friend]
```

Equivalent to $\text{round}(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.113 scalbln [1/2]

```
template<typename K >
Vector< K > scalbln (
    const Vector< K > & v,
    const long int n ) [friend]
```

Equivalent to $\text{scalbln}(v[i], n[i])$ for all i .

Parameters

v	
n	

Returns

Vector<K>

7.48.4.114 **scalbln** [2/2]

```
template<typename K >
Vector< K > scalbln (
    const Vector< K > & v,
    const Vector< long int > & n ) [friend]
```

Equivalent to *scalbn*(*v*[*i*],*n*[*i*]) for all *i* .

Exceptions

<i>Throws</i>	an expception if <i>v</i> and <i>n</i> have different lenghts.
---------------	--

Parameters

v	
n	

Returns

Vector<K>

7.48.4.115 **scalbn** [1/2]

```
template<typename K >
Vector< K > scalbn (
    const Vector< K > & v,
    const int n ) [friend]
```

Equivalent to *scalbn*(*v*[*i*],*n*) for all *i* .

Parameters

v	
n	

Returns

Vector<K>

7.48.4.116 scalbn [2/2]

```
template<typename K >
Vector< K > scalbn (
    const Vector< K > & v,
    const Vector< int > & n ) [friend]
```

Equivalent to $scalbn(v[i], n[i])$ for all i .

Exceptions

<i>Throws</i>	an exception if v and n have different lengths.
---------------	---

Parameters

v	
n	

Returns

Vector<K>

7.48.4.117 sin

```
template<typename K >
Vector< K > sin (
    const Vector< K > & v ) [friend]
```

Equivalent to $\sin(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.118 sinh

```
template<typename K >
Vector< K > sinh (
    const Vector< K > & v ) [friend]
```

Equivalent to $\sinh(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.119 sqrt

```
template<typename K >
Vector< K > sqrt (
    const Vector< K > & v ) [friend]
```

Equivalent to $\sqrt{v[i]}$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.120 sum

```
template<typename K >
K sum (
    const Vector< K > & v ) [friend]
```

Returns the sum of all the components of the vector.

Parameters

v	
-----	--

Returns

K

7.48.4.121 tan

```
template<typename K >
Vector< K > tan (
    const Vector< K > & v ) [friend]
```

Equivalent to $\tan(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.122 tanh

```
template<typename K >
Vector< K > tanh (
    const Vector< K > & v ) [friend]
```

Equivalent to $\tanh(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.123 tgamma

```
template<typename K >
Vector< K > tgamma (
    const Vector< K > & v ) [friend]
```

Equivalent to $tgamma(v[i])$ for all i .

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.124 trunc

```
template<typename K >
Vector< K > trunc (
    const Vector< K > & v ) [friend]
```

Equivalent to *trunc(v[i])* for all *i*.

Parameters

v	
-----	--

Returns

Vector<K>

7.48.4.125 vector_product_2d

```
template<typename K >
K vector_product_2d (
    const Vector< K > & v,
    const Vector< K > & w ) [friend]
```

Returns the Vector product of two 2d vectors v and w .

Exceptions

<i>Throws</i>	an exception if either the dimension of v or the dimension of w is different from two.
---------------	--

Parameters

v	
w	

Returns

K

7.48.4.126 vector_product_3d

```
template<typename K >
Vector< K > vector_product_3d (
    const Vector< K > & v,
    const Vector< K > & w ) [friend]
```

Returns the [Vector](#) product of two 3d vectors *v* and *w*.

Exceptions

<i>Throws</i>	an exception if either the dimension of <i>v</i> or the dimension of <i>w</i> is different from three.
---------------	--

Parameters

<i>v</i>	
<i>w</i>	

Returns

Vector<K>

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

- [Math/Vector.hpp](#)
- [Math/Vector.cpp](#)

7.49 CPGF::AffineSpace::Vector2d Class Reference

An object that represents a 2D vector.

```
#include <Vector2d.hpp>
```

Public Member Functions

- [Vector2d](#) & [operator+](#) ()
Returns itself.
- [Vector2d](#) & [operator-](#) ()
Multiplies all components by -1.
- [Vector2d](#) & [operator+=](#) (const [Vector2d](#) &w)
*Stores in *this the sum of *this and w.*

- **Vector2d** & **operator-=** (const **Vector2d** &w)
*Stores in *this the subtraction of *this and w.*
- **Vector2d** & **operator*=** (const **Vector2d** &w)
*Stores in *this the multiplication of *this and w.*
- **Vector2d** & **operator/=** (const **Vector2d** &w)
*Stores in *this the division of *this and w.*
- **Vector2d** **perp** () const
Returns a perpendicular vector.
- double **norm** () const
Returns the euclidean norm of the vector.
- std::string **to_string** () const
Retruns a string representation of the vector.
- **Vector2d** ()
Returns the null vector (0,0).
- **Vector2d** (double x)
Returns the vector (x,x).
- **Vector2d** (double x, double y)
Returns the vector (x,y).

Public Attributes

- double **x**
The x component of the vector.
- double **y**
The y component of the vector.

Friends

- **Vector2d** **operator+** (const **Vector2d** &v, const **Vector2d** &w)
Returns (v.x + w.x, v.y + w.y).
- **Vector2d** **operator-** (const **Vector2d** &v, const **Vector2d** &w)
Retruns (v.x - w.x, v.y - w.y).
- **Vector2d** **operator*** (const **Vector2d** &v, const **Vector2d** &w)
*Returns (v.x * w.x, v.y * w.y).*
- **Vector2d** **operator/** (const **Vector2d** &v, const **Vector2d** &w)
Retruns (v.x / w.x, v.y / w.y).
- double **operator|** (const **Vector2d** &v, const **Vector2d** &w)
Retruns the euclidean scalar product of the two vectors.

7.49.1 Detailed Description

An object that represents a 2D vector.

All usual operations +,-,*,/ are defined component wise. The | operator is used for the euclidean scalar product.

7.49.2 Constructor & Destructor Documentation

7.49.2.1 Vector2d() [1/2]

```
Vector2d::Vector2d (
    double x )
```

Returns the vector (x,x).

Parameters

<i>x</i>	a real number.
----------	----------------

7.49.2.2 Vector2d() [2/2]

```
Vector2d::Vector2d (
    double x,
    double y )
```

Returns the vector (*x*,*y*).

Parameters

<i>x</i>	a real number.
<i>y</i>	a real number.

7.49.3 Member Function Documentation**7.49.3.1 norm()**

```
double Vector2d::norm ( ) const
```

Returns the euclidean norm of the vector.

Returns

double

7.49.3.2 operator*=()

```
Vector2d & Vector2d::operator*= (
    const Vector2d & w )
```

Stores in *this the multiplication of *this and *w*.

Equivalent to *this = *this * *w*;

Parameters

<i>w</i>	a 2D vector.
----------	--------------

Returns

[Vector2d](#)&

7.49.3.3 operator+()

```
Vector2d & Vector2d::operator+ ( )
```

Returns itself.

Returns

[Vector2d](#)&

7.49.3.4 operator+=()

```
Vector2d & Vector2d::operator+= (
    const Vector2d & w )
```

Stores in *this the sum of *this and w.

Equivalent to *this = *this + w;

Parameters

<i>w</i>	a 2D vector.
----------	--------------

Returns

[Vector2d](#)&

7.49.3.5 operator-()

```
Vector2d & Vector2d::operator- ( )
```

Multiplies all components by -1.

Returns

[Vector2d](#)&

7.49.3.6 operator-=()

```
Vector2d & Vector2d::operator-= (
    const Vector2d & w )
```

Stores in *this the subtraction of *this and w.

Equivalent to `*this = *this - w;`

Parameters

w	a 2D vector.
---	--------------

Returns

Vector2d&

7.49.3.7 operator/=()

```
Vector2d & Vector2d::operator/= (
    const Vector2d & w )
```

Stores in *this the division of *this and w.

Equivalent to `*this = *this / w;`

Parameters

w	
---	--

Returns

Vector2d&

7.49.3.8 perp()

```
Vector2d Vector2d::perp ( ) const
```

Returns a perpendicular vector.

The returned vector is always positioned 90° degrees anticlockwise.

7.49.3.9 to_string()

```
std::string Vector2d::to_string ( ) const
```

Retruns a string representation of the vector.

Returns

std::string

7.49.4 Friends And Related Function Documentation

7.49.4.1 operator*

```
Vector2d operator* (
    const Vector2d & v,
    const Vector2d & w ) [friend]
```

Returns $(v.x * w.x, v.y * w.y)$.

Parameters

v	a 2D vector.
w	a 2D vector.

Returns

Vector2d

7.49.4.2 operator+

```
Vector2d operator+ (
    const Vector2d & v,
    const Vector2d & w ) [friend]
```

Returns $(v.x + w.x, v.y + w.y)$.

Parameters

v	a 2D vector.
w	a 2D vector.

Returns

[Vector2d](#)**7.49.4.3 operator-**

```
Vector2d operator- (
    const Vector2d & v,
    const Vector2d & w ) [friend]
```

Retruns (v.x - w.x, v.y - w.y).

Parameters

<i>v</i>	a 2D vector.
<i>w</i>	a 2D vector.

Returns

[Vector2d](#)**7.49.4.4 operator/**

```
Vector2d operator/ (
    const Vector2d & v,
    const Vector2d & w ) [friend]
```

Retruns (v.x / w.x, v.y / w.y).

Parameters

<i>v</i>	a 2D vector.
<i>w</i>	a 2D vector.

Returns

[Vector2d](#)**7.49.4.5 operator" |**

```
double operator| (
    const Vector2d & v,
    const Vector2d & w ) [friend]
```

Retruns the euclidean scalar product of the two vectors.

Parameters

v	a 2D vector.
w	a 2D vector.

Returns

double

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

- CPGF/AffineSpace2d/Vector2d.hpp
- CPGF/AffineSpace2d/Vector2d.cpp

Chapter 8

File Documentation

8.1 Chemistry/Reaction.hpp File Reference

This file contains objects used to store constants and to compute the kinetics of chemical reactions.

```
#include <cstdio>
#include <string>
```

Classes

- class [Chemistry::SolidGasReaction](#)

This object is used to store all properties of a solid reactant and a gas product. It is also used to compute the speed of the combustion front.

Namespaces

- namespace [Chemistry](#)

The objects of this library are used to store constants and to compute the speed of Chemical Reactions.

8.1.1 Detailed Description

This file contains objects used to store constants and to compute the kinetics of chemical reactions.

Author

Andrés Laín Sanclemente

Version

0.2.0

Date

9th September 2021

8.2 Reaction.hpp

[Go to the documentation of this file.](#)

```

1
11 #ifndef REACTION_HPP
12 #define REACTION_HPP
13
14 #include <stdio>
15 #include <string>
16
22 namespace Chemistry
23 {
30     class SolidGasReaction
31     {
32     public:
33
38         double rho_s;
39
44         double k_s;
45
50         double cV_s;
51
56         double alpha;
57
62         double cV_g;
63
68         double R_g;
69
75         double gamma;
76
81         double a;
82
87         double P_ref;
88
93         double n;
94
99         double delta_H;
100
107         double v_q(const double P) const;
108
122         SolidGasReaction(const double rho_s = 0, const double k_s = 0, const double cV_s = 0,
123             const double cV_g = 0, const double R_g = 0, const double a = 0,
124             const double P_ref = 0, const double n = 0, const double delta_H = 0);
125
131         explicit SolidGasReaction(FILE* file);
132
138         void read_from_file(FILE* file);
139
145         void write_to_file(FILE* file) const;
146
152         std::string to_string() const;
153     };
154 }
155
156 #endif // REACTION_HPP

```

8.3 Point2d.hpp

```

1
4 #ifndef POINT2D_HPP
5 #define POINT2D_HPP
6
7 #include "Vector2d.hpp"
8
9 namespace CPGF
10 {
11     namespace AffineSpace
12     {
20         class Point2d
21         {
22         public:
27             double x;
32             double y;
33
44             friend Point2d operator+(const Point2d& P, const Vector2d& v);
45
56             friend Vector2d operator-(const Point2d& P, const Point2d& Q);
57
66             friend bool operator==(const Point2d& P, const Point2d& Q);
67
76             Point2d& operator+= (const Vector2d& v);

```

```

77
78      double angle_with_respect_to(const Point2d& Q);
79
80      Point2d& rotate_with_respect_to(const Point2d& Q, const double theta);
81
82      Point2d& rotate_to_with_respect_to(const Point2d& Q, const double theta);
83
84      Point2d& scale_with_respect_to(const Point2d& Q, const Vector2d& s);
85
86      std::string to_string() const;
87
88      Point2d();
89
90      Point2d(double x, double y);
91
92 };
93
94 }
95
96 }
97
98 #endif // POINT2D_HPP

```

8.4 Vector2d.hpp

```

1  #ifndef VECTOR2D_HPP
2  #define VECTOR2D_HPP
3
4  #include <string>
5
6  namespace CPGF
7  {
8      namespace AffineSpace
9      {
10         class Vector2d
11         {
12         public:
13             double x;
14
15             double y;
16
17             friend Vector2d operator+(const Vector2d& v, const Vector2d& w);
18
19             friend Vector2d operator-(const Vector2d& v, const Vector2d& w);
20
21             friend Vector2d operator*(const Vector2d& v, const Vector2d& w);
22
23             friend Vector2d operator/(const Vector2d& v, const Vector2d& w);
24
25             friend double operator|(const Vector2d& v, const Vector2d& w);
26
27             Vector2d& operator+();
28
29             Vector2d& operator-();
30
31             Vector2d& operator+= (const Vector2d& w);
32
33             Vector2d& operator-= (const Vector2d& w);
34
35             Vector2d& operator*= (const Vector2d& w);
36
37             Vector2d& operator/= (const Vector2d& w);
38
39             Vector2d perp() const;
40
41             double norm() const;
42
43             std::string to_string() const;
44
45             Vector2d();
46
47             Vector2d(double x);
48
49             Vector2d(double x, double y);
50
51         };
52     }
53 }
54
55 #endif // VECTOR2D_HPP

```

8.5 CPGF.hpp

```

1 #ifndef CPGF_HPP
2 #define CPGF_HPP
3
4 #include "AffineSpace2d/Vector2d.hpp"
5 #include "AffineSpace2d/Point2d.hpp"
6 #include "PGFBasics/PGFConf.hpp"
7 #include "PGFBasics/Strokes2d.hpp"
8 #include "PGFBasics/Path2d.hpp"
9 #include "Objects2d/Object2d.hpp"
10 #include "Objects2d/BasicGeometries.hpp"
11 #include "Plot2d/Axis.hpp"
12 #include "Plot2d/DataPlot.hpp"
13 #include "Plot2d/Graphic.hpp"
14 #include "Plot2d/LinePlot.hpp"
15 #include "Text/Text.hpp"
16 #include "Scene2d.hpp"
17
18 #endif // CPGF_HPP

```

8.6 BasicGeometries.hpp

```

1 #ifndef BASICGEOMETRIES_HPP
2 #define BASICGEOMETRIES_HPP
3
4 #include "Object2d.hpp"
5 #include "../PGFBasics/Path2d.hpp"
6
7 namespace CPGF
8 {
9     namespace Objects2d
10     {
11         class Line: public Object2d
12         {
13         public:
14             Line(const AffineSpace::Point2d& A, const AffineSpace::Point2d& B,
15                 const Color& color = Color::BLACK, const double opacity = 1,
16                 const double line_width = LineWidth::SEMITHICK,
17                 const std::vector<double>& dash_pattern = DashPatterns::SOLID);
18             Line(const std::vector<AffineSpace::Point2d>& points,
19                 const Color& color = Color::BLACK, const double opacity = 1,
20                 const double line_width = LineWidth::SEMITHICK,
21                 const std::vector<double>& dash_pattern = DashPatterns::SOLID);
22
23             AffineSpace::Point2d start();
24             AffineSpace::Point2d end();
25
26         protected:
27             static Object2d builder(const std::vector<AffineSpace::Point2d>& points,
28                                     const Color& color, const double opacity, const double line_width,
29                                     const std::vector<double>& dash_pattern);
30         };
31
32         class Circle: public Object2d
33         {
34         public:
35             Circle(const AffineSpace::Point2d& pos, const double radius,
36                 const bool draw = true, const bool fill = false,
37                 const Color& draw_color = Color::BLACK,
38                 const Color& fill_color = Color::WHITE,
39                 const double opacity = 1,
40                 const double line_width = LineWidth::SEMITHICK,
41                 const std::vector<double>& dash_pattern = DashPatterns::SOLID);
42
43             AffineSpace::Point2d center() const;
44             double radius() const;
45
46         protected:
47             static Object2d builder(const AffineSpace::Point2d& pos, const double radius,
48                                     const bool draw, const bool fill, const Color& draw_color,
49                                     const Color& fill_color, const double opacity, const double line_width,
50                                     const std::vector<double>& dash_pattern);
51         };
52
53         class Arrow: public Object2d
54         {
55         public:
56             Arrow(const AffineSpace::Point2d& start,
57                 const AffineSpace::Point2d& end,
58                 const double arrow_head_length = 0.15, const double arrow_head_width = 0.3,
59                 const Color& color = Color::BLACK, const double opacity = 1,
60                 const double line_width = LineWidth::SEMITHICK,

```



```

61         const std::vector<double>& dash_pattern = DashPatterns::SOLID);
62
63     protected:
64     static Object2d builder(const AffineSpace::Point2d& start,
65         const AffineSpace::Point2d& end,
66         const double arrow_head_length, const double arrow_head_width,
67         const Color& color, const double opacity, const double line_width,
68         const std::vector<double>& dash_pattern);
69     };
70 }
71 }
72
73 #endif // BASICGEOMETRIES_HPP

```

8.7 Object2d.hpp

```

1  #ifndef OBJECT2D_HPP
2  #define OBJECT2D_HPP
3
4  #include <string>
5  #include <vector>
6  #include "../PGFBasics/PGFConf.hpp"
7  #include "../PGFBasics/Path2d.hpp"
8
9  namespace CPGF
10 {
11     namespace Objects2d
12     {
13         class Object2d
14         {
15         public:
16             std::vector<Basics::Path2d> paths;
17
18             Object2d();
19             Object2d(const Basics::Path2d& path);
20             Object2d(const std::vector<Basics::Path2d>& paths);
21
22             unsigned int size() const;
23
24             Object2d& translate(const AffineSpace::Vector2d& v);
25             Object2d& rotate_with_respect_to(const AffineSpace::Point2d& Q, const double theta);
26             Object2d& scale_with_respect_to(const AffineSpace::Point2d& Q, const AffineSpace::Vector2d&
27                 s);
28
29             friend Object2d operator+(const Object2d& A, const Object2d& B);
30             Object2d& operator+=(const Object2d& B);
31             Object2d& operator+=(const Basics::Path2d& path);
32
33             std::string render_to_string() const;
34         };
35     }
36 }
37 #endif // OBJECT2D_HPP

```

8.8 Path2d.hpp

```

1  #ifndef PATH2D_HPP
2  #define PATH2D_HPP
3
4  #include "PGFConf.hpp"
5  #include "Strokes2d.hpp"
6  #include <vector>
7
8  namespace CPGF
9  {
10     namespace Basics
11     {
12         class Path2d
13         {
14         public:
15             std::vector<SimpleStroke2d*> strokes;
16             PGFConf conf;
17
18             Path2d();
19             Path2d(SimpleStroke2d& stroke,
20                 const PGFConf& conf = PGFConf());
21             Path2d(const std::vector<SimpleStroke2d*>& strokes,

```

```

29         const PGFConf& conf = PGFConf();
30         Path2d(const Path2d& path);
31         Path2d& operator=(const Path2d& path);
32         ~Path2d();
33
34         Path2d& translate(const AffineSpace::Vector2d& v);
35         Path2d& rotate_with_respect_to(const AffineSpace::Point2d& Q, const double theta);
36         Path2d& scale_with_respect_to(const AffineSpace::Point2d& Q, const AffineSpace::Vector2d& v);
37
38         AffineSpace::Point2d start();
39         AffineSpace::Point2d start() const;
40         AffineSpace::Point2d end();
41         AffineSpace::Point2d end() const;
42
43         double length() const;
44         double area() const;
45
46         Path2d& operator+=(SimpleStroke2d& stroke);
47         Path2d& add_stroke(SimpleStroke2d& stroke);
48         unsigned int size() const;
49         std::string render_to_string() const;
50     };
51 }
52 }
53
54 #endif // PATH2D_HPP

```

8.9 PGFConf.hpp

```

1  #ifndef PGFCONF_HPP
2  #define PGFCONF_HPP
3
4  #include <vector>
5  #include <string>
6
7  namespace CPGF
8  {
9      enum class DrawType
10     {
11         DRAW,
12         FILL,
13     };
14
15     class LineWidth
16     {
17     public:
18         static const double ULTRA_THIN;
19         static const double VERY_THIN;
20         static const double THIN;
21         static const double SEMITHICK;
22         static const double THICK;
23         static const double VERY_THICK;
24         static const double ULTRA_THICK;
25     };
26
27     enum class LineCap
28     {
29         ROUND,
30         RECT,
31         BUTT
32     };
33
34     enum class LineJoin
35     {
36         ROUND,
37         BEVEL,
38         MITER
39     };
40
41     // Remember miter limit. Currently not available.
42
43     class Color
44     {
45     public:
46         double r;
47
48         double g;
49
50         double b;
51
52         std::string to_string();
53
54         Color();
55     };
56 }
57
58 #endif

```

