The AreaCon Library

1.0

Generated by Doxygen 1.8.6

Wed Mar 23 2016 18:08:45

Contents

Chapter 1

AreaCon: A C++ Library for Area-Constrained Partitioning

AreaCon is a light-weight, C++ library for carrying out area-constrained partitioning operations in floating-point precision. The library is primarily a numerical implementation of the area-constrained partitioning algorithm by Patel, Frasca, and Bullo, 2014 (see http://www.areacon.org/References).

Low-level source code documentation is provided here; however, additional discussion about how to use the code along with numerical examples can be found on the project's Wiki (https://github.com/jrpeters/-AreaCon/wiki).

Please cite AreaCon whenever possible. A bibtex citation is provided below:

```
@Misc{AreaCon:16,
author =        {J.R. Peters and Contributors},
title =         {The {AreaCon} Library},
howpublished =  {\texttt{http://www.areacon.org}},
year =          {2016},
note =          {v. 1},
abstract =      {A C++ library for area-constrained partitioning.},
}
```

For more information on the code and information on how to contact me, please see http://www.areacon.-org

Author

Jeffrey R. Peters (http://www.jeffreyrpeters.com , https://github.com/jrpeters)

Version

1.0

Date

Feb. 2016

Copyright

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 AreaCon Namespace Reference

Classes

- class Point
- class Poly
- class Mult_Array
- class DelaunayGraph
- class Int_Params
- class Parameters
- class Density
- class Partition

Chapter 6

Class Documentation

6.1 AreaCon::DelaunayGraph Class Reference

`#include <areacon.h>`

Public Member Functions

- DelaunayGraph (const int NRegions)
- DelaunayGraph (const DelaunayGraph &obj)
- DelaunayGraph & operator= (const DelaunayGraph &obj)
- ~DelaunayGraph ()

Public Attributes

- Point *** Graph
- const int NRegions

6.1.1 Detailed Description

Container for storing Delaunay (duel) graphs.

Author

Jeffrey R. Peters

6.1.2 Constructor & Destructor Documentation

6.1.2.1 AreaCon::DelaunayGraph::DelaunayGraph (const int *NRegions*)

Constructor.

Parameters

in	*NRegions*	The number of regions in the partition or the number of nodes in the Delaunay graph

6.1.2.2 AreaCon::DelaunayGraph::DelaunayGraph (const **DelaunayGraph** & *obj*)

Copy Constructor

Parameters

in	*obj*	Element to by copied

6.1.2.3 AreaCon::DelaunayGraph::~DelaunayGraph ( )

Destructor. Cleans up dynamically allocated memory.

6.1.3 Member Function Documentation

6.1.3.1 DelaunayGraph & AreaCon::DelaunayGraph::operator= (const DelaunayGraph & *obj*)

Copy Assignment Operator

Parameters

in	*obj*	Element to by copied

6.1.4 Member Data Documentation

6.1.4.1 Point*** AreaCon::DelaunayGraph::Graph

A multi-dimensional array whose (i,j)-th entry holds the endpoints of the line segment that is shared by region i and region j. If the two regions do not share a common edge, then at least 1 index of the (i,j)-th entry will equal INFINITY.

6.1.4.2 const int AreaCon::DelaunayGraph::NRegions

The number of regions under consideration.

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

- areacon.h
- areacon.cpp

6.2 AreaCon::Density Class Reference

`#include <areacon.h>`

Public Member Functions

- void SetNewRegion (const Poly Region, const int Nx=0, const int Ny=0, const std::vector< double > Values={})
- void SetParameters (const int Nx, const int Ny, const std::vector< double > Values)
- int GetNx (void) const
- int GetNy (void) const
- Poly GetRegion (void) const
- std::vector< bool > GetGridInRegion (void) const
- Int_Params GetIntegral (void) const
- void GetExtrema (double &minx, double &miny, double &maxx, double &maxy) const
- double LineIntegral (double spacing, const Point &p1, const Point &p2) const
- double CalculateWeightedArea (const Poly Region) const

- Point CalculateCentroid (const Poly Region, const double &Volume) const
- void SetVolumeLowerBound (const double VolumeLowerBound)
- double GetVolumeLowerBound (void)

- Density ()
- Density (const Poly Region, const int Nx=0, const int Ny=0, const std::vector< double > Values={})

Private Member Functions

- void CheckParameterSizes (void)
- void SetExtrema (void)
- void Setdxy (void)
- void PreprocessIntegral (void)
- void CreateIntegralCoefficients (void)
- double CreateIntegralVector (void)
- void NormalizeIntegralVector (const double &Total)
- double InterpolateValue (const Point &Test) const
- Point ConvertIndextoWorld (const int ii) const

Private Attributes

- Poly Region
- int Nx
- int Ny
- double dx
- double dy
- double minx
- double miny
- double maxx
- double maxy
- double Volume_Lower_Bound
- std::vector< double > Values
- std::vector< bool > GridInRegion
- Int_Params Integral

6.2.1 Detailed Description

The base class for defining probability density functions.