```

101
109     Color(const double r, const double g, const double b);
110
120     static Color mix(const Color& A, const Color& B, const double alpha);
121
122     static Color from_RGB(const unsigned char R, const unsigned char G, const unsigned char B);
123
124     // Color list taken from https://www.rapidtables.com/web/color/RGB_Color.html
125     static const Color MAROON;
126     static const Color DARK_RED;
127     static const Color BROWN;
128     static const Color FIREBRICK;
129     static const Color CRIMSON;
130     static const Color RED;
131     static const Color TOMATO;
132     static const Color CORAL;
133     static const Color INDIAN_RED;
134     static const Color LIGHT_CORAL;
135     static const Color DARK_SALMON;
136     static const Color SALMON;
137     static const Color LIGHT_SALMON;
138     static const Color ORANGE_RED;
139     static const Color DARK_ORANGE;
140     static const Color ORANGE;
141     static const Color GOLD;
142     static const Color DARK_GOLDEN_ROD;
143     static const Color GOLDEN_ROD;
144     static const Color PALE_GOLDEN_ROD;
145     static const Color DARK_KHAKI;
146     static const Color KHAKI;
147     static const Color OLIVE;
148     static const Color YELLOW;
149     static const Color YELLOW_GREEN;
150     static const Color DARK_OLIVE_GREEN;
151     static const Color OLIVE_DRAB;
152     static const Color LAWN_GREEN;
153     static const Color CHART_REUSE;
154     static const Color GREEN_YELLOW;
155     static const Color DARK_GREEN;
156     static const Color GREEN;
157     static const Color FOREST_GREEN;
158     static const Color LIME;
159     static const Color LIME_GREEN;
160     static const Color LIGHT_GREEN;
161     static const Color PALE_GREEN;
162     static const Color DARK_SEA_GREEN;
163     static const Color MEDIUM_SPRING_GREEN;
164     static const Color SPRING_GREEN;
165     static const Color SEA_GREEN;
166     static const Color MEDIUM_AQUA_MARINE;
167     static const Color MEDIUM_SEA_GREEN;
168     static const Color LIGHT_SEA_GREEN;
169     static const Color DARK_SLATE_GRAY;
170     static const Color TEAL;
171     static const Color DARK_CYAN;
172     static const Color CYAN;
173     static const Color LIGHT_CYAN;
174     static const Color DARK_TURQUOISE;
175     static const Color TURQUOISE;
176     static const Color MEDIUM_TURQUOISE;
177     static const Color PALE_TURQUOISE;
178     static const Color AQUA_MARINE;
179     static const Color POWDER_BLUE;
180     static const Color CADET_BLUE;
181     static const Color STEEL_BLUE;
182     static const Color CORN_FLOWER_BLUE;
183     static const Color DEEP_SKY_BLUE;
184     static const Color DODGER_BLUE;
185     static const Color LIGHT_BLUE;
186     static const Color SKY_BLUE;
187     static const Color LIGHT_SKY_BLUE;
188     static const Color MIDNIGHT_BLUE;
189     static const Color NAVY;
190     static const Color DARK_BLUE;
191     static const Color MEDIUM_BLUE;
192     static const Color BLUE;
193     static const Color ROYAL_BLUE;
194     static const Color BLUE_VIOLET;
195     static const Color INDIGO;
196     static const Color DARK_SLATE_BLUE;
197     static const Color SLATE_BLUE;
198     static const Color MEDIUM_SLATE_BLUE;
199     static const Color MEDIUM_PURPLE;
200     static const Color DARK_MAGENTA;
201     static const Color DARK_VIOLET;
202     static const Color DARK_ORCHID;
203     static const Color MEDIUM_ORCHID;

```

```

204     static const Color PURPLE;
205     static const Color THISTLE;
206     static const Color PLUM;
207     static const Color VIOLET;
208     static const Color MAGENTA;
209     static const Color ORCHID;
210     static const Color MEDIUM_VIOLET_RED;
211     static const Color PALE_VIOLET_RED;
212     static const Color DEEP_PINK;
213     static const Color HOT_PINK;
214     static const Color LIGHT_PINK;
215     static const Color PINK;
216     static const Color ANTIQUE_WHITE;
217     static const Color BEIGE;
218     static const Color BISQUE;
219     static const Color BLANCHED_ALMOND;
220     static const Color WHEAT;
221     static const Color CORN_SILK;
222     static const Color LEMON_CHIFFON;
223     static const Color LIGHT_GOLDEN_ROD_YELLOW;
224     static const Color LIGHT_YELLOW;
225     static const Color SADDLE_BROWN;
226     static const Color SIENNA;
227     static const Color CHOCOLATE;
228     static const Color PERU;
229     static const Color SANDY_BROWN;
230     static const Color BURLY_WOOD;
231     static const Color TAN;
232     static const Color ROSY_BROWN;
233     static const Color MOCCASIN;
234     static const Color NAVAJO_WHITE;
235     static const Color PEACH_STUFF;
236     static const Color MISTY_ROSE;
237     static const Color LAVENDER_BLUSH;
238     static const Color LINEN;
239     static const Color OLD_LACE;
240     static const Color PAPAYA_WHIP;
241     static const Color SEA_SHELL;
242     static const Color MINT_CREAM;
243     static const Color SLATE_GRAY;
244     static const Color LIGHT_SLATE_GRAY;
245     static const Color LIGHT_STEEL_BLUE;
246     static const Color LAVENDER;
247     static const Color FLORAL_WHITE;
248     static const Color ALICE_BLUE;
249     static const Color GHOST_WHITE;
250     static const Color HONEYDEW;
251     static const Color IVORY;
252     static const Color AZURE;
253     static const Color SNOW;
254     static const Color BLACK;
255     static const Color DIM_GRAY;
256     static const Color GRAY;
257     static const Color DARK_GRAY;
258     static const Color SILVER;
259     static const Color LIGHT_GRAY;
260     static const Color GAINSBORO;
261     static const Color WHITE_SMOKE;
262     static const Color WHITE;
263 };
264
265 class DashPatterns
266 {
267     public:
268     static const std::vector<double> SOLID;
269     static const std::vector<double> DASHED;
270     static const std::vector<double> DOTTED;
271 };
272
273 class PGFConf
274 {
275     public:
276     DrawType draw_type;
277     LineCap line_cap;
278     LineJoin line_join;
279     Color color;
280
281     std::vector<double> dash_pattern;
282
283     double dash_phase;
284
285     double opacity;
286
287     double line_width;
288
289     PGFConf(const DrawType draw_type = DrawType::DRAW,
290            const Color& color = Color::BLACK,

```

```

325         const double opacity = 1,
326         const double line_width = LineWidth::SEMITHICK,
327         const std::vector<double>& dash_pattern = std::vector<double>(),
328         const double dash_phase = 0,
329         const LineCap line_cap = LineCap::BUTT,
330         const LineJoin line_join = LineJoin::BEVEL
331     );
332 };
333 }
334
335
336 #endif // TIKZCONF_HPP

```

8.10 Strokes2d.hpp

```

1  #ifndef STROKES2D_HPP
2  #define STROKES2D_HPP
3
4  #include "../AffineSpace2d/Point2d.hpp"
5  #include <vector>
6
7  namespace CPGF
8  {
9      namespace Basics
10     {
11         class SimpleStroke2d
12         {
13         public:
14             // Clone function and destructor.
15             virtual SimpleStroke2d* clone() const = 0;
16             virtual ~SimpleStroke2d() = default;
17
18             // Start and end of the stroke.
19             virtual AffineSpace::Point2d& start() = 0;
20             virtual AffineSpace::Point2d start() const = 0;
21             virtual AffineSpace::Point2d& end() = 0;
22             virtual AffineSpace::Point2d end() const = 0;
23
24             // Invert stroke.
25             virtual SimpleStroke2d& invert() = 0;
26
27             // Basic affine transformations.
28             virtual SimpleStroke2d& translate(const AffineSpace::Vector2d& v) = 0;
29             virtual SimpleStroke2d& rotate_with_respect_to(const AffineSpace::Point2d& Q, const double
theta) = 0;
30             virtual SimpleStroke2d& scale_with_respect_to(const AffineSpace::Point2d& Q, const
AffineSpace::Vector2d& s) = 0;
31
32             // Length and area.
33             virtual double length() const = 0;
34             virtual double area() const = 0;
35
36             // Intersections.
37             virtual std::vector<AffineSpace::Point2d> operator/(const SimpleStroke2d& B) = 0;
38
39             // Return point in specific position. Tangent and normal unitary vectors.
40             // virtual AffineSpace::Point2d point_at_position(const double alpha) const = 0;
41             // virtual AffineSpace::Vector2d tangent_vector_at_position(const double alpha) const = 0;
42             // virtual AffineSpace::Vector2d normal_vector_at_position(const double alpha) const = 0;
43
44             // // Closest point.
45             // virtual double closest_position(const AffineSpace::Point2d& P) const = 0;
46             // virtual AffineSpace::Point2d closest_point(const AffineSpace::Point2d& P) const = 0;
47
48             // Rendering to string.
49             virtual std::string render_to_string() const = 0;
50         };
51
52         class StraightStroke2d;
53         class BezierStroke2d;
54
55         class StraightStroke2d: public SimpleStroke2d
56         {
57         public:
58             std::vector<AffineSpace::Point2d> points;
59
60             StraightStroke2d();
61             StraightStroke2d(const std::vector<AffineSpace::Point2d> points);
62             StraightStroke2d* clone() const override;
63
64             StraightStroke2d& operator+=(const AffineSpace::Point2d& P);
65             StraightStroke2d& add_point(const AffineSpace::Point2d& P);
66             unsigned int size() const;

```

```

73
74     AffineSpace::Point2d& start() override;
75     AffineSpace::Point2d start() const override;
76     AffineSpace::Point2d& end() override;
77     AffineSpace::Point2d end() const override;
78     StraightStroke2d& translate(const AffineSpace::Vector2d& v) override;
79     StraightStroke2d& rotate_with_respect_to(const AffineSpace::Point2d& Q, const double theta)
override;
80     StraightStroke2d& scale_with_respect_to(const AffineSpace::Point2d& Q, const
AffineSpace::Vector2d& s) override;
81
82     double length() const override;
83     double area() const override;
84
85     std::vector<AffineSpace::Point2d> operator/(const SimpleStroke2d& B) override;
86     std::vector<AffineSpace::Point2d> operator/(const StraightStroke2d& B);
87     // std::vector<AffineSpace::Point2d> operator/(const BezierStroke2d& B);
88
89     std::string render_to_string() const override;
90 };
91
92 class BezierStroke2d: public SimpleStroke2d
93 {
94 public:
95     // Start and end points.
96     AffineSpace::Point2d P1;
97     AffineSpace::Point2d P2;
98
99     // Control Points.
100    AffineSpace::Point2d Q1;
101    AffineSpace::Point2d Q2;
102
103    BezierStroke2d();
104    BezierStroke2d(const AffineSpace::Point2d& P1, const AffineSpace::Point2d& P2,
const AffineSpace::Point2d& Q1, const AffineSpace::Point2d& Q2);
105
106    BezierStroke2d* clone() const override;
107
108    AffineSpace::Point2d& start() override;
109    AffineSpace::Point2d start() const override;
110    AffineSpace::Point2d& end() override;
111    AffineSpace::Point2d end() const override;
112    BezierStroke2d& translate(const AffineSpace::Vector2d& v) override;
113    BezierStroke2d& rotate_with_respect_to(const AffineSpace::Point2d& Q, const double theta)
override;
114    BezierStroke2d& scale_with_respect_to(const AffineSpace::Point2d& Q, const
AffineSpace::Vector2d& s) override;
115
116    double length() const override;
117    double area() const override;
118
119    std::vector<AffineSpace::Point2d> operator/(const SimpleStroke2d& B) override;
120    // std::vector<AffineSpace::Point2d> operator/(const StraightStroke2d& B);
121    // std::vector<AffineSpace::Point2d> operator/(const BezierStroke2d& B);
122
123    std::string render_to_string() const override;
124 };
125 }
126
127 #endif // STROKES2D_HPP

```

8.11 Axis.hpp

```

1 #ifndef AXIS_HPP
2 #define AXIS_HPP
3
4 #include "../PGFBasics/PGFConf.hpp"
5 #include "../Scene2d.hpp"
6 #include <functional>
7 #include "../Utilities/FormatNumber.hpp"
8 #include "../Objects2d/BasicGeometries.hpp"
9 #include <cmath>
10
11 namespace CPGF
12 {
13     namespace Plot2d
14     {
15         enum class AxisType
16         {
17             HORIZONTAL,
18             VERTICAL,
19         };
20     }
21 }

```