Author

Jeffrey R. Peters

6.2.2 Constructor & Destructor Documentation

6.2.2.1 AreaCon::Density::Density ( )

Default Constructor.

6.2.2.2 AreaCon::Density::Density (const Poly *Region*, const int *Nx* = 0, const int *Ny* = 0, const std::vector< double > *Values* = { })

Parameters

in	*Region*	The (convex) polygonal region of interest.
in	*Nx*	The number of grid points in the x direction.
in	*Ny*	The number of grid points in the y direction.
in	*Values*	A vector containing the value of the density function at the grid-point locations (the value at the (i,j)-th grid point is stored in the (Ny∗i+j)-th entry of Values.

6.2.3 Member Function Documentation

6.2.3.1 Point AreaCon::Density::CalculateCentroid (const Poly *Region,* const double & *Volume*) const

Calculates the centroid of the polygon Region with respect to the density

Parameters

in	*Region*	The polygon of interest
in	*Volume*	The total volume of the region in question

Returns

The location of the centroid

6.2.3.2 double AreaCon::Density::CalculateWeightedArea (const Poly *Region*) const

Evaluates the integral of the density over the polygon Region

Parameters

in	*Region*	The polygon over which the integral is evaluated

Returns

The weighted area of the region

6.2.3.3 void AreaCon::Density::CheckParameterSizes (void) [private]

A function that checks consistency of parameter sizes.

6.2.3.4 Point AreaCon::Density::ConvertIndextoWorld (const int *ii*) const [private]

Returns the world coordinates of the grid-point associated with the ii-th entry of the vector Values.

Returns

The world coordinates of the associated grid point.

6.2.3.5 void AreaCon::Density::CreateIntegralCoefficients (void) [private]

Creates the coefficients that are used in numerically evaluating integrals. Results are stored in the associated Int_Params container Integral

6.2.3.6 double AreaCon::Density::CreateIntegralVector (void) [private]

Pre-calculates and stores the value of relevant integrals over individual grid squares. Results are stored in the associated Int_Params container Integral

Returns

The value of the total integral of the density under the region of interest.

6.2.3.7 void AreaCon::Density::GetExtrema (double & *minx,* double & *miny,* double & *maxx,* double & *maxy*) const [inline]

Returns the extreme x and y values.

Parameters

out	*minx,miny,maxx,maxy*	

6.2.3.8 std::vector<bool> AreaCon::Density::GetGridInRegion (void) const [inline]

Returns

GridInRegion

6.2.3.9 Int_Params AreaCon::Density::GetIntegral (void) const [inline]

Returns

Integral

6.2.3.10 int AreaCon::Density::GetNx (void) const [inline]

Returns

Nx

6.2.3.11 int AreaCon::Density::GetNy (void) const [inline]

Returns

Ny

6.2.3.12 Poly AreaCon::Density::GetRegion (void) const [inline]

Returns

Region

6.2.3.13 double AreaCon::Density::GetVolumeLowerBound (void)

Returns the lower volume bound.

Returns

Volume_Lower_Bound;

6.2.3.14 double AreaCon::Density::InterpolateValue (const Point & *Test*) const [private]

Uses interpolation to find the value of the density at the point Test, which is not necessarily a grid point.

Parameters

in	*Test*	The point of interest.

Returns

The value of the density function at Test

6.2.3.15 double AreaCon::Density::LineIntegral (double *spacing,* const Point & *p1,* const Point & *p2*) const

Calculates the line integral of the density function over the straight line connecting points p1, p2

Parameters

in	*spacing*	The spacing between evaluation points
in	*p1,p2*	The endpoints of the line in question

Returns

The value of the line integral

6.2.3.16 void AreaCon::Density::NormalizeIntegralVector (const double & *Total*) [private]

Makes sure that the values stored in the integral vector are normalized so that their sum is equal to 1 over the region of interest.

Parameters

in	*Total*	The value of the total integral of the density under the region of interest.

6.2.3.17 void AreaCon::Density::PreprocessIntegral (void) [private]

Performs pre-processing steps that are necessary for fast integration.

6.2.3.18 void AreaCon::Density::Setdxy (void) [private]

Calculates the value of dx and dy.

6.2.3.19 void AreaCon::Density::SetExtrema (void) [private]

Finds and sets the extrema for the polygon Region, i.e., sets minx, maxx, miny, maxy.

6.2.3.20 void AreaCon::Density::SetNewRegion (const Poly *Region,* const int *Nx* = 0, const int *Ny* = 0, const std::vector< double > *Values* = { })

Function used to set a new polygonal region of interest.

Parameters

in	*Region*	The (convex) polygonal region of interest.
in	*Nx*	The number of grid points in the x direction.
in	*Ny*	The number of grid points in the y direction.
in	*Values*	A vector containing the value of the density function at the grid-point locations (the value at the (i,j)-th grid point is stored in the (Ny∗i+j)-th entry of Values.