```

20
21     enum class AxisScale
22     {
23         LINEAR,
24         LOG,
25     };
26
27     enum class NumberPosition
28     {
29         LEFT,
30         RIGHT,
31     };
32
33     class Axis
34     {
35     public:
36
37         AxisType axis_type;
38
39         AxisScale axis_scale;
40
41         double scale;
42
43         bool visible;
44
45         bool inverted;
46
47         std::string label;
48
49         double position;
50
51         Color color;
52
53         double line_width;
54
55         double opacity;
56
57         std::vector<double> dash_pattern;
58
59         double aspect_ratio;
60
61         double arrow_head_length;
62
63         double arrow_head_width;
64
65         double arrow_length;
66
67         std::function<Objects2d::Object2d(const AffineSpace::Point2d& start,
68             const AffineSpace::Point2d& end,
69             const double arrow_head_length, const double arrow_head_width,
70             const Color& color, const double opacity, const double line_width,
71             const std::vector<double> dash_pattern)> arrow;
72
73         unsigned int N_major_ticks;
74
75         double major_tick_deviation;
76
77         double major_tick_line_width_divisor;
78
79         bool show_medium_ticks;
80
81         double medium_tick_deviation;
82
83         double medium_tick_line_width_divisor;
84
85         bool show_small_ticks;
86
87         double small_tick_deviation;
88
89         double small_tick_line_width_divisor;
90
91         bool show_numbers;
92
93         NumberPosition number_position;
94
95         bool show_major_grid_lines;
96
97         std::vector<double> major_grid_lines_dash_pattern;
98
99         double major_grid_line_line_width_divisor;
100
101         double major_grid_line_opacity;
102
103         bool show_medium_grid_lines;
104
105         std::vector<double> medium_grid_lines_dash_pattern;
106
107     };
108
109     double axis_arrow_length(const Axis& axis);
110
111     double axis_arrow_head_length(const Axis& axis);
112
113     double axis_arrow_head_width(const Axis& axis);
114
115     double axis_arrow_deviation(const Axis& axis);
116
117     double axis_tick_deviation(const Axis& axis, unsigned int N_ticks);
118
119     double axis_tick_line_width(const Axis& axis, unsigned int N_ticks);
120
121     double axis_line_width(const Axis& axis);
122
123     double axis_opacity(const Axis& axis);
124
125     double axis_aspect_ratio(const Axis& axis);
126
127     double axis_label_deviation(const Axis& axis);
128
129     double axis_label_line_width(const Axis& axis);
130
131     double axis_label_opacity(const Axis& axis);
132
133     double axis_number_deviation(const Axis& axis);
134
135     double axis_number_line_width(const Axis& axis);
136
137     double axis_number_opacity(const Axis& axis);
138
139     double axis_major_grid_line_deviation(const Axis& axis);
140
141     double axis_major_grid_line_line_width(const Axis& axis);
142
143     double axis_major_grid_line_opacity(const Axis& axis);
144
145     double axis_medium_grid_line_deviation(const Axis& axis);
146
147     double axis_medium_grid_line_line_width(const Axis& axis);
148
149     double axis_medium_grid_line_opacity(const Axis& axis);
150
151     double axis_small_grid_line_deviation(const Axis& axis);
152
153     double axis_small_grid_line_line_width(const Axis& axis);
154
155     double axis_small_grid_line_opacity(const Axis& axis);
156
157     double axis_line_deviation(const Axis& axis);
158
159     double axis_line_line_width(const Axis& axis);
160
161     double axis_line_opacity(const Axis& axis);
162
163     double axis_dash_pattern(const Axis& axis);
164
165     double axis_dash_deviation(const Axis& axis);
166
167     double axis_dash_line_width(const Axis& axis);
168
169     double axis_dash_opacity(const Axis& axis);
170
171     double axis_dash_aspect_ratio(const Axis& axis);
172
173     double axis_dash_label_deviation(const Axis& axis);
174
175     double axis_dash_label_line_width(const Axis& axis);
176
177     double axis_dash_label_opacity(const Axis& axis);
178
179     double axis_dash_number_deviation(const Axis& axis);
180
181     double axis_dash_number_line_width(const Axis& axis);
182
183     double axis_dash_number_opacity(const Axis& axis);
184
185     double axis_dash_major_grid_line_deviation(const Axis& axis);
186
187     double axis_dash_major_grid_line_line_width(const Axis& axis);
188
189     double axis_dash_major_grid_line_opacity(const Axis& axis);
190
191     double axis_dash_medium_grid_line_deviation(const Axis& axis);
192
193     double axis_dash_medium_grid_line_line_width(const Axis& axis);
194
195     double axis_dash_medium_grid_line_opacity(const Axis& axis);
196
197     double axis_dash_small_grid_line_deviation(const Axis& axis);
198
199     double axis_dash_small_grid_line_line_width(const Axis& axis);
200
201     double axis_dash_small_grid_line_opacity(const Axis& axis);
202
203     double axis_dash_line_deviation(const Axis& axis);
204
205     double axis_dash_line_line_width(const Axis& axis);
206
207     double axis_dash_line_opacity(const Axis& axis);
208
209     double axis_dash_dash_deviation(const Axis& axis);
210
211     double axis_dash_dash_line_width(const Axis& axis);
212
213     double axis_dash_dash_opacity(const Axis& axis);
214
215     double axis_dash_dash_aspect_ratio(const Axis& axis);
216
217     double axis_dash_dash_label_deviation(const Axis& axis);
218
219     double axis_dash_dash_label_line_width(const Axis& axis);
220
221     double axis_dash_dash_label_opacity(const Axis& axis);
222
223     double axis_dash_dash_number_deviation(const Axis& axis);
224
225     double axis_dash_dash_number_line_width(const Axis& axis);
226
227     double axis_dash_dash_number_opacity(const Axis& axis);
228
229     double axis_dash_dash_major_grid_line_deviation(const Axis& axis);
230
231     double axis_dash_dash_major_grid_line_line_width(const Axis& axis);
232
233     double axis_dash_dash_major_grid_line_opacity(const Axis& axis);
234
235     double axis_dash_dash_medium_grid_line_deviation(const Axis& axis);
236
237     double axis_dash_dash_medium_grid_line_line_width(const Axis& axis);
238
239     double axis_dash_dash_medium_grid_line_opacity(const Axis& axis);
240
241     double axis_dash_dash_small_grid_line_deviation(const Axis& axis);
242
243     double axis_dash_dash_small_grid_line_line_width(const Axis& axis);
244
245     double axis_dash_dash_small_grid_line_opacity(const Axis& axis);
246
247     double axis_dash_dash_line_deviation(const Axis& axis);
248
249     double axis_dash_dash_line_line_width(const Axis& axis);
250
251     double axis_dash_dash_line_opacity(const Axis& axis);
252
253     double axis_dash_dash_dash_deviation(const Axis& axis);
254
255     double axis_dash_dash_dash_line_width(const Axis& axis);
256
257     double axis_dash_dash_dash_opacity(const Axis& axis);
258
259     double axis_dash_dash_dash_aspect_ratio(const Axis& axis);
260
261     double axis_dash_dash_dash_label_deviation(const Axis& axis);
262
263     double axis_dash_dash_dash_label_line_width(const Axis& axis);
264
265     double axis_dash_dash_dash_label_opacity(const Axis& axis);
266
267     double axis_dash_dash_dash_number_deviation(const Axis& axis);
268
269     double axis_dash_dash_dash_number_line_width(const Axis& axis);
270
271     double axis_dash_dash_dash_number_opacity(const Axis& axis);
272
273     double axis_dash_dash_dash_major_grid_line_deviation(const Axis& axis);
274
275     double axis_dash_dash_dash_major_grid_line_line_width(const Axis& axis);
276
277     double axis_dash_dash_dash_major_grid_line_opacity(const Axis& axis);
278
279     double axis_dash_dash_dash_medium_grid_line_deviation(const Axis& axis);
280
281     double axis_dash_dash_dash_medium_grid_line_line_width(const Axis& axis);
282
283     double axis_dash_dash_dash_medium_grid_line_opacity(const Axis& axis);
284
285     double axis_dash_dash_dash_small_grid_line_deviation(const Axis& axis);
286
287     double axis_dash_dash_dash_small_grid_line_line_width(const Axis& axis);
288
289     double axis_dash_dash_dash_small_grid_line_opacity(const Axis& axis);
290
291     double axis_dash_dash_dash_line_deviation(const Axis& axis);
292
293     double axis_dash_dash_dash_line_line_width(const Axis& axis);
294
295     double axis_dash_dash_dash_line_opacity(const Axis& axis);
296
297     double axis_dash_dash_dash_dash_deviation(const Axis& axis);
298
299     double axis_dash_dash_dash_dash_line_width(const Axis& axis);
300
301     double axis_dash_dash_dash_dash_opacity(const Axis& axis);
302
303     double axis_dash_dash_dash_dash_aspect_ratio(const Axis& axis);
304
305     double axis_dash_dash_dash_dash_label_deviation(const Axis& axis);
306
307     double axis_dash_dash_dash_dash_label_line_width(const Axis& axis);
308
309     double axis_dash_dash_dash_dash_label_opacity(const Axis& axis);
310
311     double axis_dash_dash_dash_dash_number_deviation(const Axis& axis);
312
313     double axis_dash_dash_dash_dash_number_line_width(const Axis& axis);
314
315     double axis_dash_dash_dash_dash_number_opacity(const Axis& axis);
316
317     double axis_dash_dash_dash_dash_major_grid_line_deviation(const Axis& axis);
318
319     double axis_dash_dash_dash_dash_major_grid_line_line_width(const Axis& axis);
320
321     double axis_dash_dash_dash_dash_major_grid_line_opacity(const Axis& axis);
322
323     double axis_dash_dash_dash_dash_medium_grid_line_deviation(const Axis& axis);
324
325     double axis_dash_dash_dash_dash_medium_grid_line_line_width(const Axis& axis);
326
327     double axis_dash_dash_dash_dash_medium_grid_line_opacity(const Axis& axis);
328
329     double axis_dash_dash_dash_dash_small_grid_line_deviation(const Axis& axis);
330
331     double axis_dash_dash_dash_dash_small_grid_line_line_width(const Axis& axis);
332
333     double axis_dash_dash_dash_dash_small_grid_line_opacity(const Axis& axis);
334
335     double axis_dash_dash_dash_dash_line_deviation(const Axis& axis);
336
337     double axis_dash_dash_dash_dash_line_line_width(const Axis& axis);
338
339     double axis_dash_dash_dash_dash_line_opacity(const Axis& axis);
340
341     double axis_dash_dash_dash_dash_dash_deviation(const Axis& axis);
342
343     double axis_dash_dash_dash_dash_dash_line_width(const Axis& axis);
344
345     double axis_dash_dash_dash_dash_dash_opacity(const Axis& axis);
346
347     double axis_dash_dash_dash_dash_dash_aspect_ratio(const Axis& axis);
348
349     double axis_dash_dash_dash_dash_dash_label_deviation(const Axis& axis);
350
351     double axis_dash_dash_dash_dash_dash_label_line_width(const Axis& axis);
352
353     double axis_dash_dash_dash_dash_dash_label_opacity(const Axis& axis);
354
355     double axis_dash_dash_dash_dash_dash_number_deviation(const Axis& axis);
356
357     double axis_dash_dash_dash_dash_dash_number_line_width(const Axis& axis);
358
359     double axis_dash_dash_dash_dash_dash_number_opacity(const Axis& axis);
360
361     double axis_dash_dash_dash_dash_dash_major_grid_line_deviation(const Axis& axis);
362
363     double axis_dash_dash_dash_dash_dash_major_grid_line_line_width(const Axis& axis);
364
365     double axis_dash_dash_dash_dash_dash_major_grid_line_opacity(const Axis& axis);
366
367     double axis_dash_dash_dash_dash_dash_medium_grid_line_deviation(const Axis& axis);
368
369     double axis_dash_dash_dash_dash_dash_medium_grid_line_line_width(const Axis& axis);
370
371     double axis_dash_dash_dash_dash_dash_medium_grid_line_opacity(const Axis& axis);
372
373     double axis_dash_dash_dash_dash_dash_small_grid_line_deviation(const Axis& axis);
374
375     double axis_dash_dash_dash_dash_dash_small_grid_line_line_width(const Axis& axis);
376
377     double axis_dash_dash_dash_dash_dash_small_grid_line_opacity(const Axis& axis);
378
379     double axis_dash_dash_dash_dash_dash_line_deviation(const Axis& axis);
380
381     double axis_dash_dash_dash_dash_dash_line_line_width(const Axis& axis);
382
383     double axis_dash_dash_dash_dash_dash_line_opacity(const Axis& axis);
384
385     double axis_dash_dash_dash_dash_dash_dash_deviation(const Axis& axis);
386
387     double axis_dash_dash_dash_dash_dash_dash_line_width(const Axis& axis);
388
389     double axis_dash_dash_dash_dash_dash_dash_opacity(const Axis& axis);
390
391     double axis_dash_dash_dash_dash_dash_dash_aspect_ratio(const Axis& axis);
392
393     double axis_dash_dash_dash_dash_dash_dash_label_deviation(const Axis& axis);
394
395     double axis_dash_dash_dash_dash_dash_dash_label_line_width(const Axis& axis);
396
397     double axis_dash_dash_dash_dash_dash_dash_label_opacity(const Axis& axis);
398
399     double axis_dash_dash_dash_dash_dash_dash_number_deviation(const Axis& axis);
400
401     double axis_dash_dash_dash_dash_dash_dash_number_line_width(const Axis& axis);
402
403     double axis_dash_dash_dash_dash_dash_dash_number_opacity(const Axis& axis);
404
405     double axis_dash_dash_dash_dash_dash_dash_major_grid_line_deviation(const Axis& axis);
406
407     double axis_dash_dash_dash_dash_dash_dash_major_grid_line_line_width(const Axis& axis);
408
409     double axis_dash_dash_dash_dash_dash_dash_major_grid_line_opacity(const Axis& axis);
410
411     double axis_dash_dash_dash_dash_dash_dash_medium_grid_line_deviation(const Axis& axis);
412
413     double axis_dash_dash_dash_dash_dash_dash_medium_grid_line_line_width(const Axis& axis);
414
415     double axis_dash_dash_dash_dash_dash_dash_medium_grid_line_opacity(const Axis& axis);
416
417     double axis_dash_dash_dash_dash_dash_dash_small_grid_line_deviation(const Axis& axis);
418
419     double axis_dash_dash_dash_dash_dash_dash_small_grid_line_line_width(const Axis& axis);
420
421     double axis_dash_dash_dash_dash_dash_dash_small_grid_line_opacity(const Axis& axis);
422
423     double axis_dash_dash_dash_dash_dash_dash_line_deviation(const Axis& axis);
424
425     double axis_dash_dash_dash_dash_dash_dash_line_line_width(const Axis& axis);
426
427     double axis_dash_dash_dash_dash_dash_dash_line_opacity(const Axis& axis);
428
429     double axis_dash_dash_dash_dash_dash_dash_dash_deviation(const Axis& axis);
430
431     double axis_dash_dash_dash_dash_dash_dash_dash_line_width(const Axis& axis);
432
433     double axis_dash_dash_dash_dash_dash_dash_dash_opacity(const Axis& axis);
434
435     double axis_dash_dash_dash_dash_dash_dash_dash_aspect_ratio(const Axis& axis);
436
437     double axis_dash_dash_dash_dash_dash_dash_dash_label_deviation(const Axis& axis);
438
439     double axis_dash_dash_dash_dash_dash_dash_dash_label_line_width(const Axis& axis);
440
441     double axis_dash_dash_dash_dash_dash_dash_dash_label_opacity(const Axis& axis);
442
443     double axis_dash_dash_dash_dash_dash_dash_dash_number_deviation(const Axis& axis);
444
445     double axis_dash_dash_dash_dash_dash_dash_dash_number_line_width(const Axis& axis);
446
447     double axis_dash_dash_dash_dash_dash_dash_dash_number_opacity(const Axis& axis);
448
449     double axis_dash_dash_dash_dash_dash_dash_dash_major_grid_line_deviation(const Axis& axis);
450
451     double axis_dash_dash_dash_dash_dash_dash_dash_major_grid_line_line_width(const Axis& axis);
452
453     double axis_dash_dash_dash_dash_dash_dash_dash_major_grid_line_opacity(const Axis& axis);
454
455     double axis_dash_dash_dash_dash_dash_dash_dash_medium_grid_line_deviation(const Axis& axis);
456
457     double axis_dash_dash_dash_dash_dash_dash_dash_medium_grid_line_line_width(const Axis& axis);
458
459     double axis_dash_dash_dash_dash_dash_dash_dash_medium_grid_line_opacity(const Axis& axis);
460
461     double axis_dash_dash_dash_dash_dash_dash_dash_small_grid_line_deviation(const Axis& axis);
462
463     double axis_dash_dash_dash_dash_dash_dash_dash_small_grid_line_line_width(const Axis& axis);
464
465     double axis_dash_dash_dash_dash_dash_dash_dash_small_grid_line_opacity(const Axis& axis);
466
467     double axis_dash_dash_dash_dash_dash_dash_dash_line_deviation(const Axis& axis);
468
469     double axis_dash_dash_dash_dash_dash_dash_dash_line_line_width(const Axis& axis);
470
471     double axis_dash_dash_dash_dash_dash_dash_dash_line_opacity(const Axis& axis);
472
473     double axis_dash_dash_dash_dash_dash_dash_dash_dash_deviation(const Axis& axis);
474
475     double axis_dash_dash_dash_dash_dash_dash_dash_dash_line_width(const Axis& axis);
476
477     double axis_dash_dash_dash_dash_dash_dash_dash_dash_opacity(const Axis& axis);
478
479     double axis_dash_dash_dash_dash_dash_dash_dash_dash_aspect_ratio(const Axis& axis);
480
481     double axis_dash_dash_dash_dash_dash_dash_dash_dash_label_deviation(const Axis& axis);
482
483     double axis_dash_dash_dash_dash_dash_dash_dash_dash_label_line_width(const Axis& axis);
484
485     double axis_dash_dash_dash_dash_dash_dash_dash_dash_label_opacity(const Axis& axis);
486
487     double axis_dash_dash_dash_dash_dash_dash_dash_dash_number_deviation(const Axis& axis);
488
489     double axis_dash_dash_dash_dash_dash_dash_dash_dash_number_line_width(const Axis& axis);
490
491     double axis_dash_dash_dash_dash_dash_dash_dash_dash_number_opacity(const Axis& axis);
492
493     double axis_dash_dash_dash_dash_dash_dash_dash_dash_major_grid_line_deviation(const Axis& axis);
494
495     double axis_dash_dash_dash_dash_dash_dash_dash_dash_major_grid_line_line_width(const Axis& axis);
496
497     double axis_dash_dash_dash_dash_dash_dash_dash_dash_major_grid_line_opacity(const Axis& axis);
498
499     double axis_dash_dash_dash_dash_dash_dash_dash_dash_medium_grid_line_deviation(const Axis& axis);
500
501     double axis_dash_dash_dash_dash_dash_dash_dash_dash_medium_grid_line_line_width(const Axis& axis);
502
503     double axis_dash_dash_dash_dash_dash_dash_dash_dash_medium_grid_line_opacity(const Axis& axis);
504
505     double axis_dash_dash_dash_dash_dash_dash_dash_dash_small_grid_line_deviation(const Axis& axis);
506
507     double axis_dash_dash_dash_dash_dash_dash_dash_dash_small_grid_line_line_width(const Axis& axis);
508
509     double axis_dash_dash_dash_dash_dash_dash_dash_dash_small_grid_line_opacity(const Axis& axis);
509

```

```

266     double medium_grid_line_line_width_divisor;
267
272     double medium_grid_line_opacity;
273
278     bool show_small_grid_lines;
279
284     std::vector<double> small_grid_lines_dash_pattern;
285
291     double small_grid_line_line_width_divisor;
292
297     double small_grid_line_opacity;
298
299     void set_max_value(const double value);
300     void reset_max_value();
301     void set_min_value(const double value);
302     void reset_min_value();
303
304     void update_max_min(const double max_value, const double min_value);
305
306     void calculate_transformations();
307
308     double get_min_value() const;
309     double get_max_value() const;
310
316     std::function<double(double)> axis_transform() const;
317
318     std::function<double(double)> axis_transform_inverse() const;
319
320     Scene2d render_to_scene() const;
321
322
323     Axis(const AxisType axis_type, const std::string& label = "",
324          const AxisScale axis_scale = AxisScale::LINEAR,
325          const double scale = 1,
326          const bool visible = true, const bool inverted = false,
327          const double position = 0, const Color& color = Color::BLACK,
328          const double line_width = LineWidth::THIN,
329          const double opacity = 1, const std::vector<double>& dash_pattern = DashPatterns::SOLID,
330          const double aspect_ratio = 0.6,
331          const double arrow_head_length = 0.3,
332          const double arrow_head_width = 0.75,
333          const double arrow_length = 0.5,
334          std::function<Objects2d::Object2d(const AffineSpace::Point2d& start,
335          const AffineSpace::Point2d& end,
336          const double arrow_head_length, const double arrow_head_width,
337          const Color& color, const double opacity, const double line_width,
338          const std::vector<double>& dash_pattern)> arrow =
339              [] (const AffineSpace::Point2d& start,
340                 const AffineSpace::Point2d& end,
341                 const double arrow_head_length, const double arrow_head_width,
342                 const Color& color, const double opacity, const double line_width,
343                 const std::vector<double>& dash_pattern)
344                 {return Objects2d::Arrow(start, end, arrow_head_length, arrow_head_width, color,
345                 opacity, line_width, dash_pattern);},
346          const unsigned int N_major_ticks = 9,
347          const double major_tick_deviation = 0.25,
348          const double major_tick_line_width_divisor = 3,
349          const bool show_medium_ticks = true,
350          const double medium_tick_deviation = 0.20,
351          const double medium_tick_line_width_divisor = 4,
352          const bool show_small_ticks = true,
353          const double small_tick_deviation = 0.15,
354          const double small_tick_line_width_divisor = 5,
355          const bool show_numbers = true,
356          const bool show_major_grid_lines = false,
357          const std::vector<double>& major_grid_lines_dash_pattern = DashPatterns::SOLID,
358          const double major_grid_line_line_width_divisor = 2,
359          const double major_grid_line_opacity = 0.75,
360          const bool show_medium_grid_lines = false,
361          const std::vector<double>& medium_grid_lines_dash_pattern = DashPatterns::SOLID,
362          const double medium_grid_line_line_width_divisor = 4,
363          const double medium_grid_line_opacity = 0.5,
364          const bool show_small_grid_lines = false,
365          const std::vector<double>& small_grid_lines_dash_pattern = DashPatterns::SOLID,
366          const double small_grid_line_line_width_divisor = 8,
367          const double small_grid_line_opacity = 0.25);
368
369     protected:
370     bool user_defined_max_value;
371     double max_value;
372     bool user_defined_min_value;
373     double min_value;
374
375     double x_max;
376
377     // Transformation parameters.
378     double _a;

```



```

379         double _b;
380
381         int digit_max;
382         unsigned int precision;
383
384     public:
385         // Predefined constants.
386
387         constexpr static const double X_MAX = 15;
388
389         constexpr static const double X_MIN = 0;
390
391         constexpr static const double Y_MIN = 0;
392
393         constexpr static const double NUMBER_DISPLACEMENT = 0.5;
394
395         constexpr static const double LABEL_DISPLACEMENT_HORIZONTAL = 1.2;
396
397         constexpr static const double LABEL_DISPLACEMENT_VERTICAL = 2;
398     };
399 }
400 #endif // AXIS_HPP

```

8.12 DataPlot.hpp

```

1 #ifndef DATA_PLOT_HPP
2 #define DATA_PLOT_HPP
3
4 #include "../AffineSpace2d/Point2d.hpp"
5 #include "../Objects2d/BasicGeometries.hpp"
6 #include "GraphicObject.hpp"
7 #include <functional>
8
9 namespace CPGF
10 {
11     namespace Plot2d
12     {
13         class Shapes
14         {
15         public:
16             static Objects2d::Object2d Circle(const AffineSpace::Point2d& pos,
17                 const Color& color, const double opacity,
18                 const double size);
19             static Objects2d::Object2d Square(const AffineSpace::Point2d& pos,
20                 const Color& color, const double opacity,
21                 const double size);
22         };
23
24         class DataPlot: public GraphicObject
25         {
26         public:
27             std::vector<AffineSpace::Point2d> points;
28             std::function<Color(unsigned int)> color;
29             std::function<double(unsigned int)> size;
30             std::function<double(unsigned int)> opacity;
31             std::function<std::function<Objects2d::Object2d(const AffineSpace::Point2d pos,
32                 const Color& color, const double opacity, const double size)>
33                 (unsigned int)> shape;
34
35             DataPlot(const std::vector<double>& Y, const std::vector<double>& X,
36                 const Color& color = Color::BLUE, const double size = 1,
37                 const double opacity = 1, std::function<Objects2d::Object2d(
38                     const AffineSpace::Point2d pos,
39                     const Color& color, const double opacity, const double size)>
40                     shape = Shapes::Circle,
41                     const std::string& legend = "");
42
43             DataPlot(const std::vector<double>& Y, const std::vector<double>& X,
44                 std::function<Color(unsigned int)> color,
45                 std::function<double(unsigned int)> size,
46                 std::function<double(unsigned int)> opacity,
47                 std::function<std::function<Objects2d::Object2d(
48                     const AffineSpace::Point2d pos,
49                     const Color& color, const double opacity, const double size)>
50                     (unsigned int)> shape,
51                     const std::string& legend = "");
52
53             DataPlot(const std::vector<double>& Y, const std::vector<double>& X,
54                 std::function<Color(AffineSpace::Point2d)> color,
55                 std::function<double(AffineSpace::Point2d)> size,
56                 std::function<double(AffineSpace::Point2d)> opacity,
57

```

```

58         std::function<std::function<Objects2d::Object2d(
59             const AffineSpace::Point2d pos,
60             const Color& color, const double opacity, const double size)>
61             (AffineSpace::Point2d&)> shape,
62             const std::string& legend = "");
63
64     DataPlot(const std::vector<AffineSpace::Point2d>& data,
65             const Color& color = Color::BLUE, const double size = 1,
66             const double opacity = 1, std::function<Objects2d::Object2d(
67                 const AffineSpace::Point2d pos, const Color& color,
68                 const double opacity, const double size)> shape =
69                 Shapes::Circle,
70             const std::string& legend = "");
71
72     DataPlot(const std::vector<AffineSpace::Point2d>& data,
73             std::function<Color(unsigned int)> color,
74             std::function<double(unsigned int)> size,
75             std::function<double(unsigned int)> opacity,
76             std::function<std::function<Objects2d::Object2d(const AffineSpace::Point2d pos,
77                 const Color& color, const double opacity, const double size)>
78                 (unsigned int)> shape,
79             const std::string& legend = "");
80
81     DataPlot(const std::vector<AffineSpace::Point2d>& data,
82             std::function<Color(AffineSpace::Point2d)> color,
83             std::function<double(AffineSpace::Point2d)> size,
84             std::function<double(AffineSpace::Point2d)> opacity,
85             std::function<std::function<Objects2d::Object2d(const AffineSpace::Point2d pos,
86                 const Color& color, const double opacity, const double size)>
87                 (AffineSpace::Point2d)> shape,
88             const std::string& legend = "");
89
90     double x_min() const override;
91     double x_max() const override;
92     double y_min() const override;
93     double y_max() const override;
94
95     Objects2d::Object2d miniature(const AffineSpace::Point2d& pos) const override;
96
97     Scene2d render_to_scene(
98         std::function<AffineSpace::Point2d(const AffineSpace::Point2d& P)> transform,
99         const double x_min, const double x_max, const double y_min, const double y_max) const
100     override;
101     };
102 }
103
104 #endif // DATA_PLOT_HPP

```

8.13 Graphic.hpp

```

1  #ifndef GRAPHIC_HPP
2  #define GRAPHIC_HPP
3
4  #include <vector>
5  #include "GraphicObject.hpp"
6  #include "../Scene2d.hpp"
7  #include "Axis.hpp"
8
9  namespace CPGF
10 {
11     namespace Plot2d
12     {
13         enum class LegendPosition
14         {
15             LEFT,
16             ABOVE,
17             RIGHT,
18             BELOW,
19         };
20
21         class Graphic
22         {
23         public:
24             std::vector<std::tuple<GraphicObject*, Axis*, Axis*>> graphic_objects;
25
26             bool show_legend;
27
28             LegendPosition legend_position;
29
30             void add(GraphicObject* object, Axis* Y, Axis* X);
31
32             Scene2d render_to_scene() const;

```

```

33
34         Graphic(const bool show_legend = false);
35
36         std::vector<Axis*> axes;
37
38         constexpr static const double LEGEND_MARGIN = 0.25;
39         constexpr static const double LEGEND_VERTICAL_DISPLACEMENT_PER_LINE = 0.25;
40         constexpr static const double CHARACTER_WIDTH = 0.1;
41     };
42 }
43 }
44
45 #endif // GRAPHIC_HPP

```

8.14 GraphicObject.hpp

```

1 #ifndef GRAPHICOBJECT_HPP
2 #define GRAPHICOBJECT_HPP
3
4 #include "../Scene2d.hpp"
5 #include "../AffineSpace2d/Point2d.hpp"
6 #include <functional>
7
8 namespace CPGF
9 {
10     namespace Plot2d
11     {
12         class GraphicObject
13         {
14         public:
15             virtual double x_min() const = 0;
16             virtual double x_max() const = 0;
17             virtual double y_min() const = 0;
18             virtual double y_max() const = 0;
19
20             virtual Objects2d::Object2d miniature(const AffineSpace::Point2d& pos) const = 0;
21
22             virtual Scene2d render_to_scene(
23                 std::function<AffineSpace::Point2d(const AffineSpace::Point2d& P)> transform,
24                 const double x_min, const double x_max, const double y_min, const double y_max) const =
25                 0;
26
27             std::string legend;
28
29             constexpr static const double MINIATURE_HALF_WIDTH = 0.5;
30         };
31     }
32 }
33 #endif // GRAPHICOBJECT_HPP

```

8.15 LinePlot.hpp

```

1 #ifndef LINE_PLOT_HPP
2 #define LINE_PLOT_HPP
3
4 #include "../AffineSpace2d/Point2d.hpp"
5 #include "../Objects2d/BasicGeometries.hpp"
6 #include "GraphicObject.hpp"
7 #include <functional>
8
9 namespace CPGF
10 {
11     namespace Plot2d
12     {
13         class LinePlot: public GraphicObject
14         {
15         public:
16             std::vector<AffineSpace::Point2d> points;
17             std::function<Color(unsigned int)> color;
18             std::function<double(unsigned int)> line_width;
19             std::function<double(unsigned int)> opacity;
20             std::function<std::vector<double>(unsigned int)> dash_pattern;
21
22             LinePlot(const std::vector<double>& Y, const std::vector<double>& X,
23                 const Color& color = Color::BLUE, const double line_width = LineWidth::THIN,
24                 const double opacity = 1, const std::vector<double>& dash_pattern = DashPatterns::SOLID,
25                 const std::string& legend = "");
26
27             LinePlot(const std::vector<double>& Y, const std::vector<double>& X,

```

```

28         std::function<Color(unsigned int)> color, std::function<double(unsigned int)> line_width,
29         std::function<double(unsigned int)> opacity,
30         std::function<std::vector<double>(unsigned int)> dash_pattern,
31         const std::string& legend = "");
32
33     LinePlot(const std::vector<double>& Y, const std::vector<double>& X,
34             std::function<Color(AffineSpace::Point2d)> color,
35             std::function<double(AffineSpace::Point2d)> line_width,
36             std::function<double(AffineSpace::Point2d)> opacity,
37             std::function<std::vector<double>(AffineSpace::Point2d)> dash_pattern,
38             const std::string& legend = "");
39
40     LinePlot(const std::vector<AffineSpace::Point2d>& data,
41             const Color& color = Color::BLUE, const double line_width = LineWidth::THIN,
42             const double opacity = 1, const std::vector<double>& dash_pattern = DashPatterns::SOLID,
43             const std::string& legend = "");
44
45     LinePlot(const std::vector<AffineSpace::Point2d>& data,
46             std::function<Color(unsigned int)> color, std::function<double(unsigned int)> line_width,
47             std::function<double(unsigned int)> opacity,
48             std::function<std::vector<double>(unsigned int)> dash_pattern,
49             const std::string& legend = "");
50
51     LinePlot(const std::vector<AffineSpace::Point2d>& data,
52             std::function<Color(AffineSpace::Point2d)> color,
53             std::function<double(AffineSpace::Point2d)> line_width,
54             std::function<double(AffineSpace::Point2d)> opacity,
55             std::function<std::vector<double>(AffineSpace::Point2d)> dash_pattern,
56             const std::string& legend = "");
57
58     double x_min() const override;
59     double x_max() const override;
60     double y_min() const override;
61     double y_max() const override;
62
63     Objects2d::Object2d miniature(const AffineSpace::Point2d& pos) const override;
64
65     Scene2d render_to_scene(
66         std::function<AffineSpace::Point2d(const AffineSpace::Point2d& P)> transform,
67         const double x_min, const double x_max, const double y_min, const double y_max) const
68     override;
69
70     protected:
71     bool const_parameters;
72 };
73
74 class AveragePlot: public LinePlot
75 {
76     public:
77     AveragePlot(const std::vector<double>& Y, const std::vector<double>& partition,
78             const Color& color = Color::BLUE, const double line_width = LineWidth::THIN,
79             const double opacity = 1, const std::vector<double>& dash_pattern = DashPatterns::SOLID,
80             const std::string& legend = "");
81
82     AveragePlot(const std::vector<double>& Y, const std::vector<double>& partition,
83             std::function<Color(unsigned int)> color, std::function<double(unsigned int)> line_width,
84             std::function<double(unsigned int)> opacity,
85             std::function<std::vector<double>(unsigned int)> dash_pattern,
86             const std::string& legend = "");
87
88     AveragePlot(const std::vector<double>& Y, const std::vector<double>& partition,
89             std::function<Color(AffineSpace::Point2d)> color,
90             std::function<double(AffineSpace::Point2d)> line_width,
91             std::function<double(AffineSpace::Point2d)> opacity,
92             std::function<std::vector<double>(AffineSpace::Point2d)> dash_pattern,
93             const std::string& legend = "");
94
95     protected:
96     static LinePlot builder(const std::vector<double>& Y, const std::vector<double>& partition,
97             const Color& color, const double line_width,
98             const double opacity, const std::vector<double>& dash_pattern,
99             const std::string& legend);
100
101     static LinePlot builder(const std::vector<double>& Y, const std::vector<double>& partition,
102             std::function<Color(unsigned int)> color, std::function<double(unsigned int)>
103             line_width,
104             std::function<double(unsigned int)> opacity,
105             std::function<std::vector<double>(unsigned int)> dash_pattern,
106             const std::string& legend);
107
108     static LinePlot builder(const std::vector<double>& Y, const std::vector<double>& partition,
109             std::function<Color(AffineSpace::Point2d)> color,
110             std::function<double(AffineSpace::Point2d)> line_width,
111             std::function<double(AffineSpace::Point2d)> opacity,
112             std::function<std::vector<double>(AffineSpace::Point2d)> dash_pattern,
113             const std::string& legend);
114
115     static LinePlot builder(const std::vector<double>& Y, const std::vector<double>& partition,
116             std::function<Color(AffineSpace::Point2d)> color,
117             std::function<double(AffineSpace::Point2d)> line_width,
118             std::function<double(AffineSpace::Point2d)> opacity,
119             std::function<std::vector<double>(AffineSpace::Point2d)> dash_pattern,
120             const std::string& legend);
121 };

```

```

121     }
122 }
123
124 #endif // LINE_PLOT_HPP

```

8.16 Scene2d.hpp

```

1 #ifndef SCENE2D_HPP
2 #define SCENE2D_HPP
3
4 #include <vector>
5 #include <string>
6
7 #include "Objects2d/Object2d.hpp"
8 #include "Text/Text.hpp"
9
10 namespace CPGF
11 {
12     class Scene2d
13     {
14     public:
15         std::vector<Objects2d::Object2d*> objects;
16         std::vector<Text*> texts;
17
18         Scene2d(const std::vector<Objects2d::Object2d*>& objects = std::vector<Objects2d::Object2d*>(),
19                 const std::vector<Text*>& texts = std::vector<Text*>());
20
21         Scene2d& add(Objects2d::Object2d& object);
22         Scene2d& add(Text& text);
23
24         Scene2d friend operator+(const Scene2d& S1, const Scene2d& S2);
25         Scene2d& operator+=(const Scene2d& S2);
26         Scene2d& operator+=(Objects2d::Object2d& Obj);
27         Scene2d& operator+=(Text& text);
28
29         std::string render_to_string() const;
30         void render(const std::string& filename, const unsigned int density = 100) const;
31     };
32 }
33
34 #endif // SCENE2D_HPP

```

8.17 Text.hpp

```

1 #ifndef TEXT_HPP
2 #define TEXT_HPP
3
4 #include "../AffineSpace2d/Point2d.hpp"
5 #include "../PGFBasics/PGFConf.hpp"
6
7 namespace CPGF
8 {
9     enum class TextAlignment
10     {
11         CENTER,
12
13         LEFT,
14
15         RIGHT,
16
17         TOP,
18
19         BOTTOM,
20
21         BASE,
22
23         TOP_LEFT,
24
25         TOP_RIGHT,
26
27         BOTTOM_LEFT,
28
29         BOTTOM_RIGHT,
30
31         BASE_LEFT,
32
33         BASE_RIGHT,
34     };
35 }
36
37 #endif

```

```

96     class Text
97     {
98     public:
103         std::string text;
104
109         AffineSpace::Point2d pos;
110
115         double rot;
116
121         TextAlignment text_alignment;
122
127         Color color;
128
129         Text(const AffineSpace::Point2d& pos = AffineSpace::Point2d(0,0),
130              const std::string& text = "",
131              const Color& color = Color::BLACK,
132              const TextAlignment text_alignment = TextAlignment::CENTER,
133              const double rot = 0);
134
135         std::string render_to_string() const;
136     };
137 }
138
139 #endif // TEXT_HPP

```

8.18 Math/AlgebraicSolvers.hpp File Reference

This file contains functions that can be used to solve non-linear algebraic equations.

```

#include <functional>
#include "DualNumbers.hpp"
#include "Vector.hpp"
#include "Matrix.hpp"

```

Classes

- class [Math::AlgebraicSolvers::LinearSystemSolver](#)
Solves linear systems of the form $Ax=b$ through LU decomposition with partial pivoting.

Namespaces

- namespace [Math](#)
Namespace that includes mathematical objects such as vectors, matrixes, dual numbers and useful numerical methods such as numerical integrators, ode solvers and solvers for algebraic equations.
- namespace [Math::AlgebraicSolvers](#)
Contains solvers for algebraic equations.

Functions

- double [Math::AlgebraicSolvers::bisection](#) (std::function< double(const double x)> f, double a, double b, const unsigned int iter_max, const double abs_tol, const double rel_tol)
Applies the bisection method to the function f in the interval (a,b).
- double [Math::AlgebraicSolvers::secant](#) (std::function< double(const double x)> f, double x1, double x2, const unsigned int iter_max, const double abs_tol, const double rel_tol)
Applies the secant method to the function f with initial guesses x1 and x2.
- double [Math::AlgebraicSolvers::newton_raphson](#) (std::function< double(const double x)>f, std::function< double(const double x)>dfdx, const double x0, const unsigned int iter_max, const double abs_tol, const double rel_tol)
Applies the Newton-Raphson method to the function f with initial guess x0.
- double [Math::AlgebraicSolvers::newton_raphson](#) (std::function< [DualNumber](#)< double >(const [DualNumber](#)< double > x)>f, const double x0, const unsigned int iter_max, const double abs_tol, const double rel_tol)
Applies the Newton-Raphson method to the function f with initial guess x0.

8.18.1 Detailed Description

This file contains functions that can be used to solve non-linear algebraic equations.

Author

Andrés Laín Sanclemente

Version

0.2.0

Date

9th September 2021

8.19 AlgebraicSolvers.hpp

[Go to the documentation of this file.](#)

```

1
10 #ifndef ALGEBRAIC_SOLVERS_HPP
11 #define ALGEBRAIC_SOLVERS_HPP
12
13 #include <functional>
14 #include "DualNumbers.hpp"
15 #include "Vector.hpp"
16 #include "Matrix.hpp"
17
18 namespace Math
19 {
20     namespace AlgebraicSolvers
21     {
22         double bisection(std::function<double(const double x)> f,
23             double a, double b,
24             const unsigned int iter_max, const double abs_tol, const double rel_tol);
25
26         double secant(std::function<double(const double x)> f,
27             double x1, double x2,
28             const unsigned int iter_max, const double abs_tol, const double rel_tol);
29
30         double newton_raphson(std::function<double(const double x)>f,
31             std::function<double(const double x)>dfdx, const double x0,
32             const unsigned int iter_max, const double abs_tol, const double rel_tol);
33
34         double newton_raphson(std::function<DualNumber<double>(const DualNumber<double> x)>f,
35             const double x0,
36             const unsigned int iter_max, const double abs_tol, const double rel_tol);
37
38         class LinearSystemSolver
39         {
40         public:
41             Vector<double> solve(const Vector<double>& b) const;
42
43             explicit LinearSystemSolver(const Matrix<double>& A);
44
45         protected:
46
47             Matrix<double> L;
48
49             Matrix<double> U;
50
51             std::vector<unsigned int> P;
52         };
53     }
54 }
55 #endif // ALGEBRAIC_SOLVERS_HPP

```

8.20 Math/DualNumbers.hpp File Reference

This file implements dual numbers in one variable to allow for automatic differentiation.

```
#include <string>
```

Classes

- class [Math::DualNumber< K >](#)

Represents a dual number. A mathematical object written like $x = a + b$, where $b^2 = 0$.

Namespaces

- namespace [Math](#)

Namespace that includes mathematical objects such as vectors, matrixes, dual numbers and useful numerical methods such as numerical integrators, ode solvers and solvers for algebraic equations.

Functions

- `template<typename K >`
`DualNumber< K > Math::operator+ (const DualNumber< K > &x, const DualNumber< K > &y)`
- `template<typename K >`
`DualNumber< K > Math::operator+ (const DualNumber< K > &x, const K y)`
- `template<typename K >`
`DualNumber< K > Math::operator+ (const K x, const DualNumber< K > &y)`
- `template<typename K >`
`DualNumber< K > Math::operator- (const DualNumber< K > &x, const DualNumber< K > &y)`
- `template<typename K >`
`DualNumber< K > Math::operator- (const DualNumber< K > &x, const K y)`
- `template<typename K >`
`DualNumber< K > Math::operator- (const K x, const DualNumber< K > &y)`
- `template<typename K >`
`DualNumber< K > Math::operator* (const DualNumber< K > &x, const DualNumber< K > &y)`
- `template<typename K >`
`DualNumber< K > Math::operator* (const DualNumber< K > &x, const K y)`
- `template<typename K >`
`DualNumber< K > Math::operator* (const K x, const DualNumber< K > &y)`
- `template<typename K >`
`DualNumber< K > Math::operator/ (const DualNumber< K > &x, const DualNumber< K > &y)`
- `template<typename K >`
`DualNumber< K > Math::operator/ (const DualNumber< K > &x, const K y)`
- `template<typename K >`
`DualNumber< K > Math::operator/ (const K x, const DualNumber< K > &y)`
- `template<typename K >`
`DualNumber< K > Math::cos (const DualNumber< K > &x)`
Returns $\cos(x.a) - x.b \cdot \sin(x.a)$
- `template<typename K >`
`DualNumber< K > Math::sin (const DualNumber< K > &x)`
Returns $\sin(x.a) + x.b \cdot \cos(x.a)$
- `template<typename K >`
`DualNumber< K > Math::tan (const DualNumber< K > &x)`

- Returns $\tan(x.a) + x.b/\cos(x.a)/\cos(x.a)*$*

 - `template<typename K >`
`DualNumber< K > Math::acos` (const `DualNumber< K > &x`)

*Returns $\arccos(x.a) - 1/\sqrt{1-x.a*x.a}*x.b*$*
- `template<typename K >`
`DualNumber< K > Math::asin` (const `DualNumber< K > &x`)

*Returns $\arcsin(x.a) + 1/\sqrt{1-x.a*x.a}*x.b*$*
- `template<typename K >`
`DualNumber< K > Math::atan` (const `DualNumber< K > &x`)

*Returns $\arctan(x.a) + 1/(1+x.a*x.a)*x.b*$*
- `template<typename K >`
`DualNumber< K > Math::cosh` (const `DualNumber< K > &x`)

*Returns $\cosh(x.a) + \sinh(x.a)*x.b*$*
- `template<typename K >`
`DualNumber< K > Math::sinh` (const `DualNumber< K > &x`)

*Returns $\sinh(x.a) + \cosh(x.a)*x.b*$*
- `template<typename K >`
`DualNumber< K > Math::tanh` (const `DualNumber< K > &x`)

*Returns $\tanh(x.a) + 1/\cosh(x.a)/\cosh(x.a)*x.b*$*
- `template<typename K >`
`DualNumber< K > Math::acosh` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > Math::asinh` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > Math::atanh` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > Math::exp` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > Math::log` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > Math::log10` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > Math::exp2` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > Math::expm1` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > Math::log1p` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > Math::log2` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > Math::logb` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > Math::pow` (const `DualNumber< K > &x`, const `DualNumber< K > &y`)
- `template<typename K >`
`DualNumber< K > Math::pow` (const `DualNumber< K > &x`, const `K &y`)
- `template<typename K >`
`DualNumber< K > Math::pow` (const `K &x`, const `DualNumber< K > &y`)
- `template<typename K >`
`DualNumber< K > Math::sqrt` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > Math::cbrt` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > Math::erf` (const `DualNumber< K > &x`)
- `template<typename K >`
`DualNumber< K > Math::erfc` (const `DualNumber< K > &x`)

- `template<typename K >`
`DualNumber< K > Math::tgamma (const DualNumber< K > &x)`
- `template<typename K >`
`DualNumber< K > Math::lgamma (const DualNumber< K > &x)`
- `template<typename K >`
`DualNumber< K > Math::fabs (const DualNumber< K > &x)`

8.20.1 Detailed Description

This file implements dual numbers in one variable to allow for automatic differentiation.

Author

Andrés Laín Sanclemente

Version

0.2.0

Date

9th September 2021

8.21 DualNumbers.hpp

[Go to the documentation of this file.](#)

```

1
11 #ifndef DUALNUMBERS_HPP
12 #define DUALNUMBERS_HPP
13
14 #include <string>
15
16 namespace Math
17 {
18     // We make a forward declaration of the class in order to be able to declare
19     // friend operators.
20     template <typename K>
21     class DualNumber;
22
23     // Forward declaration of friend operators.
24     template <typename K>
25     DualNumber<K> operator+(const DualNumber<K>& x, const DualNumber<K>& y);
26     template <typename K>
27     DualNumber<K> operator+(const DualNumber<K>& x, const K y);
28     template <typename K>
29     DualNumber<K> operator+(const K x, const DualNumber<K>& y);
30     template <typename K>
31     DualNumber<K> operator-(const DualNumber<K>& x, const DualNumber<K>& y);
32     template <typename K>
33     DualNumber<K> operator-(const DualNumber<K>& x, const K y);
34     template <typename K>
35     DualNumber<K> operator-(const K x, const DualNumber<K>& y);
36     template <typename K>
37     DualNumber<K> operator*(const DualNumber<K>& x, const DualNumber<K>& y);
38     template <typename K>
39     DualNumber<K> operator*(const DualNumber<K>& x, const K y);
40     template <typename K>
41     DualNumber<K> operator*(const K x, const DualNumber<K>& y);
42     template <typename K>
43     DualNumber<K> operator/(const DualNumber<K>& x, const DualNumber<K>& y);
44     template <typename K>
45     DualNumber<K> operator/(const DualNumber<K>& x, const K y);
46     template <typename K>
47     DualNumber<K> operator/(const K x, const DualNumber<K>& y);
48
49     template <typename K>

```

```

59     class DualNumber
60     {
61     public:
62         K a;
63
64         K b;
65
66         static const DualNumber<K> epsilon;
67
68         DualNumber();
69
70         DualNumber(const K a);
71
72         DualNumber(const K a, const K b);
73
74         friend DualNumber<K> operator+<K>(const DualNumber<K>& x, const DualNumber<K>& y);
75
76         friend DualNumber<K> operator+<K>(const DualNumber<K>& x, const K y);
77
78         friend DualNumber<K> operator+<K>(const K x, const DualNumber<K>& y);
79
80         friend DualNumber<K> operator-<K>(const DualNumber<K>& x, const DualNumber<K>& y);
81
82         friend DualNumber<K> operator-<K>(const DualNumber<K>& x, const K y);
83
84         friend DualNumber<K> operator-<K>(const K x, const DualNumber<K>& y);
85
86         friend DualNumber<K> operator*<K>(const DualNumber<K>& x, const DualNumber<K>& y);
87
88         friend DualNumber<K> operator*<K>(const DualNumber<K>& x, const K y);
89
90         friend DualNumber<K> operator*<K>(const K x, const DualNumber<K>& y);
91
92         friend DualNumber<K> operator/<K>(const DualNumber<K>& x, const DualNumber<K>& y);
93
94         friend DualNumber<K> operator/<K>(const DualNumber<K>& x, const K y);
95
96         friend DualNumber<K> operator/<K>(const K x, const DualNumber<K>& y);
97
98         std::string to_string() const;
99     };
100
101     template <typename K>
102     DualNumber<K> cos(const DualNumber<K>& x);
103
104     template <typename K>
105     DualNumber<K> sin(const DualNumber<K>& x);
106
107     template <typename K>
108     DualNumber<K> tan(const DualNumber<K>& x);
109
110     template <typename K>
111     DualNumber<K> acos(const DualNumber<K>& x);
112
113     template <typename K>
114     DualNumber<K> asin(const DualNumber<K>& x);
115
116     template <typename K>
117     DualNumber<K> atan(const DualNumber<K>& x);
118
119     template <typename K>
120     DualNumber<K> cosh(const DualNumber<K>& x);
121
122     template <typename K>
123     DualNumber<K> sinh(const DualNumber<K>& x);
124
125     template <typename K>
126     DualNumber<K> tanh(const DualNumber<K>& x);
127
128     template <typename K>
129     DualNumber<K> acosh(const DualNumber<K>& x);
130
131     template <typename K>
132     DualNumber<K> asinh(const DualNumber<K>& x);
133
134     template <typename K>
135     DualNumber<K> atanh(const DualNumber<K>& x);
136
137     template <typename K>
138     DualNumber<K> exp(const DualNumber<K>& x);
139
140     template <typename K>
141     DualNumber<K> log(const DualNumber<K>& x);
142
143     template <typename K>
144     DualNumber<K> log10(const DualNumber<K>& x);
145
146     template <typename K>
147     DualNumber<K> exp2(const DualNumber<K>& x);
148
149     template <typename K>
150     DualNumber<K> expm1(const DualNumber<K>& x);
151
152     template <typename K>
153     DualNumber<K> log1p(const DualNumber<K>& x);
154
155     template <typename K>

```

```

338     DualNumber<K> log2(const DualNumber<K>& x);
339     template <typename K>
340     DualNumber<K> logb(const DualNumber<K>& x);
341     template <typename K>
342     DualNumber<K> pow(const DualNumber<K>& x, const DualNumber<K>& y);
343     template <typename K>
344     DualNumber<K> pow(const DualNumber<K>& x, const K& y);
345     template <typename K>
346     DualNumber<K> pow(const K& x, const DualNumber<K>& y);
347     template <typename K>
348     DualNumber<K> sqrt(const DualNumber<K>& x);
349     template <typename K>
350     DualNumber<K> cbrt(const DualNumber<K>& x);
351     template <typename K>
352     DualNumber<K> erf(const DualNumber<K>& x);
353     template <typename K>
354     DualNumber<K> erfc(const DualNumber<K>& x);
355     /* To do. Not implemented. */
356     template <typename K>
357     DualNumber<K> tgamma(const DualNumber<K>& x);
358     /* To do. Not implemented. */
359     template <typename K>
360     DualNumber<K> lgamma(const DualNumber<K>& x);
361     template <typename K>
362     DualNumber<K> fabs(const DualNumber<K>& x);
363 }
364
365 #endif // DUALNUMBERS_HPP

```

8.22 Integration.hpp

```

1 #ifndef INTEGRATION_HPP
2 #define INTEGRATION_HPP
3
4 #include <functional>
5
6 namespace Math
7 {
8     namespace Integrators
9     {
10         double rectangle_rule(std::function<double(double x)> f, const double a, const double b,
11                               const unsigned int N);
12
13         double trapezoidal_rule(std::function<double(double x)> f, const double a, const double b,
14                                 const unsigned int N);
15
16         double Gauss_Konrad_G7_K15(std::function<double(double x)> f, const double a, const double b,
17                                     const double tol = 1e-6);
18     }
19 }
20
21 #endif // INTEGRATION_HPP

```

8.23 Interpolation.hpp