6.2.3.21 void AreaCon::Density::SetParameters (const int *Nx*, const int *Ny*, const std::vector< double > *Values*)

Function used to re-set the values of the grid-parameters.

Parameters

in	*Nx*	The number of grid points in the x direction.
in	*Ny*	The number of grid points in the y direction.
in	*Values*	A vector containing the value of the density function at the grid-point locations (the value at the (i,j)-th grid point is stored in the (Ny∗i+j)-th entry of Values.

6.2.3.22 void AreaCon::Density::SetVolumeLowerBound (const double *VolumeLowerBound*)

Sets the lower volume bound (default = 0). This bound is used to avoid numerical instability in partition calculations.

Parameters

in	*VolumeLower-Bound*	The new bound value;

6.2.4 Member Data Documentation

6.2.4.1 double AreaCon::Density::dx [private]

The grid spacing: dx = maxx-minx/(Nx-1)

6.2.4.2 double AreaCon::Density::dy [private]

The grid spacing: dy = maxy-miny/(Ny-1)

6.2.4.3 std::vector<bool> AreaCon::Density::GridInRegion [private]

A vector whose entries indicate whether or not grid points lie within Region. If the (i,j)-th grid point lies within the polygonal region of interest, then the (Ny∗i+j)-th entry of GridInRegion is true, otherwise it is false.

6.2.4.4 Int_Params AreaCon::Density::Integral [private]

Container that holds parameters relevant to quickly calculating area integrals.

6.2.4.5 double AreaCon::Density::maxx [private]

The maximum x coordinate of the polygon

6.2.4.6 double AreaCon::Density::maxy [private]

The maximum y coordinate of the polygon

6.2.4.7 double AreaCon::Density::minx [private]

The minimum x coordinate of the polygon

6.2.4.8 double AreaCon::Density::miny [private]

The minimum y coordinate of the polygon

6.2.4.9 int AreaCon::Density::Nx [private]

The number of grid points in the x direction

6.2.4.10 int AreaCon::Density::Ny [private]

The number of grid points in the y direction

6.2.4.11 Poly AreaCon::Density::Region [private]

The region of interest

6.2.4.12 std::vector<double> AreaCon::Density::Values [private]

A vector containing the value of the density function at the grid-point locations (the value at the (i,j)-th grid point is stored in the (Ny∗i+j)-th entry of Values.

6.2.4.13 double AreaCon::Density::Volume_Lower_Bound [private]

A lower bound on any calculated volume (default = 0). This parameter is used to avoid numerical instability in partition calculations.

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

- areacon.h
- areacon.cpp

6.3 AreaCon::Int_Params Class Reference

`#include <areacon.h>`

Public Member Functions

- void CheckParameters (void) const
- Int_Params (std::vector< double > Coefficient_a={}, std::vector< double > Coefficient_b={}, std::vector< double > Coefficient_c={}, std::vector< double > Coefficient_d={}, std::vector< double > Int={}, std::vector< double > Intx={}, std::vector< double > Inty={}, double UnweightedArea=0)

Public Attributes

- std::vector< double > Coefficient_a
- std::vector< double > Coefficient_b
- std::vector< double > Coefficient_c
- std::vector< double > Coefficient_d
- std::vector< double > Int
- std::vector< double > Intx
- std::vector< double > Inty
- double Unweighted_Area

6.3.1 Detailed Description

A container for values used in quickly calculating area integrals over polygonal regions of interest.

Author

Jeffrey R. Peters

6.3.2 Constructor & Destructor Documentation

6.3.2.1 AreaCon::Int_Params::Int_Params (std::vector< double > *Coefficient_a* = { }, std::vector< double > *Coefficient_b* = { }, std::vector< double > *Coefficient_c* = { }, std::vector< double > *Coefficient_d* = { }, std::vector< double > *Int* = { }, std::vector< double > *Intx* = { }, std::vector< double > *Inty* = { }, double *UnweightedArea* = 0)

6.3.3 Member Function Documentation

6.3.3.1 void AreaCon::Int_Params::CheckParameters (void) const

Checks to make sure that the user-input parameters satisfy the required bounds

6.3.4 Member Data Documentation

6.3.4.1 std::vector<double> AreaCon::Int_Params::Coefficient_a

6.3.4.2 std::vector<double> AreaCon::Int_Params::Coefficient_b

6.3.4.3 std::vector<double> AreaCon::Int_Params::Coefficient_c

6.3.4.4 std::vector<double> AreaCon::Int_Params::Coefficient_d

Coefficients used in quickly calculating area integrals. Usually populated as a part of the function Density.FindIntegralCoefficients.

6.3.4.5 std::vector<double> AreaCon::Int_Params::Int

6.3.4.6 std::vector<double> AreaCon::Int_Params::Intx

6.3.4.7 std::vector<double> AreaCon::Int_Params::Inty

Parameters representing area integrals over grid squares. Int represents the total integral, Intx represents the integral of x∗f(x,y), and Inty represents the integral of y∗f(x,y). Usually populated as a part of the function Density.FindIntegralVector.