```

1 #ifndef INTERPOLATION_HPP
2 #define INTERPOLATION_HPP
3
4 #include <vector>
5
6 namespace Math
7 {
8     namespace Interpolation
9     {
10         class AverageLinearInterpolation
11         {
12         public:
13
14             AverageLinearInterpolation(const std::vector<double>& averages,
15                                       const std::vector<double>& partition);
16             AverageLinearInterpolation(double* averages,
17                                       double *partition, unsigned int N_cells);
18
19             AverageLinearInterpolation(const AverageLinearInterpolation& f);
20             AverageLinearInterpolation& operator=(const AverageLinearInterpolation& f);
21             ~AverageLinearInterpolation();
22
23             double operator()(const double x) const;
24         };
25     }
26 }

```

```

33         public:
34             unsigned int N_cells;
35             double* x_part;
36             double* b;
37             double* c;
38     };
39
40     class AverageQuadraticInterpolation
41     {
42     public:
43
44         AverageQuadraticInterpolation(const std::vector<double>& averages,
45             const std::vector<double>& partition);
46         AverageQuadraticInterpolation(double* averages,
47             double *partition, unsigned int N_cells);
48
49         AverageQuadraticInterpolation(const AverageQuadraticInterpolation& f);
50         AverageQuadraticInterpolation& operator=(const AverageQuadraticInterpolation& f);
51         ~AverageQuadraticInterpolation();
52
53         double operator()(const double x) const;
54
55     protected:
56         unsigned int N_cells;
57         double* x_part;
58         double* a;
59         double* b;
60         double* c;
61     };
62
63     class LinearInterpolation
64     {
65     public:
66         LinearInterpolation(const std::vector<double>& Y, const std::vector<double>& X);
67         LinearInterpolation(const double* Y, const double* X, const unsigned int N);
68
69         LinearInterpolation(const LinearInterpolation& I);
70         LinearInterpolation& operator=(const LinearInterpolation& I);
71         ~LinearInterpolation();
72
73         double operator()(const double x) const;
74
75     protected:
76         unsigned int N;
77         double* x_part;
78         double* b;
79         double* c;
80     };
81 }
82
83 #endif // INTERPOLATION_HPP

```

8.24 Matrix.hpp

```

1  #ifndef MATRIX_HPP
2  #define MATRIX_HPP
3
4  #include "Vector.hpp"
5  #include <vector>
6
7  double inline fabs(std::complex<double> z)
8  {
9      return std::abs(z);
10 }
11
12 namespace Math
13 {
14     // Forward declaration to make function declaration possible.
15     template<typename K>
16     class Matrix;
17
18     // We declare all friend operators.
19     template <typename K>
20     Matrix<K> operator+(const Matrix<K>& A, const Matrix<K>& B);
21     template <typename K>
22     Matrix<K> operator+(const Matrix<K>& A, const K& alpha);
23     template <typename K>
24     Matrix<K> operator+( const K& alpha, const Matrix<K>& A);
25     template <typename K>
26     Matrix<K> operator-(const Matrix<K>& A, const Matrix<K>& B);
27     template <typename K>
28     Matrix<K> operator-(const Matrix<K>& A, const K& alpha);

```

```

29     template <typename K>
30     Matrix<K> operator-( const K& alpha, const Matrix<K>& A);
31     template <typename K>
32     Matrix<K> operator*(const Matrix<K>& A, const K& alpha);
33     template <typename K>
34     Matrix<K> operator*( const K& alpha, const Matrix<K>& A);
35     template <typename K>
36     Matrix<K> operator/(const Matrix<K>& A, const K& alpha);
37     template <typename K>
38     Matrix<K> operator|(const Matrix<K>& A, const Matrix<K>& B);
39     template <typename K>
40     Vector<K> operator|(const Matrix<K>& A, const Vector<K>& v);
41     template <typename K>
42     Vector<K> operator|(const Vector<K>& v, const Matrix<K>& A);
43
44     // In the future, define exp, log, sqrt, ...
45
46     template <typename K>
47     class Matrix
48     {
49     public:
50         Matrix(unsigned int m = 0, unsigned int n = 0);
51         Matrix(unsigned int m, unsigned int n, std::vector<K> elements);
52
53         Matrix(const Matrix<K>& B);
54         Matrix& operator=(const Matrix<K>& B);
55         ~Matrix();
56
57         operator Matrix<std::complex<K>>() const;
58
59         friend Matrix<K> operator+<K>(const Matrix<K>& A, const Matrix<K>& B);
60         friend Matrix<K> operator+<K>(const Matrix<K>& A, const K& alpha);
61         friend Matrix<K> operator+<K>(const K& alpha, const Matrix<K>& A);
62         friend Matrix<K> operator-<K>(const Matrix<K>& A, const Matrix<K>& B);
63         friend Matrix<K> operator-<K>(const Matrix<K>& A, const K& alpha);
64         friend Matrix<K> operator-<K>(const K& alpha, const Matrix<K>& A);
65         friend Matrix<K> operator*<K>(const Matrix<K>& A, const K& alpha);
66         friend Matrix<K> operator*<K>(const K& alpha, const Matrix<K>& A);
67         friend Matrix<K> operator/<K>(const Matrix<K>& A, const K& alpha);
68         friend Matrix<K> operator|<K>(const Matrix<K>& A, const Matrix<K>& B);
69         friend Vector<K> operator|<K>(const Matrix<K>& A, const Vector<K>& v);
70         friend Vector<K> operator|<K>(const Vector<K>& v, const Matrix<K>& A);
71
72         Matrix<K>& operator+=(const K& alpha);
73         Matrix<K>& operator-=(const K& alpha);
74         Matrix<K>& operator*=(const K& alpha);
75         Matrix<K>& operator/=(const K& alpha);
76
77         Vector<K>& operator() (const unsigned int i, const std::string& j);
78
79         Vector<K> operator() (const unsigned int i, const std::string& j) const;
80
81         Vector<K> operator() (const std::string& i, const unsigned int j) const;
82
83         K& operator() (const unsigned int i, const unsigned int j);
84         K operator() (const unsigned int i, const unsigned int j) const;
85
86         static Matrix<K> zero(const unsigned int m, const unsigned int n);
87         static Matrix<K> zero(const unsigned int n);
88         static Matrix<K> identity(const unsigned int n);
89
90         unsigned int m() const;
91         unsigned int n() const;
92
93         bool is_square() const;
94
95         void LU(Matrix<K>& L, Matrix<K>& U, std::vector<unsigned int>& P) const;
96
97         K det() const;
98
99         K tr() const;
100
101         std::vector<std::complex<double>> eigenvalues() const;
102
103         std::vector<Vector<std::complex<double>>> eigenvectors(
104             std::vector<std::complex<double>>& eigenvalues,
105             const bool calculate_eigenvalues = true) const;
106
107         std::string to_string() const;
108
109     protected:
110         Vector<K>* vec;
111         unsigned int _m;
112     };
113 }
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157 }
158

```

```
159 #endif // MATRIX_HPP
```

8.25 ODESolvers.hpp

```
1 #ifndef ODE_SOLVERS_HPP
2 #define ODE_SOLVERS_HPP
3
4 #include "Vector.hpp"
5 #include <functional>
6
7 namespace Math
8 {
9     namespace ODESolvers
10    {
11        Vector<double> euler_explicit(const Vector<double>& x, const double t,
12                                     std::function<Vector<double>(const Vector<double>&, const double)> f, const double dt);
13        Vector<double> euler_explicit(const Vector<double>& x,
14                                     std::function<Vector<double>(const Vector<double>&)> f, const double dt);
15
16        Vector<double> runge_kutta_order_4(const Vector<double>& x, const double t,
17                                           std::function<Vector<double>(const Vector<double>&, const double)> f, const double dt);
18        Vector<double> runge_kutta_order_4(const Vector<double>& x,
19                                           std::function<Vector<double>(const Vector<double>&)> f, const double dt);
20
21        void runge_kutta_fehlberg(Vector<double>& x, double& t, double& dt,
22                                  std::function<Vector<double>(const Vector<double>&, const double)> f,
23                                  const Vector<double>& abs_tol);
24        void runge_kutta_fehlberg(Vector<double>& x, double& t, double& dt,
25                                  std::function<Vector<double>(const Vector<double>&)> f,
26                                  const Vector<double>& abs_tol);
27    }
28 #endif // ODE_SOLVERS_HPP
```

8.26 Rational.hpp

```
1 #ifndef RATIONAL_HPP
2 #define RATIONAL_HPP
3
4 #include <string>
5
6
7 namespace Math
8 {
9     // Forward declaration to make function declaration possible.
10    template <typename K>
11    class Rational;
12
13    // And we now declare all friend operators.
14    template <typename K>
15    Rational<K>& operator+(const Rational<K>& p, const Rational<K>& q);
16    template <typename K>
17    Rational<K>& operator-(const Rational<K>& p, const Rational<K>& q);
18    template <typename K>
19    Rational<K>& operator*(const Rational<K>& p, const Rational<K>& q);
20    template <typename K>
21    Rational<K>& operator/(const Rational<K>& p, const Rational<K>& q);
22    template <typename K>
23    bool operator==(const Rational<K>& p, const Rational<K>& q);
24    template <typename K>
25    bool operator!=(const Rational<K>& p, const Rational<K>& q);
26    template <typename K>
27    bool operator<(const Rational<K>& p, const Rational<K>& q);
28    template <typename K>
29    bool operator<=(const Rational<K>& p, const Rational<K>& q);
30    template <typename K>
31    bool operator>(const Rational<K>& p, const Rational<K>& q);
32    template <typename K>
33    bool operator>=(const Rational<K>& p, const Rational<K>& q);
34
35
36    template <typename K>
37    class Rational
38    {
39    public:
40
41        K num;
42        K den;
```

```

45     Rational();
46     Rational(const K num);
47     Rational(const K num, const K den);
48
49     std::string to_string() const;
50
51     friend Rational<K>& operator+<K>(const Rational<K>& p, const Rational<K>& q);
52     friend Rational<K>& operator-<K>(const Rational<K>& p, const Rational<K>& q);
53     friend Rational<K>& operator*<K>(const Rational<K>& p, const Rational<K>& q);
54     friend Rational<K>& operator/<K>(const Rational<K>& p, const Rational<K>& q);
55     friend bool operator==<K>(const Rational<K>& p, const Rational<K>& q);
56     friend bool operator!=<K>(const Rational<K>& p, const Rational<K>& q);
57     friend bool operator<<K>(const Rational<K>& p, const Rational<K>& q);
58     friend bool operator<=<K>(const Rational<K>& p, const Rational<K>& q);
59     friend bool operator><K>(const Rational<K>& p, const Rational<K>& q);
60     friend bool operator>=<K>(const Rational<K>& p, const Rational<K>& q);
61 };
62 }
63
64
65 #endif // RATIONAL_HPP

```

8.27 Math/Vector.hpp File Reference

This file contains all prototypes related to class `Vector<K>`.

```

#include <string>
#include <complex>

```

Classes

- class `Math::Vector< K >`
Generic `Vector` over the field `K` with any number of components.

Namespaces

- namespace `Math`
Namespace that includes mathematical objects such as vectors, matrixes, dual numbers and useful numerical methods such as numerical integrators, ode solvers and solvers for algebraic equations.

Functions

- char `Math::conj` (const char &val)
This function exists to make the definition of the scalar product more general. It just returns `val`.
- unsigned char `Math::conj` (const unsigned char &val)
This function exists to make the definition of the scalar product more general. It just returns `val`.
- short int `Math::conj` (const short int &val)
This function exists to make the definition of the scalar product more general. It just returns `val`.
- unsigned short int `Math::conj` (const unsigned short int &val)
This function exists to make the definition of the scalar product more general. It just returns `val`.
- int `Math::conj` (const int &val)
This function exists to make the definition of the scalar product more general. It just returns `val`.
- unsigned int `Math::conj` (const unsigned int &val)
This function exists to make the definition of the scalar product more general. It just returns `val`.
- long int `Math::conj` (const long int &val)

- This function exists to make the definition of the scalar product more general. It just returns val .*
- unsigned long int [Math::conj](#) (const unsigned long int &val)
- This function exists to make the definition of the scalar product more general. It just returns val .*
- long long int [Math::conj](#) (const long long int &val)
- This function exists to make the definition of the scalar product more general. It just returns val .*
- unsigned long long int [Math::conj](#) (const unsigned long long int &val)
- This function exists to make the definition of the scalar product more general. It just returns val .*
- float [Math::conj](#) (const float &val)
- This function exists to make the definition of the scalar product more general. It just returns val .*
- double [Math::conj](#) (const double &val)
- This function exists to make the definition of the scalar product more general. It just returns val .*
- long double [Math::conj](#) (const long double &val)
- This function exists to make the definition of the scalar product more general. It just returns val .*
- wchar_t [Math::conj](#) (const wchar_t &val)
- This function exists to make the definition of the scalar product more general. It just returns val .*
- template<typename K >
std::complex< K > [Math::conj](#) (const std::complex< K > &val)
- This function exists to make the definition of the scalar product more general. It just returns the complex conjugate of val .*
- template<typename K >
std::string [Math::to_string](#) (const std::complex< K > &z)
- template<typename K >
[Vector](#)< K > [Math::operator+](#) (const [Vector](#)< K > &v, const [Vector](#)< K > &w)
- template<typename K >
[Vector](#)< K > [Math::operator+](#) (const [Vector](#)< K > &v, const K &alpha)
- template<typename K >
[Vector](#)< K > [Math::operator+](#) (const K &alpha, const [Vector](#)< K > &w)
- template<typename K >
[Vector](#)< K > [Math::operator-](#) (const [Vector](#)< K > &v, const [Vector](#)< K > &w)
- template<typename K >
[Vector](#)< K > [Math::operator-](#) (const [Vector](#)< K > &v, const K &alpha)
- template<typename K >
[Vector](#)< K > [Math::operator-](#) (const K &alpha, const [Vector](#)< K > &w)
- template<typename K >
[Vector](#)< K > [Math::operator*](#) (const [Vector](#)< K > &v, const [Vector](#)< K > &w)
- template<typename K >
[Vector](#)< K > [Math::operator*](#) (const K &alpha, const [Vector](#)< K > &v)
- template<typename K >
[Vector](#)< K > [Math::operator*](#) (const [Vector](#)< K > &v, const K &alpha)
- template<typename K >
[Vector](#)< K > [Math::operator/](#) (const [Vector](#)< K > &v, const [Vector](#)< K > &w)
- template<typename K >
[Vector](#)< K > [Math::operator/](#) (const [Vector](#)< K > &v, const K &alpha)
- template<typename K >
[Vector](#)< K > [Math::operator/](#) (const K &alpha, const [Vector](#)< K > &w)
- template<typename K >
bool [Math::operator](#)< (const [Vector](#)< K > &v, const [Vector](#)< K > &w)
- template<typename K >
bool [Math::operator](#)< (const [Vector](#)< K > &v, const K &alpha)
- template<typename K >
bool [Math::operator](#)< (const K &alpha, const [Vector](#)< K > &v)
- template<typename K >
bool [Math::operator](#)<= (const [Vector](#)< K > &v, const [Vector](#)< K > &w)
- template<typename K >
bool [Math::operator](#)<= (const [Vector](#)< K > &v, const K &alpha)

- `template<typename K >`
`bool Math::operator<= (const K &alpha, const Vector< K > &v)`
- `template<typename K >`
`bool Math::operator> (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`bool Math::operator> (const Vector< K > &v, const K &alpha)`
- `template<typename K >`
`bool Math::operator> (const K &alpha, const Vector< K > &v)`
- `template<typename K >`
`bool Math::operator>= (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`bool Math::operator>= (const Vector< K > &v, const K &alpha)`
- `template<typename K >`
`bool Math::operator>= (const K &alpha, const Vector< K > &v)`
- `template<typename K >`
`bool Math::operator== (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`bool Math::operator== (const Vector< K > &v, const K &alpha)`
- `template<typename K >`
`bool Math::operator== (const K &alpha, const Vector< K > &v)`
- `template<typename K >`
`bool Math::operator!= (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`bool Math::operator!= (const Vector< K > &v, const K &alpha)`
- `template<typename K >`
`bool Math::operator!= (const K &alpha, const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::operator& (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`Vector< K > Math::operator& (const Vector< K > &v, const K &w)`
- `template<typename K >`
`Vector< K > Math::operator& (const K &v, const Vector< K > &w)`
- `template<typename K >`
`K Math::operator| (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`K Math::vector_product_2d (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`Vector< K > Math::vector_product_3d (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`std::ostream & Math::operator<< (std::ostream &os, const Vector< K > &v)`
- `template<typename K >`
`K Math::min (const Vector< K > &v)`
- `template<typename K >`
`K Math::max (const Vector< K > &v)`
- `template<typename K >`
`K Math::sum (const Vector< K > &v)`
- `template<typename K >`
`K Math::multiply (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::cos (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::sin (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::tan (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::acos (const Vector< K > &v)`

- `template<typename K >`
`Vector< K > Math::asin (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::atan (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::atan2 (const K v, const Vector< K > &w)`
- `template<typename K >`
`Vector< K > Math::atan2 (const Vector< K > &v, const K w)`
- `template<typename K >`
`Vector< K > Math::atan2 (const Vector< K > &v, const Vector< K > &w)`
- `template<typename K >`
`Vector< K > Math::cosh (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::sinh (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::tanh (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::acosh (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::asinh (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::atanh (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::exp (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::frexp (const Vector< K > &v, Vector< int > *exp)`
- `template<typename K >`
`Vector< K > Math::ldexp (const Vector< K > &v, const int exp)`
- `template<typename K >`
`Vector< K > Math::ldexp (const Vector< K > &v, const Vector< int > &exp)`
- `template<typename K >`
`Vector< K > Math::log (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::log10 (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::modf (const Vector< K > &v, Vector< K > *intpart)`
- `template<typename K >`
`Vector< K > Math::exp2 (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::expm1 (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::ilogb (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::log1p (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::log2 (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::logb (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::scalbn (const Vector< K > &v, const int n)`
- `template<typename K >`
`Vector< K > Math::scalbn (const Vector< K > &v, const Vector< int > &n)`
- `template<typename K >`
`Vector< K > Math::scalbln (const Vector< K > &v, const long int n)`
- `template<typename K >`
`Vector< K > Math::scalbln (const Vector< K > &v, const Vector< long int > &n)`

- `template<typename K >`
`Vector< K > Math::pow (const K v, const Vector< K > &exponent)`
- `template<typename K >`
`Vector< K > Math::pow (const Vector< K > &v, const K exponent)`
- `template<typename K >`
`Vector< K > Math::pow (const Vector< K > &v, const Vector< K > &exponent)`
- `template<typename K >`
`Vector< K > Math::sqrt (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::cbrt (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::hypot (const K x, const Vector< K > &y)`
- `template<typename K >`
`Vector< K > Math::hypot (const Vector< K > &x, const K y)`
- `template<typename K >`
`Vector< K > Math::hypot (const Vector< K > &x, const Vector< K > &y)`
- `template<typename K >`
`Vector< K > Math::erf (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::erfc (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::tgamma (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::lgamma (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::ceil (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::floor (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::fmod (const K numer, const Vector< K > &denom)`
- `template<typename K >`
`Vector< K > Math::fmod (const Vector< K > &numer, const K denom)`
- `template<typename K >`
`Vector< K > Math::fmod (const Vector< K > &numer, const Vector< K > &denom)`
- `template<typename K >`
`Vector< K > Math::trunc (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::round (const Vector< K > &v)`
- `template<typename K >`
`Vector< long int > Math::lround (const Vector< K > &v)`
- `template<typename K >`
`Vector< long long int > Math::llround (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::rint (const Vector< K > &v)`
- `template<typename K >`
`Vector< long int > Math::lrint (const Vector< K > &v)`
- `template<typename K >`
`Vector< long long int > Math::llrint (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::nearbyint (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::remainder (const K numer, const Vector< K > &denom)`
- `template<typename K >`
`Vector< K > Math::remainder (const Vector< K > &numer, const K denom)`
- `template<typename K >`
`Vector< K > Math::remainder (const Vector< K > &numer, const Vector< K > &denom)`

- `template<typename K >`
`Vector< K > Math::remquo (const K numer, const Vector< K > &denom, Vector< int > *quot)`
- `template<typename K >`
`Vector< K > Math::remquo (const Vector< K > &numer, const K denom, Vector< int > *quot)`
- `template<typename K >`
`Vector< K > Math::remquo (const Vector< K > &numer, const Vector< K > &denom, Vector< int > *quot)`
- `template<typename K >`
`Vector< K > Math::copysign (const Vector< K > &x, const K y)`
- `template<typename K >`
`Vector< K > Math::copysign (const Vector< K > &x, const Vector< K > &y)`
- `template Vector< std::complex< double > > Math::nan (const unsigned int N, const char *tagp)`
- `template<typename K >`
`Vector< K > Math::nextafter (const Vector< K > &x, const Vector< K > &y)`
- `template<typename K >`
`Vector< K > Math::fdim (const K x, const Vector< K > &y)`
- `template<typename K >`
`Vector< K > Math::fdim (const Vector< K > &x, K y)`
- `template<typename K >`
`Vector< K > Math::fdim (const Vector< K > &x, const Vector< K > &y)`
- `template<typename K >`
`Vector< double > Math::fabs (const Vector< K > &v)`
- `template<typename K >`
`Vector< double > Math::abs (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::fma (const K x, const Vector< K > &y, const Vector< K > &z)`
- `template<typename K >`
`Vector< K > Math::fma (const Vector< K > &x, const K y, const Vector< K > &z)`
- `template<typename K >`
`Vector< K > Math::fma (const Vector< K > &x, const Vector< K > &y, const K z)`
- `template<typename K >`
`Vector< K > Math::fma (const K x, const K y, const Vector< K > &z)`
- `template<typename K >`
`Vector< K > Math::fma (const K x, const Vector< K > &y, const K z)`
- `template<typename K >`
`Vector< K > Math::fma (const Vector< K > &x, const K y, const K z)`
- `template<typename K >`
`Vector< K > Math::fma (const Vector< K > &x, const Vector< K > &y, const Vector< K > &z)`
- `template<typename K >`
`Vector< int > Math::fpclassify (const Vector< K > &v)`
- `template<typename K >`
`bool Math::isfinite (const Vector< K > &v)`
- `template<typename K >`
`bool Math::isinf (const Vector< K > &v)`
- `template<typename K >`
`bool Math::isnan (const Vector< K > &v)`
- `template<typename K >`
`bool Math::isnormal (const Vector< K > &v)`
- `template<typename K >`
`Vector< K > Math::real (const Vector< std::complex< K > > &v)`
- `template<typename K >`
`Vector< K > Math::imag (const Vector< std::complex< K > > &v)`
- `template<typename K >`
`Vector< K > Math::arg (const Vector< std::complex< K > > &v)`
- `template<typename K >`
`Vector< K > Math::conj (const Vector< K > &v)`

8.27.1 Detailed Description

This file contains all prototypes related to class `Vector<K>`.

Author

Andrés Laín Sanclemente

Version

0.9.0

Date

9th September 2021

8.28 Vector.hpp

[Go to the documentation of this file.](#)

```
1
10 #ifndef VECTOR_HPP
11 #define VECTOR_HPP
12
13 #include <string>
14 #include <complex>
15
21 namespace Math
22 {
30     char inline conj(const char& val)
31     {
32         return val;
33     }
34
42     unsigned char inline conj(const unsigned char& val)
43     {
44         return val;
45     }
46
54     short int inline conj(const short int& val)
55     {
56         return val;
57     }
58
66     unsigned short int inline conj(const unsigned short int& val)
67     {
68         return val;
69     }
70
78     int inline conj(const int& val)
79     {
80         return val;
81     }
82
90     unsigned int inline conj(const unsigned int& val)
91     {
92         return val;
93     }
94
102    long int inline conj(const long int& val)
103    {
104        return val;
105    }
106
114    unsigned long int inline conj(const unsigned long int& val)
115    {
116        return val;
117    }
118
126    long long int inline conj(const long long int& val)
127    {
128        return val;
129    }
```

```

129     }
130
131     unsigned long long int inline conj(const unsigned long long int& val)
132     {
133         return val;
134     }
135
136     float inline conj(const float& val)
137     {
138         return val;
139     }
140
141     double inline conj(const double& val)
142     {
143         return val;
144     }
145
146     long double inline conj(const long double& val)
147     {
148         return val;
149     }
150
151     wchar_t inline conj(const wchar_t& val)
152     {
153         return val;
154     }
155
156     template<typename K>
157     std::complex<K> inline conj(const std::complex<K>& val)
158     {
159         return std::conj(val);
160     }
161
162     template<typename K>
163     std::string inline to_string(const std::complex<K>& z)
164     {
165         using std::to_string;
166         return to_string(z.real()) + " + " + to_string(z.imag()) + "i";
167     }
168
169     // Forward declaration to make function declaration possible.
170     template<typename K>
171     class Vector;
172
173     // And we now declare all friend operators.
174     template<typename K>
175     Vector<K> operator+(const Vector<K>& v, const Vector<K>& w);
176     template<typename K>
177     Vector<K> operator+(const Vector<K>& v, const K& alpha);
178     template<typename K>
179     Vector<K> operator-(const K& alpha, const Vector<K>& w);
180     template<typename K>
181     Vector<K> operator-(const Vector<K>& v, const Vector<K>& w);
182     template<typename K>
183     Vector<K> operator-(const Vector<K>& v, const K& alpha);
184     template<typename K>
185     Vector<K> operator*(const K& alpha, const Vector<K>& w);
186     template<typename K>
187     Vector<K> operator*(const Vector<K>& v, const Vector<K>& w);
188     template<typename K>
189     Vector<K> operator*(const K& alpha, const Vector<K>& v);
190     template<typename K>
191     Vector<K> operator/(const Vector<K>& v, const K& alpha);
192     template<typename K>
193     Vector<K> operator/(const Vector<K>& v, const Vector<K>& w);
194     template<typename K>
195     Vector<K> operator/(const Vector<K>& v, const K& alpha);
196     template<typename K>
197     Vector<K> operator/(const K& alpha, const Vector<K>& w);
198
199     template<typename K>
200     bool operator<(const Vector<K>& v, const Vector<K>& w);
201     template<typename K>
202     bool operator<(const Vector<K>& v, const K& alpha);
203     template<typename K>
204     bool operator<(const K& alpha, const Vector<K>& v);
205     template<typename K>
206     bool operator<=(const Vector<K>& v, const Vector<K>& w);
207     template<typename K>
208     bool operator<=(const Vector<K>& v, const K& alpha);
209     template<typename K>
210     bool operator<=(const K& alpha, const Vector<K>& v);
211     template<typename K>
212     bool operator>(const Vector<K>& v, const Vector<K>& w);
213     template<typename K>
214     bool operator>(const Vector<K>& v, const K& alpha);
215     template<typename K>
216     bool operator>(const K& alpha, const Vector<K>& v);

```

```

259     bool operator>(const K& alpha, const Vector<K>& v);
260     template<typename K>
261     bool operator>=(const Vector<K>& v, const Vector<K>& w);
262     template<typename K>
263     bool operator>=(const Vector<K>& v, const K& alpha);
264     template<typename K>
265     bool operator>=(const K& alpha, const Vector<K>& v);
266     template<typename K>
267     bool operator==(const Vector<K>& v, const Vector<K>& w);
268     template<typename K>
269     bool operator==(const Vector<K>& v, const K& alpha);
270     template<typename K>
271     bool operator==(const K& alpha, const Vector<K>& v);
272     template<typename K>
273     bool operator!=(const Vector<K>& v, const Vector<K>& w);
274     template<typename K>
275     bool operator!=(const Vector<K>& v, const K& alpha);
276     template<typename K>
277     bool operator!=(const K& alpha, const Vector<K>& v);
278
279     template<typename K>
280     Vector<K> operator&(const Vector<K>& v, const Vector<K>& w);
281     template<typename K>
282     Vector<K> operator&(const Vector<K>& v, const K& w);
283     template<typename K>
284     Vector<K> operator&(const K& v, const Vector<K>& w);
285
286     template<typename K>
287     K operator|(const Vector<K>& v, const Vector<K>& w);
288     template<typename K>
289     K vector_product_2d(const Vector<K>& v, const Vector<K>& w);
290     template<typename K>
291     Vector<K> vector_product_3d(const Vector<K>& v, const Vector<K>& w);
292
293     template<typename K>
294     std::ostream& operator<< (std::ostream& os, const Vector<K>& v);
295
296     template<typename K>
297     K min(const Vector<K>& v);
298     template<typename K>
299     K max(const Vector<K>& v);
300     template<typename K>
301     K sum(const Vector<K>& v);
302     template<typename K>
303     K multiply(const Vector<K>& v);
304     template<typename K>
305     Vector<K> cos(const Vector<K>& v);
306     template<typename K>
307     Vector<K> sin(const Vector<K>& v);
308     template<typename K>
309     Vector<K> tan(const Vector<K>& v);
310     template<typename K>
311     Vector<K> acos(const Vector<K>& v);
312     template<typename K>
313     Vector<K> asin(const Vector<K>& v);
314     template<typename K>
315     Vector<K> atan(const Vector<K>& v);
316     template<typename K>
317     Vector<K> atan2(const K v, const Vector<K>& w);
318     template<typename K>
319     Vector<K> atan2(const Vector<K>& v, const K w);
320     template<typename K>
321     Vector<K> atan2(const Vector<K>& v, const Vector<K>& w);
322     template<typename K>
323     Vector<K> cosh(const Vector<K>& v);
324     template<typename K>
325     Vector<K> sinh(const Vector<K>& v);
326     template<typename K>
327     Vector<K> tanh(const Vector<K>& v);
328     template<typename K>
329     Vector<K> acosh(const Vector<K>& v);
330     template<typename K>
331     Vector<K> asinh(const Vector<K>& v);
332     template<typename K>
333     Vector<K> atanh(const Vector<K>& v);
334     template<typename K>
335     Vector<K> exp(const Vector<K>& v);
336     template<typename K>
337     Vector<K> frexp(const Vector<K>& v, Vector<int>*& exp);
338     template<typename K>
339     Vector<K> ldexp(const Vector<K>& v, const int exp);
340     template<typename K>
341     Vector<K> ldexp(const Vector<K>& v, const Vector<int>& exp);
342     template<typename K>
343     Vector<K> log(const Vector<K>& v);
344     template<typename K>
345     Vector<K> log10(const Vector<K>& v);

```



```

346     template<typename K>
347     Vector<K> modf(const Vector<K>& v, Vector<K>* intpart);
348     template<typename K>
349     Vector<K> exp2(const Vector<K>& v);
350     template<typename K>
351     Vector<K> expm1(const Vector<K>& v);
352     template<typename K>
353     Vector<K> ilogb(const Vector<K>& v);
354     template<typename K>
355     Vector<K> loglp(const Vector<K>& v);
356     template<typename K>
357     Vector<K> log2(const Vector<K>& v);
358     template<typename K>
359     Vector<K> logb(const Vector<K>& v);
360     template<typename K>
361     Vector<K> scalbn(const Vector<K>& v, const int n);
362     template<typename K>
363     Vector<K> scalbn(const Vector<K>& v, const Vector<int>& n);
364     template<typename K>
365     Vector<K> scalbln(const Vector<K>& v, const long int n);
366     template<typename K>
367     Vector<K> scalbln(const Vector<K>& v, const Vector<long int>& n);
368     template<typename K>
369     Vector<K> pow(const K v, const Vector<K>& exponent);
370     template<typename K>
371     Vector<K> pow(const Vector<K>& v, const K exponent);
372     template<typename K>
373     Vector<K> pow(const Vector<K>& v, const Vector<K>& exponent);
374     template<typename K>
375     Vector<K> sqrt(const Vector<K>& v);
376     template<typename K>
377     Vector<K> cbrt(const Vector<K>& v);
378     template<typename K>
379     Vector<K> hypot(const K x, const Vector<K>& y);
380     template<typename K>
381     Vector<K> hypot(const Vector<K>& x, const K y);
382     template<typename K>
383     Vector<K> hypot(const Vector<K>& x, const Vector<K>& y);
384     template<typename K>
385     Vector<K> erf(const Vector<K>& v);
386     template<typename K>
387     Vector<K> erfc(const Vector<K>& v);
388     template<typename K>
389     Vector<K> tgamma(const Vector<K>& v);
390     template<typename K>
391     Vector<K> lgamma(const Vector<K>& v);
392     template<typename K>
393     Vector<K> ceil(const Vector<K>& v);
394     template<typename K>
395     Vector<K> floor(const Vector<K>& v);
396     template<typename K>
397     Vector<K> fmod(const K numer, const Vector<K>& denom);
398     template<typename K>
399     Vector<K> fmod(const Vector<K>& numer, const K denom);
400     template<typename K>
401     Vector<K> fmod(const Vector<K>& numer, const Vector<K>& denom);
402     template<typename K>
403     Vector<K> trunc(const Vector<K>& v);
404     template<typename K>
405     Vector<K> round(const Vector<K>& v);
406     template<typename K>
407     Vector<long int> lround(const Vector<K>& v);
408     template<typename K>
409     Vector<long long int> llround(const Vector<K>& v);
410     template<typename K>
411     Vector<K> rint(const Vector<K>& v);
412     template<typename K>
413     Vector<long int> lrint(const Vector<K>& v);
414     template<typename K>
415     Vector<long long int> llrint(const Vector<K>& v);
416     template<typename K>
417     Vector<K> nearbyint(const Vector<K>& v);
418     template<typename K>
419     Vector<K> remainder(const K numer, const Vector<K>& denom);
420     template<typename K>
421     Vector<K> remainder(const Vector<K>& numer, const K denom);
422     template<typename K>
423     Vector<K> remainder(const Vector<K>& numer, const Vector<K>& denom);
424     template<typename K>
425     Vector<K> remquo(const K numer, const Vector<K>& denom, Vector<int>* quot);
426     template<typename K>
427     Vector<K> remquo(const Vector<K>& numer, const K denom, Vector<int>* quot);
428     template<typename K>
429     Vector<K> remquo(const Vector<K>& numer, const Vector<K>& denom, Vector<int>* quot);
430     template<typename K>
431     Vector<K> copysign(const Vector<K>& x, const K y);
432     template<typename K>

```

```

433 Vector<K> copysign(const Vector<K>& x, const Vector<K>& y);
434 template<typename K>
435 Vector<K> nan(const unsigned int N, const char* tagp);
436 template<typename K>
437 Vector<K> nextafter(const Vector<K>& x, const Vector<K>& y);
438 template<typename K>
439 Vector<K> fdim(const K x, const Vector<K>& y);
440 template<typename K>
441 Vector<K> fdim(const Vector<K>& x, K y);
442 template<typename K>
443 Vector<K> fdim(const Vector<K>& x, const Vector<K>& y);
444 template<typename K>
445 Vector<double> fabs(const Vector<K>& v);
446 template<typename K>
447 Vector<double> abs(const Vector<K>& v);
448 template<typename K>
449 Vector<K> fma(const K x, const Vector<K>& y, const Vector<K>& z);
450 template<typename K>
451 Vector<K> fma(const Vector<K>& x, const K y, const Vector<K>& z);
452 template<typename K>
453 Vector<K> fma(const Vector<K>& x, const Vector<K>& y, const K z);
454 template<typename K>
455 Vector<K> fma(const K x, const K y, const Vector<K>& z);
456 template<typename K>
457 Vector<K> fma(const K x, const Vector<K>& y, const K z);
458 template<typename K>
459 Vector<K> fma(const Vector<K>& x, const K y, const K z);
460 template<typename K>
461 Vector<K> fma(const Vector<K>& x, const Vector<K>& y, const Vector<K>& z);
462 template<typename K>
463 Vector<int> fpclassify(const Vector<K>& v);
464 template<typename K>
465 bool isfinite(const Vector<K>& v);
466 template<typename K>
467 bool isinf(const Vector<K>& v);
468 template<typename K>
469 bool isnan(const Vector<K>& v);
470 template<typename K>
471 bool isnormal(const Vector<K>& v);
472
473 template<typename K>
474 Vector<K> real(const Vector<std::complex<K>& v);
475 template<typename K>
476 Vector<K> imag(const Vector<std::complex<K>& v);
477 template<typename K>
478 Vector<K> arg(const Vector<std::complex<K>& v);
479 template<typename K>
480 Vector<K> conj(const Vector<K>& v);
481
482
483 template<typename K>
484 class Vector
485 {
486 public:
487     K* components;
488
489     unsigned int N;
490
491 public:
492     // Constructors and destructor.
493
494     Vector<K>();
495
496     explicit Vector<K>(const unsigned int N);
497
498     Vector<K>(const Vector<K>& w);
499
500     Vector<K>(Vector<K>&& w);
501
502     Vector<K>(const K* components, const unsigned int N);
503
504     Vector<K>(std::initializer_list<K> list);
505
506     ~Vector<K>();
507
508     // Assignment operators
509
510     Vector<K>& operator=(const K& alpha);
511
512     Vector<K>& operator=(const Vector<K>& w);
513
514     Vector<K>& operator=(Vector<K>&& w);
515
516     // Operators defined componentwise
517
518     friend Vector<K> operator+<K>(const Vector<K>& v, const Vector<K>& w);
519

```

```

653     friend Vector<K> operator+<K>(const Vector<K>& v, const K& alpha);
654
664     friend Vector<K> operator+<K>(const K& alpha, const Vector<K>& w);
665
680     friend Vector<K> operator-<K>(const Vector<K>& v, const Vector<K>& w);
681
691     friend Vector<K> operator-<K>(const Vector<K>& v, const K& alpha);
692
702     friend Vector<K> operator-<K>(const K& alpha, const Vector<K>& w);
703
716     friend Vector<K> operator*<K>(const Vector<K>& v, const Vector<K>& w);
717
727     friend Vector<K> operator*<K>(const K& alpha, const Vector<K>& v);
728
738     friend Vector<K> operator*<K>(const Vector<K>& v, const K& alpha);
739
752     friend Vector<K> operator/<K>(const Vector<K>& v, const Vector<K>& w);
753
763     friend Vector<K> operator/<K>(const Vector<K>& v, const K& alpha);
764
774     friend Vector<K> operator/<K>(const K& alpha, const Vector<K>& w);
775
776     // The following operators are done component by component and && together.
788     friend bool operator< <K>(const Vector<K>& v, const Vector<K>& w);
789
798     friend bool operator< <K>(const Vector<K>& v, const K& alpha);
799
808     friend bool operator< <K>(const K& alpha, const Vector<K>& v);
809
821     friend bool operator<= <K>(const Vector<K>& v, const Vector<K>& w);
822
831     friend bool operator<= <K>(const Vector<K>& v, const K& alpha);
832
841     friend bool operator<= <K>(const K& alpha, const Vector<K>& v);
842
854     friend bool operator> <K>(const Vector<K>& v, const Vector<K>& w);
855
864     friend bool operator> <K>(const Vector<K>& v, const K& alpha);
865
874     friend bool operator> <K>(const K& alpha, const Vector<K>& v);
875
887     friend bool operator>= <K>(const Vector<K>& v, const Vector<K>& w);
888
897     friend bool operator>= <K>(const Vector<K>& v, const K& alpha);
898
907     friend bool operator>= <K>(const K& alpha, const Vector<K>& v);
908
919     friend bool operator== <K>(const Vector<K>& v, const Vector<K>& w);
920
929     friend bool operator== <K>(const Vector<K>& v, const K& alpha);
930
939     friend bool operator== <K>(const K& alpha, const Vector<K>& v);
940
949     friend bool operator!= <K>(const Vector<K>& v, const Vector<K>& w);
950
959     friend bool operator!= <K>(const Vector<K>& v, const K& alpha);
960
969     friend bool operator!= <K>(const K& alpha, const Vector<K>& v);
970
981     friend Vector<K> operator&<K>(const Vector<K>& v, const Vector<K>& w);
982
993     friend Vector<K> operator&<K>(const Vector<K>& v, const K& w);
994
1005     friend Vector<K> operator&<K>(const K& v, const Vector<K>& w);
1006
1018     friend K operator|<K>(const Vector<K>& v, const Vector<K>& w);
1019
1030     friend K vector_product_2d<K>(const Vector<K>& v, const Vector<K>& w);
1031
1042     friend Vector<K> vector_product_3d<K>(const Vector<K>& v, const Vector<K>& w);
1043
1044     // Unary operators: +, -
1050     Vector<K> operator+();
1051
1057     Vector<K> operator-();
1058
1059     // Basic math-assignment operators: +=, -=, *=, /=
1066     Vector<K>& operator+=(const K& alpha);
1067
1080     Vector<K>& operator+=(const Vector<K>& w);
1081
1088     Vector<K>& operator-=(const K& alpha);
1089
1102     Vector<K>& operator-=(const Vector<K>& w);
1103
1110     Vector<K>& operator*=(const K& alpha);
1111

```

```

1123     Vector<K>& operator*=(const Vector<K>& w);
1124
1131     Vector<K>& operator/=(const K& alpha);
1132
1144     Vector<K>& operator/=(const Vector<K>& w);
1145
1146     // Basic unary operators: +, -
1152     Vector<K> operator+() const;
1153
1159     Vector<K> operator-() const;
1160
1161     // Other operators and functions
1172     K& operator[] (const int i);
1173
1180     K operator[] (const int i) const;
1181
1194     Vector<K> slice(int first, int last) const;
1195
1202     Vector<K> reverse() const;
1203
1209     unsigned int size() const;
1210
1216     double norm_2() const;
1217
1223     double norm_1() const;
1224
1230     double norm_inf() const;
1231
1238     double norm_p(const double p) const;
1239
1246     std::string to_string() const;
1247
1255     friend std::ostream& operator<< <K>(std::ostream& os, const Vector<K>& v);
1256
1263     void read_from_file(FILE* file);
1264
1271     void write_to_file(FILE* file) const;
1272
1279     friend K min<K>(const Vector<K>& v);
1280
1287     friend K max<K>(const Vector<K>& v);
1288
1295     friend K sum<K>(const Vector<K>& v);
1296
1303     friend K multiply<K>(const Vector<K>& v);
1304
1305     // All math functions component by component
1312     friend Vector<K> cos<K>(const Vector<K>& v);
1313
1320     friend Vector<K> sin<K>(const Vector<K>& v);
1321
1328     friend Vector<K> tan<K>(const Vector<K>& v);
1329
1336     friend Vector<K> acos<K>(const Vector<K>& v);
1337
1344     friend Vector<K> asin<K>(const Vector<K>& v);
1345
1352     friend Vector<K> atan<K>(const Vector<K>& v);
1353
1361     friend Vector<K> atan2<K>(const K v, const Vector<K>& w);
1362
1370     friend Vector<K> atan2<K>(const Vector<K>& v, const K w);
1371
1382     friend Vector<K> atan2<K>(const Vector<K>& v, const Vector<K>& w);
1383
1390     friend Vector<K> cosh<K>(const Vector<K>& v);
1391
1398     friend Vector<K> sinh<K>(const Vector<K>& v);
1399
1406     friend Vector<K> tanh<K>(const Vector<K>& v);
1407
1414     friend Vector<K> acosh<K>(const Vector<K>& v);
1415
1422     friend Vector<K> asinh<K>(const Vector<K>& v);
1423
1430     friend Vector<K> atanh<K>(const Vector<K>& v);
1431
1438     friend Vector<K> exp<K>(const Vector<K>& v);
1439
1447     friend Vector<K> frexp<K>(const Vector<K>& v, Vector<int>* exp);
1448
1456     friend Vector<K> ldexp<K>(const Vector<K>& v, const int exp);
1457
1465     friend Vector<K> ldexp<K>(const Vector<K>& v, const Vector<int>& exp);
1466
1476     friend Vector<K> log<K>(const Vector<K>& v);
1477

```