6.3.4.8 double AreaCon::Int_Params::Unweighted_Area

The overall area of some polygonal region of interest

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

- areacon.h
- areacon.cpp

6.4 AreaCon::Mult_Array Class Reference

`#include <areacon.h>`

Public Member Functions

- Mult_Array (const int N)
- Mult_Array (const Mult_Array &obj)
- Mult_Array & operator= (const Mult_Array &obj)
- ~Mult_Array (void)

Public Attributes

- double ** Array
- const int N

6.4.1 Detailed Description

Container for storing two-dimensional arrays of size NxN.

Author

Jeffrey R. Peters

6.4.2 Constructor & Destructor Documentation

6.4.2.1 AreaCon::Mult_Array::Mult_Array (const int *N*)

Constructor

Parameters

in	*N*	The size of the array

6.4.2.2 AreaCon::Mult_Array::Mult_Array (const Mult_Array & *obj*)

Copy Constructor

Parameters

in	*obj*	Element to by copied

6.4.2.3 AreaCon::Mult_Array::∼Mult_Array (void)

Destructor. Cleans up dynamically allocated memory.

6.4.3 Member Function Documentation

6.4.3.1 Mult_Array & AreaCon::Mult_Array::operator= (const Mult_Array & *obj*)

Copy Assignment Operator

Parameters

in	*obj*	Element to by copied

6.4.4 Member Data Documentation

6.4.4.1 double∗∗ AreaCon::Mult_Array::Array

A two-dimensional array

6.4.4.2 const int AreaCon::Mult_Array::N

The size of the array.

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

- areacon.h
- areacon.cpp

6.5 AreaCon::Parameters Class Reference

`#include <areacon.h>`

Public Member Functions

- Parameters (const double line_int_step=0.1, const double weights_step=0.1, const double centers_step=1, const double volume_tolerance=0.002, const double convergence_criterion=0.02, const int max_iterations_volume=200, const int max_iterations_centers=500, const double Volume_Lower_Bound=10e-6, const double Robustness_Constant=10e-8)

Public Attributes

- const double line_int_step
- const double weights_step
- const double centers_step
- const double volume_tolerance
- const double convergence_criterion
- const int max_iterations_volume
- const int max_iterations_centers
- const double Volume_Lower_Bound
- const double Robustness_Constant

Private Member Functions

- void CheckParameters (void)

6.5.1 Constructor & Destructor Documentation

6.5.1.1 AreaCon::Parameters::Parameters (const double *line_int_step* = 0.1, const double *weights_step* = 0.1, const double *centers_step* = 1, const double *volume_tolerance* = 0.002, const double *convergence_criterion* = 0.02, const int *max_iterations_volume* = 200, const int *max_iterations_centers* = 500, const double *Volume_Lower_Bound* = 10e-6, const double *Robustness_Constant* = 10e-8)

Default Constructor. Default values are recommended to produce reasonable solutions with reasonable efficiency in most scenarios.

Parameters

in	*line_int_step*	Spacing parameter used for calculating line integrals
in	*weights_step*	Step-size used in updating the weighting parameters
in	*centers_step*	Step-size used in updating the center locations
in	*volume_tolerance*	Parameter used in determining whether desired volumes have been acheived
in	*convergence_criterion*	Parameter used as an algorithmic stopping criterion
in	*max_iterations_volume*	Upper bound on the number of volumetric iterations
in	*max_iterations_centers*	Upper bound on the number of centroidal movement iterations
in	*Volume_Lower_Bound*	A lower bound on the weighted area of each region
in	*Robustness_Constant*	A constant used to enhace numerical robustness (see Point class)

6.5.2 Member Function Documentation

6.5.2.1 void AreaCon::Parameters::CheckParameters (void) [private]

Checks to see if user-input parameters satisfy the required bounds

6.5.3 Member Data Documentation

6.5.3.1 const double AreaCon::Parameters::centers_step

Step-size used in updating the center locations

6.5.3.2 const double AreaCon::Parameters::convergence_criterion

Parameter used as an algorithmic stopping criterion

6.5.3.3 const double AreaCon::Parameters::line_int_step

Spacing parameter used for calculating line integrals

6.5.3.4 const int AreaCon::Parameters::max_iterations_centers

Upper bound on the number of centroidal movement iterations

6.5.3.5 const int AreaCon::Parameters::max_iterations_volume

Upper bound on the number of volumetric iterations

6.5.3.6 const double AreaCon::Parameters::Robustness_Constant

6.5.3.7 const double AreaCon::Parameters::Volume_Lower_Bound

6.5.3.8 const double AreaCon::Parameters::volume_tolerance

Parameter used in determining whether desired volumes have been acheived

6.5.3.9 const double AreaCon::Parameters::weights_step

Step-size used in updating the weighting parameters

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

- areacon.h
- areacon.cpp

6.6 AreaCon::Partition Class Reference

`#include <areacon.h>`

Public Member Functions

- Partition (int NRegions=0, Density Prior=Density(), std::vector< double > desired_area={}, Parameters Alg_Params=Parameters())