```

1484     friend Vector<K> log10<K>(const Vector<K>& v);
1485
1493     friend Vector<K> modf<K>(const Vector<K>& v, Vector<K>* intpart);
1494
1501     friend Vector<K> exp2<K>(const Vector<K>& v);
1502
1509     friend Vector<K> expm1<K>(const Vector<K>& v);
1510
1517     friend Vector<K> ilogb<K>(const Vector<K>& v);
1518
1525     friend Vector<K> log1p<K>(const Vector<K>& v);
1526
1533     friend Vector<K> log2<K>(const Vector<K>& v);
1534
1541     friend Vector<K> logb<K>(const Vector<K>& v);
1542
1550     friend Vector<K> scalbn<K>(const Vector<K>& v, const int n);
1551
1561     friend Vector<K> scalbn<K>(const Vector<K>& v, const Vector<int>& n);
1562
1570     friend Vector<K> scalbln<K>(const Vector<K>& v, const long int n);
1571
1581     friend Vector<K> scalbln<K>(const Vector<K>& v, const Vector<long int>& n);
1582
1590     friend Vector<K> pow<K>(const K v, const Vector<K>& exponent);
1591
1599     friend Vector<K> pow<K>(const Vector<K>& v, const K exponent);
1600
1610     friend Vector<K> pow<K>(const Vector<K>& v, const Vector<K>& exponent);
1611
1618     friend Vector<K> sqrt<K>(const Vector<K>& v);
1619
1626     friend Vector<K> cbrt<K>(const Vector<K>& v);
1627
1635     friend Vector<K> hypot<K>(const K x, const Vector<K>& y);
1636
1644     friend Vector<K> hypot<K>(const Vector<K>& x, const K y);
1645
1655     friend Vector<K> hypot<K>(const Vector<K>& x, const Vector<K>& y);
1656
1663     friend Vector<K> erf<K>(const Vector<K>& v);
1664
1671     friend Vector<K> erfc<K>(const Vector<K>& v);
1672
1679     friend Vector<K> tgamma<K>(const Vector<K>& v);
1680
1687     friend Vector<K> lgamma<K>(const Vector<K>& v);
1688
1695     friend Vector<K> ceil<K>(const Vector<K>& v);
1696
1703     friend Vector<K> floor<K>(const Vector<K>& v);
1704
1712     friend Vector<K> fmod<K>(const K numer, const Vector<K>& denom);
1713
1721     friend Vector<K> fmod<K>(const Vector<K>& numer, const K denom);
1722
1732     friend Vector<K> fmod<K>(const Vector<K>& numer, const Vector<K>& denom);
1733
1740     friend Vector<K> trunc<K>(const Vector<K>& v);
1741
1748     friend Vector<K> round<K>(const Vector<K>& v);
1749
1756     friend Vector<long int> lround<K>(const Vector<K>& v);
1757
1764     friend Vector<long long int> llround<K>(const Vector<K>& v);
1765
1772     friend Vector<K> rint<K>(const Vector<K>& v);
1773
1780     friend Vector<long int> lrint<K>(const Vector<K>& v);
1781
1788     friend Vector<long long int> llrint<K>(const Vector<K>& v);
1789
1796     friend Vector<K> nearbyint<K>(const Vector<K>& v);
1797
1805     friend Vector<K> remainder<K>(const K numer, const Vector<K>& denom);
1806
1814     friend Vector<K> remainder<K>(const Vector<K>& numer, const K denom);
1815
1825     friend Vector<K> remainder<K>(const Vector<K>& numer, const Vector<K>& denom);
1826
1835     friend Vector<K> remquo<K>(const K numer, const Vector<K>& denom, Vector<int>* quot);
1836
1845     friend Vector<K> remquo<K>(const Vector<K>& numer, const K denom, Vector<int>* quot);
1846
1857     friend Vector<K> remquo<K>(const Vector<K>& numer, const Vector<K>& denom, Vector<int>* quot);
1858
1866     friend Vector<K> copysign<K>(const Vector<K>& x, const K y);

```

```

1867
1877     friend Vector<K> copysign<K>(const Vector<K>& x, const Vector<K>& y);
1878
1886     friend Vector<K> nan<K>(const unsigned int N, const char* tagp);
1887
1897     friend Vector<K> nextafter<K>(const Vector<K>& x, const Vector<K>& y);
1898
1906     friend Vector<K> fdim<K>(const K x, const Vector<K>& y);
1907
1915     friend Vector<K> fdim<K>(const Vector<K>& x, K y);
1916
1926     friend Vector<K> fdim<K>(const Vector<K>& x, const Vector<K>& y);
1927
1934     friend Vector<double> fabs<K>(const Vector<K>& v);
1935
1942     friend Vector<double> abs<K>(const Vector<K>& v);
1943
1955     friend Vector<K> fma<K>(const K x, const Vector<K>& y, const Vector<K>& z);
1956
1968     friend Vector<K> fma<K>(const Vector<K>& x, const K y, const Vector<K>& z);
1969
1981     friend Vector<K> fma<K>(const Vector<K>& x, const Vector<K>& y, const K z);
1982
1991     friend Vector<K> fma<K>(const K x, const K y, const Vector<K>& z);
1992
2001     friend Vector<K> fma<K>(const K x, const Vector<K>& y, const K z);
2002
2011     friend Vector<K> fma<K>(const Vector<K>& x, const K y, const K z);
2012
2023     friend Vector<K> fma<K>(const Vector<K>& x, const Vector<K>& y, const Vector<K>& z);
2024
2031     friend Vector<int> fpclassify<K>(const Vector<K>& v);
2032
2040     friend bool isfinite<K>(const Vector<K>& v);
2041
2049     friend bool isinf<K>(const Vector<K>& v);
2050
2058     friend bool isnan<K>(const Vector<K>& v);
2059
2067     friend bool isnormal<K>(const Vector<K>& v);
2068
2069     // Complex functions
2076     friend Vector<K> conj<K>(const Vector<K>& v);
2077 };
2078 }
2079
2080 #endif // VECTOR_HPP

```

8.29 Mesh/Cell.hpp File Reference

This files contains all declarations related to the basic objects of all meshes: the cells.

```

#include "../Math/Vector.hpp"
#include "../Chemistry/Reaction.hpp"
#include <functional>
#include <string>

```

Classes

- class [BaseCell](#)

The basic structure of the space discretization.

- class [GasCell](#)
- class [SolidCell](#)

8.29.1 Detailed Description

This files contains all declarations related to the basic objects of all meshes: the cells.

Author

Andrés Laín Sanclemente

Version

0.2.0

Date

9th September 2021

8.30 Cell.hpp

[Go to the documentation of this file.](#)

```

1
10 #ifndef CELL_HPP
11 #define CELL_HPP
12
13 #include "../Math/Vector.hpp"
14 #include "../Chemistry/Reaction.hpp"
15 #include <functional>
16 #include <string>
17
18
19 class BaseCell
20 {
21 public:
22     Math::Vector<double> U;
23
24     double a;
25
26     double b;
27
28     double A;
29
30     Chemistry::SolidGasReaction* QR;
31
32     BaseCell(const Math::Vector<double>& U, const double a, const double b, const double A,
33             Chemistry::SolidGasReaction* QR);
34
35     BaseCell(FILE* file, Chemistry::SolidGasReaction* QR);
36
37     double x() const;
38
39     double len() const;
40
41     void read_from_file(FILE* file);
42
43     void write_to_file(FILE* file) const;
44
45     std::string to_string() const;
46 };
47
48 class GasCell: public BaseCell
49 {
50 public:
51     // Constructors.
52     GasCell(const Math::Vector<double>& U, double a, double b, const double A,
53             Chemistry::SolidGasReaction* QR);
54
55     GasCell(FILE* file, Chemistry::SolidGasReaction* QR);
56
57     Math::Vector<double> F() const;
58
59     GasCell* right_neighbour;

```

```

147
152     GasCell* left_neighbour;
153
159     double rho;
160
166     double c;
167
173     double v;
174
180     double T;
181
187     double P;
188
193     double H;
194
199     double E;
200
205     double M;
206
211     void update();
212
218     void read_from_file(FILE* file);
219
225     std::string to_string() const;
226 };
227
228 class SolidCell: public BaseCell
229 {
230     public:
240     SolidCell(const Math::Vector<double>& U, double a, double b, const double A,
241             Chemistry::SolidGasReaction* QR);
242
248     SolidCell(FILE* file, Chemistry::SolidGasReaction* QR);
249
254     SolidCell* right_neighbour;
255
260     SolidCell* left_neighbour;
261
266     double T;
267
272     void update();
273
279     void read_from_file(FILE* file);
280
286     std::string to_string() const;
287 };
288
289 #endif // CELL_HPP

```

8.31 Mesh/Mesh.hpp File Reference

This files contains all declarations related to the mesh.

```

#include "Cell.hpp"
#include <functional>
#include <vector>

```

Classes

- class [Mesh< Cell >](#)
This object represents a collection of cells. The number of cells used can be changed. Methods for adaptive refinement are included.
- class [Mesh< Cell >::iterator](#)
Iterator object to loop through all the cells of the mesh.
- class [GasMesh](#)
- class [SolidMesh](#)

8.31.1 Detailed Description

This files contains all declarations related to the mesh.

Author

Andrés Laín Sanclemente

Version

0.2.0

Date

9th September 2021

8.32 Mesh.hpp

[Go to the documentation of this file.](#)

```
1
10 #ifndef MESH_HPP
11 #define MESH_HPP
12
13 #include "Cell.hpp"
14 #include <functional>
15 #include <vector>
16
23 template <class Cell>
24 class Mesh
25 {
26     public:
31     Cell* first_cell;
32
37     Cell* last_cell;
38
43     Chemistry::SolidGasReaction QR;
44
50     unsigned int N_cells() const;
51
57     double detail_subdivide_threshold;
58
64     double detail_merge_threshold;
65
71     double max_length_factor;
72
78     double min_length_factor;
79
84     double boundary_cell_max_length_factor;
85
91     std::function<double(const double x)> A_func;
92
97     class Iterator
98     {
99     public:
104         Cell* cell;
105
110         Iterator();
111
117         Iterator(Cell* cell);
118
124         Iterator(const Iterator& J);
125
132         Iterator& operator=(const Iterator& J);
133
140         Iterator operator+(const int i);
141
148         Iterator operator-(const int i);
149
155         Iterator operator++(int);
156
162         Iterator& operator++();
163
```

```

169         Iterator operator--(int);
170
176         Iterator& operator--();
177
183         Cell& operator*();
184
190         Cell* operator->();
191
197         operator Cell*() const;
198
208         bool operator==(const Iterator& J);
209
219         bool operator!=(const Iterator& J);
220     };
221
222     // Constructors.
223
237     Mesh(Cell* first_cell = nullptr, Cell* last_cell = nullptr,
238         const Chemistry::SolidGasReaction& QR = Chemistry::SolidGasReaction(),
239         std::function<double(const double x)> A_func = [] (double x){return 0;},
240         const double detail_subdivide_threshold = 0.01,
241         const double detail_merge_threshold = 0.001,
242         const double max_length_factor = 1./50,
243         const double min_length_factor = 1./2000,
244         const double boundary_cell_max_length_factor = 1./1000);
245
251     Mesh(const Mesh& mesh);
252
258     Mesh(Mesh&& mesh);
259
266     Mesh& operator=(const Mesh& mesh);
267
272     ~Mesh();
273
278     void free();
279
285     Mesh::Iterator begin() const;
286
292     Mesh::Iterator rbegin() const;
293
300     Mesh::Iterator end() const;
301
307     Cell* subdivide_at(Cell* C);
308
316     Cell* merge_cells(Cell* L, Cell* R);
317
325     void calculate_variable_ranges();
326
335     double detail(Cell* L, Cell* R) const;
336
343     void optimize_mesh();
344
350     std::vector<double> x() const;
351
358     std::vector<double> x_partition() const;
359
365     std::vector<double> A() const;
366
372     std::vector<Math::Vector<double>> U() const;
373
379     void read_from_file(FILE* file);
380
386     void write_to_file(FILE* file) const;
387
388     protected:
394     double ranges;
395 };
396
397 class GasMesh: public Mesh<GasCell>
398 {
399     public:
413     GasMesh(GasCell* first_cell = nullptr, GasCell* last_cell = nullptr,
414         const Chemistry::SolidGasReaction& QR = Chemistry::SolidGasReaction(),
415         std::function<double(const double x)> A_func = [] (double x){return 0;},
416         const double detail_subdivide_threshold = 0.01,
417         const double detail_merge_threshold = 0.0005,
418         const double max_length_factor = 1./20,
419         const double min_length_factor = 1./10000,
420         const double boundary_cell_max_length_factor = 1./1000);
421
427     std::vector<double> c() const;
428
434     std::vector<double> rho() const;
435
441     std::vector<double> v() const;
442
448     std::vector<double> T() const;

```

```

449
455     std::vector<double> P() const;
456
462     std::vector<double> M() const;
463 };
464
465 class SolidMesh: public Mesh<SolidCell>
466 {
467     public:
481     SolidMesh(SolidCell* first_cell = nullptr, SolidCell* last_cell = nullptr,
482             const Chemistry::SolidGasReaction& QR = Chemistry::SolidGasReaction(),
483             std::function<double(const double x)> A_func = [] (double x){return 0;},
484             const double detail_subdivide_threshold = 0.01,
485             const double detail_merge_threshold = 0.0005,
486             const double max_length_factor = 1./20,
487             const double min_length_factor = 1./2000,
488             const double boundary_cell_max_length_factor = 1./1000);
489
495     std::vector<double> T() const;
496 };
497
498
499 #endif // MESH_HPP

```

8.33 Simulation.hpp

```

1 #ifndef SIMULATION_HPP
2 #define SIMULATION_HPP
3
4 #include "Mesh/Mesh.hpp"
5 #include "Chemistry/Reaction.hpp"
6 #include "Utilities/FileArray.hpp"
7 #include "Math/Integration.hpp"
8 #include "CPGF/Plot2d/Graphic.hpp"
9 #include "Utilities/Progress.hpp"
10 #include "Solvers/Solid/SolidSolvers.hpp"
11 #include <functional>
12 #include <limits>
13 #include <vector>
14
20 enum class GasBoundaryConditionsType
21 {
27     WALL,
28
33     FREE,
34
40     PERIODIC,
41
47     FIXED_PRESSURE,
48
54     FIXED_FLOW_ENTHALPY,
55
61     FIXED_DENSITY_SPEED_PRESSURE,
62
68     NONE,
69 };
70
76 class GasBoundaryConditions
77 {
78     public:
83     GasBoundaryConditionsType left;
84
89     GasBoundaryConditionsType right;
90
96     std::function<double(double t)> left_condition_1;
97
103     std::function<double(double t)> left_condition_2;
104
110     std::function<double(double t)> left_condition_3;
111
117     std::function<double(double t)> right_condition_1;
118
124     std::function<double(double t)> right_condition_2;
125
131     std::function<double(double t)> right_condition_3;
132
145     GasBoundaryConditions(const GasBoundaryConditionsType& left = GasBoundaryConditionsType::WALL,
146             const GasBoundaryConditionsType& right = GasBoundaryConditionsType::WALL,
147             std::function<double(double t)> left_condition_1 = [] (double t){return 0;},
148             std::function<double(double t)> left_condition_2 = [] (double t){return 0;},
149             std::function<double(double t)> left_condition_3 = [] (double t){return 0;},
150             std::function<double(double t)> right_condition_1 = [] (double t){return 0;},
151             std::function<double(double t)> right_condition_2 = [] (double t){return 0;},

```

```

152         std::function<double(double t)> right_condition_3 = [] (double t){return 0;});
153     };
154
155     enum class SimulationType
156     {
157         GAS,
158         SOLID,
159         BOTH,
160     };
161
162     class Simulation
163     {
164     public:
165         GasMesh instant_gas_mesh;
166         SolidMesh instant_solid_mesh;
167
168         double instant_v_q;
169         double instant_x_q;
170         double instant_t;
171
172         unsigned int refine;
173
174         std::string name;
175
176         SimulationType simulation_type;
177
178         Utilities::FileArray<GasMesh>* gas_mesh;
179
180         Utilities::FileArray<SolidMesh>* solid_mesh;
181
182         std::vector<double> x_q_array;
183         std::vector<double> v_q_array;
184         std::vector<double> t_array;
185
186         GasBoundaryConditions gas_BC;
187         SolidBoundaryConditions solid_BC;
188
189         double CFL;
190
191         unsigned int N_saves;
192
193         bool adaptive_refinement_solid;
194         bool adaptive_refinement_gas;
195
196         unsigned int adaptive_refinement_period;
197         unsigned int N_tasks;
198
199         std::function<double(const GasCell& C, const double t)> external_forces;
200
201         std::function<void(const GasCell& A, const GasCell& B, Math::Vector<double>* F, double* S_max)>
202         convection_solver;
203
204         std::function<void(SolidMesh& mesh, double& dt, const double CFL, const SolidBoundaryConditions BC,
205         const double t)> diffusion_solver;
206
207         Utilities::Progress<double> progress;
208
209         Simulation
210         (
211             const std::string& name,
212             const Chemistry::SolidGasReaction& QR,
213             const double a,
214             const double b,
215             const unsigned int N_cells_solid,
216             const unsigned int N_cells_gas,
217             const bool adaptive_refinement_solid,
218             const bool adaptive_refinement_gas,
219             const unsigned int N_tasks,
220             const double CFL,
221             const unsigned int N_saves,
222             const SolidBoundaryConditions& solid_BC,
223             const GasBoundaryConditions& gas_BC,
224             const double x_q,
225             std::function<double(double x)> v,
226             std::function<double(double x)> P,
227             std::function<double(double x)> T,

```

```

413     std::function<double(double x)> A,
414     std::function<void(const GasCell& A, const GasCell& B, Math::Vector<double>* F, double* S_max)>
convection_solver,
415     std::function<void(SolidMesh& mesh, double& dt, const double CFL,
416     const SolidBoundaryConditions BC, const double t)> diffusion_solver,
417     std::function<double(std::function<double(double x)> f, const double a, const double b)>
integrator =
418     [] (std::function<double(double x)> f, const double a, const double b)
419     {return Math::Integrators::Gauss_Konrad_G7_K15(f, a, b);}),
420     std::function<double(const GasCell& C, const double t)> external_forces = [] (const GasCell& C,
const double t){return 0.;}),
421     const unsigned int adaptive_refinement_period = 1,
422     const double detail_subdivide_threshold = 0.01,
423     const double detail_merge_threshold = 0.001,
424     const double max_length_factor = 1./50,
425     const double min_length_factor = 1./2000,
426     const double boundary_cell_max_length_factor = 1./1000
427 );
428
461 Simulation
462 (
463     const std::string& name,
464     const Chemistry::SolidGasReaction& QR,
465     const double a,
466     const double b,
467     const unsigned int N_cells,
468     const bool adaptive_refinement,
469     const unsigned int N_tasks,
470     const double CFL,
471     const unsigned int N_saves,
472     const GasBoundaryConditions& BC,
473     std::function<double(double x)> rho,
474     std::function<double(double x)> v,
475     std::function<double(double x)> P,
476     std::function<double(double x)> A,
477     std::function<void(const GasCell& A, const GasCell& B, Math::Vector<double>* F, double* S_max)>
convection_solver,
478     std::function<double(std::function<double(double x)> f, const double a, const double b)>
integrator =
479     [] (std::function<double(double x)> f, const double a, const double b)
480     {return Math::Integrators::Gauss_Konrad_G7_K15(f, a, b);}),
481     std::function<double(const GasCell& C, const double t)> external_forces = [] (const GasCell& C,
const double t){return 0.;}),
482     const unsigned int adaptive_refinement_period = 1,
483     const double detail_subdivide_threshold = 0.01,
484     const double detail_merge_threshold = 0.001,
485     const double max_length_factor = 1./50,
486     const double min_length_factor = 1./2000,
487     const double boundary_cell_max_length_factor = 1./1000
488 );
489
518 Simulation
519 (
520     const std::string& name,
521     const Chemistry::SolidGasReaction& QR,
522     const double a,
523     const double b,
524     const unsigned int N_cells,
525     const bool adaptive_refinement,
526     const unsigned int N_tasks,
527     const double CFL,
528     const unsigned int N_saves,
529     const SolidBoundaryConditions& BC,
530     std::function<double(double x)> T,
531     std::function<double(double x)> A,
532     std::function<void(SolidMesh& mesh, double& dt, const double CFL,
533     const SolidBoundaryConditions BC, const double t)> diffusion_solver,
534     std::function<double(std::function<double(double x)> f, const double a, const double b)>
integrator =
535     [] (std::function<double(double x)> f, const double a, const double b)
536     {return Math::Integrators::Gauss_Konrad_G7_K15(f, a, b);}),
537     const unsigned int adaptive_refinement_period = 1,
538     const double detail_subdivide_threshold = 0.01,
539     const double detail_merge_threshold = 0.001,
540     const double max_length_factor = 1./50,
541     const double min_length_factor = 1./2000,
542     const double boundary_cell_max_length_factor = 1./1000
543 );
544
550 explicit Simulation(const std::string& file);
551
552 ~Simulation();
553
559 void update(double dt = std::numeric_limits<double>::max());
560
567 void simulate_until(const double t);
568

```