- void SetPartitionVariables (int NRegions=0, Density Prior=Density(), std::vector< double > desired_area={})
- void InitializePartition (std::vector< Point > Centers={}, std::vector< double > Weights={})
- std::vector< Poly > GetCovering (void)
- std::vector< Point > GetCenters (void)
- std::vector< double > GetWeights (void)
- void CalculatePartition (bool WriteToFile, std::string filename_partition="", std::string filename_centers="")

Private Member Functions

- void CheckParams (void)
- bool CreateDefaultCenters (const Poly Region, const double multiplier)
- void CreateDefaultCenters (const Poly Region, const double initial_multiplier, const int max_steps)
- bool CreatePowerDiagram (void)
- void CleanCovering (const double tolerance, const long int &mult)
- void CreateDelaunayGraph (DelaunayGraph &Delaunay) const
- double GradientStepCenter (const std::vector< double > &volumes)
- void GradientStepCenter (const double &temp_step, const std::vector< double > &volumes)
- void GradientStepWeights (const std::vector< double > &volumes, const DelaunayGraph &SharedEdges)
- double CalculateError (const std::vector< double > &volumes)
- std::vector< double > CalculateVolumes (void)

Private Attributes

- std::vector< Point > Centers
- std::vector< Poly > Covering
- std::vector< double > Weights

- const Parameters Alg_Params
- std::vector< double > desired_area
- Density Prior
- int NRegions

6.6.1 Detailed Description

The base class for defining a Partition.

Author

Jeffrey R. Peters

6.6.2 Constructor & Destructor Documentation

6.6.2.1 AreaCon::Partition::Partition (int *NRegions* = 0, Density *Prior* = Density (), std::vector< double > *desired_area* = {}, Parameters *Alg_Params* = Parameters ())

Constructors.

Parameters

in	*NRegions*	The number of regions desired.
in	*Prior*	The density function goverining partition creation
in	*desired_area*	A vector containing the desired areas of the regions in the resulting configuration. Defaults to equal area.
in	*Alg_Params*	Various algorithmic parameters

6.6.3 Member Function Documentation

6.6.3.1 double AreaCon::Partition::CalculateError (const std::vector< double > & *volumes*) [private]

Calculates a measure of volumetric error

Parameters

in	*volumes*	The volumes of the regions in Covering.

Returns

A measure of the volumetric error.

6.6.3.2 void AreaCon::Partition::CalculatePartition (bool *WriteToFile*, std::string *filename_partition* = " ", std::string *filename_centers* = " ")

The main function used for calculating partitions. Partitions are calculated and the resultant configuration is stored in the containers Centers and Covering. If WriteToFile = true, then the evolution of the centers and partitions will be written to the files filename_centers and filename_partitions, respectively.

Parameters

in	*WriteToFile*	Flag indicating if result should be written to file
in	*filename_partition*	Output filename to be written with partition data
in	*filename_centers*	Output filename to be written with center data

6.6.3.3 std::vector< double > AreaCon::Partition::CalculateVolumes (void) [private]

A function that calculates the volumes of the regions in Covering

Returns

A vector containing the volumes of the regions in Covering.

6.6.3.4 void AreaCon::Partition::CheckParams (void) [private]

Checks Parameters to ensure consistent sizes

6.6.3.5 void AreaCon::Partition::CleanCovering (const double *tolerance*, const long int & *mult*) [private]

A function used to eliminate redundancies in the covering that may arise due to numerical error.

Parameters

in	*tolerance*	A tolerance for determining whether distinct vertices should be combined into a single vertex
in	*mult*	A multiplier that affects the degree of numerical accuracy

6.6.3.6 bool AreaCon::Partition::CreateDefaultCenters (const Poly *Region*, const double *multiplier*) [private]

Creates default centers within the region defined by the polygon Region. The parameter multiplier is used to create center points which are ensured to lie within the polygon.

Parameters

in	*Region*	The region of interest
in	*multiplier*	A parameter used in center creation

Returns

A flag indicating whether centers were successfully created

6.6.3.7 void AreaCon::Partition::CreateDefaultCenters (const Poly *Region*, const double *initial_multiplier*, const int *max_steps*) [private]

Creates default centers within the region defined by the polygon Region. The parameter multiplier is used to create center points which are ensured to lie within the polygon.

Parameters

in	*Region*	The region of interest
in	*initial_multiplier*	A parameter used in center creation
in	*max_steps*	An upper bound on the number of center creation attempts

6.6.3.8 void AreaCon::Partition::CreateDelaunayGraph (DelaunayGraph & *Delaunay*) const [private]

Creates the Delaunay graph (Duel Graph) based on the partition stored in Covering.

Parameters

out	*Delaunay*	The Delaunay graph.

6.6.3.9 bool AreaCon::Partition::CreatePowerDiagram (void) [private]

Creates the power diagram generated from the current values of Centers and Weights.

6.6.3.10 std::vector<Point> AreaCon::Partition::GetCenters (void) [inline]

Returns

The current value of Centers

6.6.3.11 std::vector<Poly> AreaCon::Partition::GetCovering (void) [inline]

Returns

The current value of Covering

6.6.3.12 std::vector<double> AreaCon::Partition::GetWeights (void) [inline]

Returns

The current value of Weights

6.6.3.13 double AreaCon::Partition::GradientStepCenter (const std::vector< double > & *volumes*) [private]

Update the center locations based on the current configuration

Parameters

in	*volumes*	The current volumes of the regions in Covering

Returns

The sum of the squares of the Euclidean distance moved by each of the centers

6.6.3.14 void AreaCon::Partition::GradientStepCenter (const double & *temp_step,* const std::vector< double > & *volumes*) [private]

Update the center locations based on the current configuration (This function is used for the initial center step, and can speed convergence in cases where the center stepsize parameter is not chosen equal to 1).

Parameters

in	*temp_step*	A step-size that overwrites the value indicated in Alg_Params.
in	*volumes*	The current volumes of the regions in Covering

Returns

The sum of the squares of the Euclidean distance moved by each of the centers

6.6.3.15 void AreaCon::Partition::GradientStepWeights (const std::vector< double > & *volumes*, const DelaunayGraph & *SharedEdges*) [private]

Update the weights based on the current configuration.

Parameters

in	*volumes*	The current volumes of the regions in Covering
in	*SharedEdges*	The current Delaunay graph.

6.6.3.16 void AreaCon::Partition::InitializePartition (std::vector< Point > *Centers* = {}, std::vector< double > *Weights* = {})

Used to initialize algorithmic process variables Centers and Weights. If no input is given, default centers and weights are created.

Parameters

in	*Centers*	The initial center locations.
in	*Weights*	The initial weight values.

6.6.3.17 void AreaCon::Partition::SetPartitionVariables (int *NRegions* = 0, Density *Prior* = Density (), std::vector< double > *desired_area* = {})

Sets the partition variables.

Parameters

in	*NRegions*	The number of regions desired.
in	*Prior*	The density function goverining partition creation
in	*desired_area*	A vector containing the desired areas of the regions in the resulting configuration. Defaults to equal area.

6.6.4 Member Data Documentation

6.6.4.1 const Parameters AreaCon::Partition::Alg_Params [private]

Algorithmic parameters.

6.6.4.2 std::vector<Point> AreaCon::Partition::Centers [private]

The vector of center locations.

6.6.4.3 std::vector<Poly> AreaCon::Partition::Covering [private]

The covering of the area of interest.

6.6.4.4 std::vector<double> AreaCon::Partition::desired_area [private]

A vector specifying the desired areas of the resultant configurations.

6.6.4.5 int AreaCon::Partition::NRegions [private]

The number of regions desired.

6.6.4.6 Density AreaCon::Partition::Prior [private]

The prior probability density function.

6.6.4.7 std::vector<double> AreaCon::Partition::Weights [private]

The vector of weights associated with each area.

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

- areacon.h
- areacon.cpp

6.7 AreaCon::Point Class Reference

`#include <areacon.h>`

Public Member Functions

- Point (const double x=INFINITY, const double y=INFINITY)

- double Norm () const
- Point AddPoint (const Point Test) const
- Point FindPerpDirection (const Point Test, const double Norm) const
- void FlipDirection (void)
- double PerpDistanceToLine (const Point Test1, const Point Test2) const
- bool AreCollinear (const Point Test1, const Point Test2) const
- bool AreBetween (const Point Test1, const Point Test2) const
- void Mult (const double factor)

Static Public Member Functions

- static bool IsEqual (const Point Test1, const Point Test2)
- static double Distance (const Point Test1, const Point Test2)
- static Point FindPointAlongLine (const Point Test1, const Point Test2, const double distance)
- static double Norm (const Point Test)
- static Point AddPoints (const Point Test1, const Point Test2)
- static Point FindPerpDirection (const Point Test1, const Point Test2, const double Norm)
- static double PerpDistanceToLine (const Point Test1, const Point Test2, const Point Test3)
- static bool AreCollinear (const Point Test1, const Point Test2, const Point Test3)
- static bool AreBetween (const Point Test1, const Point Test2, const Point Test3)
- static std::vector< Point > FindCollinearIntersection (const Point p1, const Point p2, const Point p3, const Point p4)

Public Attributes

- double x
- double y

Static Public Attributes

- static double Robustness_Constant

6.7.1 Detailed Description

The base class for defining a point in the two-dimensional plane.