```

574 void write_to_file(const std::string& file) const;
575
585 double x_q(const double t) const;
586
595 double v_q(const double t) const;
596
605 std::function<double(const double x)> A(const double t) const;
606
615 std::function<double(const double x)> rho(const double t) const;
616
625 std::function<double(const double x)> c(const double t) const;
626
635 std::function<double(const double x)> v(const double t) const;
636
645 std::function<double(const double x)> T(const double t) const;
646
655 std::function<double(const double x)> P(const double t) const;
656
665 std::function<double(const double x)> M(const double t) const;
666
673 CPGF::Plot2d::Graphic* mesh_plot() const;
674
675 protected:
681 double a;
682
688 double b;
689
694 double d;
695
700 double rho_g;
701
706 double v_g;
707 };
708
709 #endif // SIMULATION_HPP

```

8.34 ExactRiemannSolver.hpp

```

1 #ifndef EXACT_RIEMANN_SOLVER_HPP
2 #define EXACT_RIEMANN_SOLVER_HPP
3
4 namespace Solvers
5 {
6     namespace Gas
7     {
13         enum class VacuumState
14         {
15             VACUUM_LEFT_STATE,
16             VACUUM_RIGHT_STATE,
17             GENERATED_VACUUM,
18             NO_VACUUM,
19         };
20
34         class ExactRiemannSolver
35         {
36         public:
44             double rho(const double x, const double t) const;
45
53             double v(const double x, const double t) const;
54
62             double P(const double x, const double t) const;
63
69             double S_max() const;
70
84             ExactRiemannSolver(const double rho_L, const double v_L, const double P_L,
85                               const double rho_R, const double v_R, const double P_R, const double gamma,
86                               const double tol);
87
88         protected:
89
94             double gamma;
95
100             VacuumState vacuum_state;
101
106             double rho_L;
107
112             double v_L;
113
118             double P_L;
119
124             double rho_R;
125
130             double v_R;

```

```

131
136     double P_R;
137
138     // Gamma constants.
143     double G1;
144
149     double G2;
150
155     double G3;
156
161     double G4;
162
167     double G5;
168
173     double G6;
174
179     double G7;
180
185     double G8;
186
191     double A_L;
192
197     double A_R;
198
203     double B_L;
204
209     double B_R;
210
215     double c_L;
216
221     double c_R;
222
229     double f_L(const double P) const;
230
237     double f_R(const double P) const;
238
245     double f(const double P) const;
246
253     double df_LdP(const double P) const;
254
261     double df_RdP(const double P) const;
262
269     double dfdP(const double P) const;
270
275     double S_L;
276
281     double S_HL;
282
287     double S_TL;
288
293     double S_starL;
294
299     double rho_starL;
300
305     double v_star;
306
311     double P_star;
312
317     double S_R;
318
323     double S_HR;
324
329     double S_TR;
330
335     double S_starR;
336
341     double rho_starR;
342
349     double rho_Lrf(const double S) const;
350
357     double v_Lrf(const double S) const;
358
365     double P_Lrf(const double S) const;
366
373     double rho_Rrf(const double S) const;
374
381     double v_Rrf(const double S) const;
382
389     double P_Rrf(const double S) const;
390
397     double rho_L0(const double S) const;
398
405     double v_L0(const double S) const;
406
413     double P_L0(const double S) const;
414
421     double rho_R0(const double S) const;

```

```

422
429         double v_R0(const double S) const;
430
437         double P_R0(const double S) const;
438     };
439 }
440 }
441
442 #endif // EXACT_RIEMANN_SOLVER_HPP

```

8.35 ExactSteadySolver.hpp

```

1 #ifndef EXACT_STEADY_SOLVER_HPP
2 #define EXACT_STEADY_SOLVER_HPP
3
4 #include "../Math/Interpolation.hpp"
5 #include <functional>
6 #include <vector>
7
8 namespace Solvers
9 {
10     namespace Gas
11     {
12         {
13             enum class SolutionType
14             {
15                 SUBSONIC,
16                 SUPERSONIC,
17             };
18
19             class ExactSteadySolver
20             {
21             public:
22                 ExactSteadySolver(const double rho_0, const double P_0, const double T_0,
23                     const double M_x, const double A_x, const double gamma,
24                     const SolutionType solution_type);
25
26                 double M(const double A) const;
27
28                 double rho(const double A) const;
29
30                 double v(const double A) const;
31
32                 double P(const double A) const;
33
34                 double T(const double A) const;
35
36             protected:
37                 SolutionType solution_type;
38
39                 double C;
40
41                 double rho_0;
42
43                 double P_0;
44
45                 double T_0;
46
47                 double R;
48
49                 double gamma;
50
51                 double G1;
52
53                 double G2;
54
55                 double G3;
56
57                 double G4;
58             };
59         }
60     }
61 }
62
63 #endif // EXACT_STEADY_SOLVER_HPP

```

8.36 GasSolvers.hpp

```

1 #ifndef GAS_SOLVERS_HPP
2 #define GAS_SOLVERS_HPP

```



```

3
4 #include "../Math/Vector.hpp"
5 #include "../Mesh/Cell.hpp"
6
7 namespace Solvers
8 {
9     namespace Gas
10    {
11        const double EXACT_RIEMANN_SOLVER_RELATIVE_TOLERANCE = 1e-6;
12
13        void exact(const GasCell& L, const GasCell& R, Math::Vector<double>* F, double* S_max);
14
15        void HLL(const GasCell& L, const GasCell& R, Math::Vector<double>* F, double* S_max);
16
17        void HLLC(const GasCell& L, const GasCell& R, Math::Vector<double>* F, double* S_max);
18
19        void Roe(const GasCell& L, const GasCell& R, Math::Vector<double>* F, double* S_max);
20    }
21 }
22 #endif // GAS_SOLVERS_HPP

```

8.37 RocketSolver.hpp

```

1 #ifndef ROCKET_SOLVER_HPP
2 #define ROCKET_SOLVER_HPP
3
4 #include <string>
5
6 namespace Solvers
7 {
8     namespace Rocket
9     {
10        class SteadySolver
11        {
12        public:
13            SteadySolver(const double T_c, const double M_catm, const double n,
14                        const double gamma, const double R, const double A_c);
15
16            void solve_for_exit_area(const double A_e);
17
18            void calculate_optimum_parameters();
19
20            double A_c;
21
22            double A_e;
23
24            double M_catm;
25
26            double gamma;
27
28            double R;
29
30            double n;
31
32            double T_c;
33
34            double v_c;
35
36            double M_c;
37
38            double P_c;
39
40            double rho_c;
41
42            double v_e;
43
44            double M_e;
45
46            double P_e;
47
48            double T_e;
49
50            double rho_e;
51
52            double m_dot;
53
54            double thrust;
55
56            std::string to_string() const;
57
58        protected:
59            constexpr static double P_atm = 101325;
60        };
61    }
62 }
63 #endif

```

```

166
171         double G1;
172
177         double G2;
178
183         double G3;
184
189         double G4;
190     };
191 }
192 }
193
194 #endif // ROCKET_SOLVER_HPP

```

8.38 SolidSolvers.hpp

```

1 #ifndef SOLID_SOLVERS_HPP
2 #define SOLID_SOLVERS_HPP
3
4 #include <functional>
5 #include "../Mesh/Mesh.hpp"
6
11 enum class SolidBoundaryConditionsType
12 {
17     FIXED_TEMPERATURE,
18
23     FIXED_GRADIENT,
24
29     COMBUSTION_FRONT,
30 };
31
36 class SolidBoundaryConditions
37 {
38     public:
43     SolidBoundaryConditionsType left;
44
49     SolidBoundaryConditionsType right;
50
55     std::function<double(double T, double t)> left_condition;
56
61     std::function<double(double T, double t)> right_condition;
62
71     SolidBoundaryConditions(const SolidBoundaryConditionsType left =
        SolidBoundaryConditionsType::FIXED_TEMPERATURE,
72         const SolidBoundaryConditionsType right = SolidBoundaryConditionsType::FIXED_TEMPERATURE,
73         std::function<double(double T, double t)> left_condition = [](double T, double t){return 0;}),
74         std::function<double(double T, double t)> right_condition = [](double T, double t){return 0;});
75 };
76
77 namespace Solvers
78 {
79     namespace Solid
80     {
91         void euler_explicit(SolidMesh& mesh, double& dt, const double CFL, const SolidBoundaryConditions
            BC, const double t);
92
104         void euler_implicit(SolidMesh& mesh, double& dt, const double CFL, const SolidBoundaryConditions
            BC, const double t);
105     }
106 }
107
108 #endif // SOLID_SOLVERS_HPP

```

8.39 FileArray.hpp

```

1 #ifndef FILE_ARRAY_HPP
2 #define FILE_ARRAY_HPP
3
4 #include <cstdio>
5 #include <initializer_list>
6 #include <string>
7 #include <vector>
8
9 namespace Utilities
10 {
11     template <typename T>
12     class FileArray
13     {
14     public:

```

```

15     FILE* file;
16
22     unsigned long size() const;
23
24     class Iterator
25     {
26     public:
27
28         FileArray* file_array;
29         unsigned long i;
30
31         explicit Iterator(FileArray* file_array, const unsigned long i = 0);
32
33         bool operator==(const Iterator& J);
34         bool operator!=(const Iterator& J);
35
36         operator T() const;
37
38         Iterator& operator=(const T& val);
39
40         Iterator& operator++();
41         Iterator operator++(int);
42         Iterator& operator--();
43         Iterator operator--(int);
44         Iterator& operator+(const unsigned long j);
45         Iterator& operator-(const unsigned long j);
46     };
47
48     T operator[](const unsigned long i) const;
49     Iterator operator[](const unsigned long i);
50     T at(const unsigned long i) const;
51     Iterator at(const unsigned long i);
52     T front() const;
53     Iterator front();
54     T back() const;
55     Iterator back();
56
57     Iterator begin();
58     Iterator end();
59     Iterator rbegin();
60     Iterator rend();
61
62     void push_back(const T& val);
63     void pop_back();
64
65     // void insert(const unsigned long i, const T& val);
66     // void insert(const unsigned long i, const unsigned long n, const T& val);
67     // void insert(const unsigned long i, const std::initializer_list<T>& list);
68     // void erase(const unsigned long i);
69     // void erase(const unsigned long start, unsigned long stop);
70     void clear();
71
72     explicit FileArray();
73     explicit FileArray(char* file, const bool temp = false);
74     explicit FileArray(const std::string& file, const bool temp = false);
75
76     explicit FileArray(const std::vector<T>& vector);
77     FileArray(char* file, const std::vector<T>& vector, const bool temp = false);
78     FileArray(const std::string& file, const std::vector<T>& vector, const bool temp = false);
79
80     FileArray(T* array, unsigned int N);
81     FileArray(char* file, T* array, unsigned int N, const bool temp = false);
82     FileArray(const std::string& file, T* array, unsigned int N, const bool temp = false);
83
84     explicit FileArray(const std::initializer_list<T>& list);
85     FileArray(char* file, const std::initializer_list<T>& list, const bool temp = false);
86     FileArray(const std::string& file, const std::initializer_list<T>& list, const bool temp =
false);
87
88     // There is no copy constructor and assignment operator.
89     FileArray(const FileArray<T>& f_array) = delete;
90     FileArray& operator=(const FileArray<T>& f_array) = delete;
91     ~FileArray();
92
93     protected:
94
95     unsigned long N;
96
97     std::string filename;
98
99     bool temp;
100
101     std::vector<unsigned long> indexes;
102 };
103 }
104
105 #endif // FILE_ARRAY_HPP

```

8.40 FileOperations.hpp

```
1 #ifndef FILE_OPERATIONS_HPP
2 #define FILE_OPERATIONS_HPP
3
4 #include <complex>
5 #include <vector>
6 #include <cstdio>
7 #include <string>
8
9 template <class T>
10 void inline read_from_file(T& object, FILE* file)
11 {
12     object.read_from_file(file);
13 }
14
15 void inline read_from_file(char& val, FILE* file)
16 {
17     fread(&val, sizeof(char), 1, file);
18 }
19
20 void inline read_from_file(signed char& val, FILE* file)
21 {
22     fread(&val, sizeof(signed char), 1, file);
23 }
24
25 void inline read_from_file(unsigned char& val, FILE* file)
26 {
27     fread(&val, sizeof(unsigned char), 1, file);
28 }
29
30 void inline read_from_file(short int& val, FILE* file)
31 {
32     fread(&val, sizeof(short int), 1, file);
33 }
34
35 void inline read_from_file(unsigned short int& val, FILE* file)
36 {
37     fread(&val, sizeof(unsigned short int), 1, file);
38 }
39
40 void inline read_from_file(int& val, FILE* file)
41 {
42     fread(&val, sizeof(int), 1, file);
43 }
44
45 void inline read_from_file(unsigned int& val, FILE* file)
46 {
47     fread(&val, sizeof(unsigned int), 1, file);
48 }
49
50 void inline read_from_file(long int& val, FILE* file)
51 {
52     fread(&val, sizeof(long int), 1, file);
53 }
54
55 void inline read_from_file(unsigned long int& val, FILE* file)
56 {
57     fread(&val, sizeof(unsigned long int), 1, file);
58 }
59
60 void inline read_from_file(long long int& val, FILE* file)
61 {
62     fread(&val, sizeof(long long int), 1, file);
63 }
64
65 void inline read_from_file(unsigned long long int& val, FILE* file)
66 {
67     fread(&val, sizeof(unsigned long long int), 1, file);
68 }
69
70 void inline read_from_file(float& val, FILE* file)
71 {
72     fread(&val, sizeof(float), 1, file);
73 }
74
75 void inline read_from_file(double& val, FILE* file)
76 {
77     fread(&val, sizeof(double), 1, file);
78 }
79
80 void inline read_from_file(long double& val, FILE* file)
81 {
82     fread(&val, sizeof(long double), 1, file);
83 }
84
85 void inline read_from_file(wchar_t& val, FILE* file)
```

```

86 {
87     fread(&val, sizeof(wchar_t), 1, file);
88 }
89
90 template <class K>
91 void inline read_from_file(std::complex<K>& z, FILE* file)
92 {
93     K x, y;
94     read_from_file(x, file);
95     read_from_file(y, file);
96     z = std::complex<K>(x, y);
97 }
98
99 void inline read_from_file(std::string& val, FILE* file)
100 {
101     char buffer[1024];
102     unsigned int i = 0;
103     fread(buffer, sizeof(char), 1, file);
104     while(buffer[i] != '\0')
105     {
106         i++;
107         fread(buffer + i, sizeof(char), 1, file);
108     }
109     val = std::string(buffer);
110 }
111
112 template <class T>
113 void inline read_from_file(std::vector<T>& object, FILE* file)
114 {
115     unsigned int size;
116     read_from_file(size, file);
117     object = std::vector<T>(size);
118     for (unsigned int i = 0; i < size; i++)
119     {
120         read_from_file(object[i], file);
121     }
122 }
123
124 void inline read_from_file(bool& val, FILE* file)
125 {
126     char temp;
127     read_from_file(temp, file);
128     val = temp;
129 }
130
131
132 template <class T>
133 void inline write_to_file(const T& object, FILE* file)
134 {
135     object.write_to_file(file);
136 }
137
138 void inline write_to_file(const signed char& val, FILE* file)
139 {
140     fwrite(&val, sizeof(signed char), 1, file);
141 }
142
143 void inline write_to_file(const char& val, FILE* file)
144 {
145     fwrite(&val, sizeof(char), 1, file);
146 }
147
148 void inline write_to_file(const unsigned char& val, FILE* file)
149 {
150     fwrite(&val, sizeof(unsigned char), 1, file);
151 }
152
153 void inline write_to_file(const short int& val, FILE* file)
154 {
155     fwrite(&val, sizeof(short int), 1, file);
156 }
157
158 void inline write_to_file(const unsigned short int& val, FILE* file)
159 {
160     fwrite(&val, sizeof(unsigned short int), 1, file);
161 }
162
163 void inline write_to_file(const int& val, FILE* file)
164 {
165     fwrite(&val, sizeof(int), 1, file);
166 }
167
168 void inline write_to_file(const unsigned int& val, FILE* file)
169 {
170     fwrite(&val, sizeof(unsigned int), 1, file);
171 }
172

```

```

173 void inline write_to_file(const long int& val, FILE* file)
174 {
175     fwrite(&val, sizeof(long int), 1, file);
176 }
177
178 void inline write_to_file(const unsigned long int& val, FILE* file)
179 {
180     fwrite(&val, sizeof(unsigned long int), 1, file);
181 }
182
183 void inline write_to_file(const long long int& val, FILE* file)
184 {
185     fwrite(&val, sizeof(long long int), 1, file);
186 }
187
188 void inline write_to_file(const unsigned long long int& val, FILE* file)
189 {
190     fwrite(&val, sizeof(unsigned long long int), 1, file);
191 }
192
193 void inline write_to_file(const float& val, FILE* file)
194 {
195     fwrite(&val, sizeof(float), 1, file);
196 }
197
198 void inline write_to_file(const double& val, FILE* file)
199 {
200     fwrite(&val, sizeof(double), 1, file);
201 }
202
203 void inline write_to_file(const long double& val, FILE* file)
204 {
205     fwrite(&val, sizeof(long double), 1, file);
206 }
207
208 void inline write_to_file(const wchar_t& val, FILE* file)
209 {
210     fwrite(&val, sizeof(wchar_t), 1, file);
211 }
212
213 template <class K>
214 void inline write_to_file(const std::complex<K>& z, FILE* file)
215 {
216     K x = z.real();
217     K y = z.imag();
218     fwrite(&x, sizeof(K), 1, file);
219     fwrite(&y, sizeof(K), 1, file);
220 }
221
222 void inline write_to_file(const std::string& val, FILE* file)
223 {
224     fwrite(val.c_str(), sizeof(char), val.size()+1, file);
225 }
226
227 template <class T>
228 void inline write_to_file(const std::vector<T>& object, FILE* file)
229 {
230     write_to_file((unsigned int)object.size(), file);
231     for (unsigned int i = 0; i < object.size(); i++)
232     {
233         write_to_file(object[i], file);
234     }
235 }
236
237 void inline write_to_file(const bool& val, FILE* file)
238 {
239     write_to_file((char) val, file);
240 }
241
242 #endif // FILE_OPERATIONS_HPP

```

8.41 FormatNumber.hpp

```

1 #ifndef FORMAT_NUMBER_HPP
2 #define FORMAT_NUMBER_HPP
3
4 #include <functional>
5 #include <string>
6
7 namespace Utilities
8 {
9     int position_of_most_significant_digit(const double x);
10

```

```

11     unsigned long long pow10(const int n);
12
13     double round_to_precision(double x, const unsigned int precision);
14     double ceil_to_precision(double x, const unsigned int precision);
15     double floor_to_precision(double x, const unsigned int precision);
16
17     std::string format_number(double x, const bool latex = true,
18                             const unsigned int precision = 3, const bool show_sign = false,
19                             const int lim_inf = -3, const int lim_sup = 3);
20 }
21
22 #endif // FORMAT_NUMBER_HPP

```

8.42 Progress.hpp

```

1  #ifndef PROGRESS_HPP
2  #define PROGRESS_HPP
3
4  #include <string>
5  #include <vector>
6  #include <atomic>
7  #include <chrono>
8  #include <map>
9  #include <mutex>
10 #include <future>
11
12 namespace Utilities
13 {
14     std::string time_to_string(const std::chrono::system_clock::time_point date);
15     std::string duration_to_string(unsigned int seconds);
16     std::string duration_to_string(const double seconds);
17     std::string duration_to_string(const std::chrono::system_clock::duration time);
18
19     enum class ProgressStatus
20     {
21         HOLDING,
22         RUNNING,
23         PAUSED,
24         FINISHED,
25     };
26
27     enum class ProgressEstimation
28     {
29         LINEAR,
30         LOGARITHMIC,
31     };
32
33     class ProgressBase
34     {
35     protected:
36         ProgressEstimation estimation;
37         std::string name;
38         std::chrono::system_clock::time_point t_start;
39         std::chrono::system_clock::time_point t_paused;
40         std::chrono::system_clock::time_point t_finish;
41         std::chrono::system_clock::duration inactive_time;
42         std::map<std::string, ProgressBase*> children;
43         std::mutex M;
44         std::future<void> task;
45
46     public:
47         ProgressStatus status;
48         void start();
49         void pause();
50         void resume();
51         void finish();
52         virtual std::string report(const unsigned int level) const;
53
54         ProgressBase(const std::string& name = "", const ProgressEstimation estimation =
ProgressEstimation::LINEAR);
55         ProgressBase(const ProgressBase& progress);
56         ProgressBase& operator=(const ProgressBase& progress);
57
58         ProgressBase& operator[] (std::string name);
59
60         void add_child(ProgressBase& progress);
61         void eliminate_child(ProgressBase& progress);
62
63         void update_to_terminal(unsigned int period = 250);
64     };
65
66     template<class T>

```

```

68     class Progress: public ProgressBase
69     {
70     protected:
71         T initial;
72         T objective;
73         std::atomic<T> current;
74
75     public:
76         Progress(const std::string& name = "", ProgressEstimation estimation =
ProgressEstimation::LINEAR,
77                 const T initial = T(), const T objective = T());
78         Progress(Progress<T>& progress);
79         Progress<T>& operator=(const Progress<T>& progress);
80         std::string report(const unsigned int level) const override;
81
82         Progress<T>& operator=(const T& val);
83         Progress<T>& operator+=(const T& val);
84         Progress<T>& operator++();
85         Progress<T>& operator--(const T& val);
86         Progress<T>& operator--();
87         Progress<T>& operator*=(const T& val);
88         Progress<T>& operator/=(const T& val);
89         Progress<T>& operator%=(const T& val);
90
91         operator std::atomic<T>&();
92         operator T() const;
93
94     };
95 }
96
97 #endif // PROGRESS_HPP

```

8.43 ToString.hpp

```

1  #ifndef TO_STRING_HPP
2  #define TO_STRING_HPP
3
4  #include <string>
5
6  template<class T>
7  std::string inline read_from_file(const T& object)
8  {
9      return object.to_string();
10 }
11
12 #endif // TO_STRING_HPP

```