Author

Jeffrey R. Peters

6.7.2 Constructor & Destructor Documentation

6.7.2.1 AreaCon::Point::Point (const double *x* = INFINITY, const double *y* = INFINITY)

Default Constructor

Parameters

in	*x*	The x coordinate
in	*y*	The y coordinate

6.7.3 Member Function Documentation

6.7.3.1 Point AreaCon::Point::AddPoint (const Point *Test*) const

Adds the point Test to the current point component-wise

Returns

Sum = Point(Test.x+x, Test.y+y)

6.7.3.2 Point AreaCon::Point::AddPoints (const Point *Test1*, const Point *Test2*) [static]

Adds the points Test1, Test2 component-wise

Parameters

in	*Test1*	The first summand
in	*Test2*	The second summand

Returns

Sum of the points Test1, Test2

6.7.3.3 bool AreaCon::Point::AreBetween (const Point *Test1*, const Point *Test2*) const

Tests to see if the point object lies (numerically) on the line between Test1, Test2

Parameters

in	*Test1*	The first test point
in	*Test2*	The second test point

Returns

Indicator stating if point object is between the test points

6.7.3.4 bool AreaCon::Point::AreBetween (const Point *Test1,* const Point *Test2,* const Point *Test3*) [static]

Tests to see if Test3 lies (numerically) on the line between Test1, Test2

Parameters

in	*Test1*	The first test point
in	*Test2*	The second test point
in	*Test3*	The third test point

Returns

Indicator stating if point object is between the test points

6.7.3.5 bool AreaCon::Point::AreCollinear (const Point *Test1,* const Point *Test2*) const

Tests to see if the points Test1, Test2 are numerically collinear with the point object.

Parameters

in	*Test1*	The first test point
in	*Test2*	The second test point

Returns

Indicator of collinearity

6.7.3.6 bool AreaCon::Point::AreCollinear (const Point *Test1,* const Point *Test2,* const Point *Test3*) [static]

Tests to see if Test3 lies (numerically) on the line between Test1, Test2

Parameters

in	*Test1*	The first test point
in	*Test2*	The second test point
in	*Test3*	The third test point

Returns

Indicator stating if point object is between the test points

6.7.3.7 double AreaCon::Point::Distance (const Point *Test1,* const Point *Test2*) [static]

Finds the Euclidean distance between the points Test1, Test2.

Parameters

in	*Test1*	The first test point
in	*Test2*	The second test point

Returns

Euclidean distance

6.7.3.8 std::vector< Point > AreaCon::Point::FindCollinearIntersection (const Point *p1,* const Point *p2,* const Point *p3,* const Point *p4*) [static]

Returns the end-points of the line which forms the intersection of the lines connecting p1, p2 and p3, p4, respectively. Note that p1,p2,p3,p4 must be collinear. If the lines do not intersect, or only intersect at a single point, the return vector will contain less than 2 entries

Parameters

in	*p1*	The first end-point of the first line
in	*p2*	The second end-point of the first line
in	*p3*	The first end-point of the second line
in	*p4*	The second end-point of the second line

Returns

A vector containing the end-points of the intersection line;

6.7.3.9 Point AreaCon::Point::FindPerpDirection (const Point *Test,* const double *Norm*) const

Returns a point (vector) of length Norm that points in a direction perpendicular to the line between Test and the current point.

Parameters

in	*Test*	The test point
in	*Norm*	The desired norm of the resulting point (vector)

Returns

Perpendicular Point

6.7.3.10 Point AreaCon::Point::FindPerpDirection (const Point *Test1,* const Point *Test2,* const double *Norm*) [static]

Returns a point (vector) of length Norm that points in a direction perpendicular to the line between Test1 and Test2

Parameters

in	*Test1*	The first test point
in	*Test2*	The second test point
in	*Norm*	The desired norm of the resulting point (vector)

Returns

Perpendicular Point

6.7.3.11 Point AreaCon::Point::FindPointAlongLine (const Point *Test1,* const Point *Test2,* const double *distance*) [static]

Finds a point along the line connecting Test1, Test2 that is a normalized distance away from the point Test1, e.g, distance = 0.5 corresponds to the point that is exactly half-way between Test1, Test2.

Parameters

in	*Test1*	The first test point
in	*Test2*	The second test point
in	*distance*	The normalized distance along the line connecting Test1, Test2

Returns

The resulting point along the line

6.7.3.12 void AreaCon::Point::FlipDirection (void)

Reverses the orientation of the current point (vector), i.e., multiplies by -1

6.7.3.13 bool AreaCon::Point::IsEqual (const Point *Test1,* const Point *Test2*) [static]

Tests to see if Test1, Test2 are equal.

Parameters

in	*Test1*	The first test point
in	*Test2*	The second test point

Returns

Equality indicator

6.7.3.14 void AreaCon::Point::Mult (const double *factor*)

Multiplies the point object (vector) by a constant factor

Parameters

in	*factor*	The multiplication factor

6.7.3.15 double AreaCon::Point::Norm ( ) const

Calculates the Euclidean norm of the point

Returns

Euclidean norm

6.7.3.16 double AreaCon::Point::Norm (const Point *Test*) [static]

Finds the Euclidean norm of the point Test

Parameters

in	*Test*	The test point

Returns

The Euclidean distance

6.7.3.17 double AreaCon::Point::PerpDistanceToLine (const Point *Test1,* const Point *Test2*) const

Finds the perpendicular distance of the point to the line connecting Test1, Test2

Parameters

in	*Test1*	The first test point
in	*Test2*	The second test Point

Returns

The perpendicular distance

6.7.3.18 double AreaCon::Point::PerpDistanceToLine (const Point *Test1*, const Point *Test2*, const Point *Test3*) [static]

Finds the perpendicular distance of Test3 to the line connecting Test1, Test2

Parameters

in	*Test1*	The first point defining the line
in	*Test2*	The second point defining the line
in	*Test3*	The test point

Returns

The perpendicular distance

6.7.4 Member Data Documentation

6.7.4.1 double AreaCon::Point::Robustness_Constant [static]

A constant used to enhance numerical robustness. Loosely, when a Euclidean distance is less than the robustness constant, the distance is considered to be 0.

6.7.4.2 double AreaCon::Point::x

6.7.4.3 double AreaCon::Point::y

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

- areacon.h
- areacon.cpp

6.8 AreaCon::Poly Class Reference

`#include <areacon.h>`

Public Member Functions

- void SetVertices (const std::vector< Point > Vertices, const bool GetExtrema=true)
- std::vector< Point > GetVertices (void) const
- int GetNVertices (void) const
- bool pnpoly (const Point Test) const
- void GetExtrema (double &minx, double &miny, double &maxx, double &maxy) const

- Poly (std::vector< Point > Vertices={})

Private Member Functions

- void InitializePoly (void)

Private Attributes

- double minx
- double miny
- double maxx
- double maxy
- std::vector< Point > Vertices
- int NPoly

6.8.1 Detailed Description

The base class for defining (convex) polygons.

Author

Jeffrey R. Peters

6.8.2 Constructor & Destructor Documentation

6.8.2.1 AreaCon::Poly::Poly (std::vector< Point > *Vertices* = { })

Default Constructor

Parameters

in	*Vertices*	The points defining the vertices of the polygon in counter-clockwise order (the first vertex is not repeated).

6.8.3 Member Function Documentation

6.8.3.1 void AreaCon::Poly::GetExtrema (double & *minx*, double & *miny*, double & *maxx*, double & *maxy*) const

Returns the extreme x and y values.

Parameters

out	*minx,miny,maxx,maxy*	

6.8.3.2 int AreaCon::Poly::GetNVertices (void) const

Returns

The number of vertices.

6.8.3.3 std::vector< Point > AreaCon::Poly::GetVertices (void) const

Returns

The list of Vertices

6.8.3.4 void AreaCon::Poly::InitializePoly (void) [private]

A function used to initialize the polygon by calculating extrema and running checks for dimensional consistency.

6.8.3.5 bool AreaCon::Poly::pnpoly (const Point *Test*) const

Determines if the point Test lies within the polygon.

Parameters

in	*Test*	The test point

Returns

Indicator of whether or not Test is inside the polygon

6.8.3.6 void AreaCon::Poly::SetVertices (const std::vector< Point > *Vertices,* const bool *GetExtrema* = true)

A function used for setting the vertices of the polygon.

Parameters

in	*Vertices*	The points defining the vertices of the polygon in counter-clockwise order (the first vertex is not repeated).
in	*GetExtrema*	Flag that determines whether the extrema should be calculated and re-set

6.8.4 Member Data Documentation

6.8.4.1 double AreaCon::Poly::maxx [private]

The maximum x coordinate of the polygon

6.8.4.2 double AreaCon::Poly::maxy [private]

The maximum y coordinate of the polygon

6.8.4.3 double AreaCon::Poly::minx [private]

The minimum x coordinate of the polygon

6.8.4.4 double AreaCon::Poly::miny [private]

The minimum y coordinate of the polygon

6.8.4.5 int AreaCon::Poly::NPoly [private]

The number of vertices that the polygon has

6.8.4.6 std::vector<Point> AreaCon::Poly::Vertices [private]

The Vertices of the polygon

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

- areacon.h
- areacon.cpp

Chapter 7

File Documentation

7.1 areacon.cpp File Reference

```
#include "areacon.h"
```

Namespaces

- AreaCon

7.2 areacon.h File Reference

```
#include <stdio.h>
#include <cmath>
#include <algorithm>
#include <iostream>
#include <fstream>
#include "clipper.hpp"
```

Classes

- class AreaCon::Point
- class AreaCon::Poly
- class AreaCon::Mult_Array
- class AreaCon::DelaunayGraph
- class AreaCon::Int_Params
- class AreaCon::Parameters
- class AreaCon::Density
- class AreaCon::Partition

Namespaces

- AreaCon