Contents

Chapter 1

Introduction

The NSPACE statistical package is designed to solve problems in spatial statistics. The work is supported by The Research Council of Norway through the research program "Toolkits in Industrial mathematics",where the Norwegian Computing Center (NR) is one of three participants, the other two being SINTEF Oslo and the University of Oslo (UiO).

NSPACE contains routines for estimation of the parameters of the trend and variogram of a spatial model, optimal spatial prediction and simulation of stochastic fields. The routines are implemented in C++, and the NSPACE module is organized in class hierarchies so as to make it simple to add new models and methods. In the manual, private members, and protected members and member functions considered as implementation details, are in general not included in the documentations, if not needed for clarity.

The manual is organized in six chapters, including this introduction. Chapter two contains classes representing spatial point sets and data sets, and chapter three classes describing the parametric trend and covariance models considered. A class hierarchy for estimation of the spatial model parameters is documented in section four, and similar and related class hierarchies for spatial prediction and simulation in sections five and six. In the appendix, some classes and functions for internal usage in the library are documented.

Overviews of the class hierarchies for parameter estimation, prediction and simulation are given in the documentation of the three base classes in sections 4.1, 5.1 and 6.1.

The implementations make extensive use of classes representing vectors and matrices and a class hierarchy for least squares computations, included in the package NUTILITY. The random number generator used in the simulation hierarcy, is also a part of that module. The NUTILITY statistical package, version 2.0, is documented in Aldrin, M., Follestad, T., Helgeland, J., Orøy, O.E. and Andersen, T: "The NUTILITY statistical package. Version 2.0.", NR-note, STAT/15/95.

The NSPACE and NUTILITY packages make use of basic tools implemented in the module Diffpack, developed jointly at SINTEF Applied Mathematics and University of Oslo, Department of Mathematics and Department of Informatics. Diffpack is released as public access software, that is public domain software for non-commercial use.

Differences between versions 1.0 and 2.0.

In version 2.0, the version 2.0 of the classes in the matrix library of NUTILITY are included. The class hierarchies for the spatial data representation and the simulation methods are re-organized. A class hierarcy for parametric models for the semivariogram is introduced. A method for maximum likelihood estimation of the parameters of a spatial model, with or without prior information on the parameters, is included, as well as several new methods for simulation of spatial scalar- and vector-fields.

In addition, minor changes are done to several classes, and errors are corrected.

Chapter 2

Spatial data and point sets

2.1 Spatial data

2.1.1 SpatialData

NAME

SpatialData - an abstract base class for representations of a set of spatial data.

INCLUDE

```
include "SpatialData.h"
```

SYNTAX

```
class SpatialData: public DataSet, public virtual HandleId
{
 protected:
  int nobs;           // number of observations
  int nresp;          // number of response variables
  nMatrix Resp;       // nobs x nresp matrix of response values

 public:
  SpatialData (int np, int rd=1)
    : Resp(np,rd) { nobs = np; nresp = rd; }
  SpatialData (const nMatrix& R)
    : Resp(R) { nobs=R.getRdim(); nresp=R.getCdim(); }
  SpatialData (const SpatialData& sdata)
    : Resp(sdata.Resp) { nobs=sdata.nobs; nresp=sdata.nresp; }
  SpatialData ()
    : Resp() { nobs=0; nresp=0;}
  ~SpatialData () { };

  // redimensioning, the response values are set to zero:
  virtual BooLean redim(int nobs, int dim, int rd=1) = 0;

  // input and output of coordinates and response values:
  virtual int getSpaceDim () const = 0;        // spatial dimension
  virtual nMatrix getCoord () const = 0;       // coordinate matrix
  virtual nVector getCoord (int no) const = 0; // coordinates for obs. no. no
  int getNobs () const { return nobs; }        // number of observations
  int getNresp () const { return nresp; }      // number of response values
  nMatrix getResp () const { return Resp; }    // response values
  nVector getResp (int no) const { return Resp.row(no); }
                                               // response values for obs. no. no
  void setResp (const nMatrix& R);             // sets response values

  // range of coordinate values for dimension  d,
  // minc = minimum value, maxc = maximum value:
  virtual void coordRange (int d, double& minc, double& maxc) const = 0;

  // range of response values for response value no. d,
  // minr = minimum value, maxr = maximum value:
  void respRange (int rd, double& minr, double& maxr) const
    { minr = Resp.col(rd).minval(); maxr = Resp.col(rd).maxval();}

  virtual String typeId() const { return "SpatialData"; }

  // creates a copy of an instance of a derived class:
  SpatialData* createCopy() const;
};
```

KEYWORDS

data set, spatial data

DESCRIPTION

The class is a base class for representations of spatial data, containing a set of spatial points and one or more response variables. The class is derived from class `DataSet`. The class contains the response variables, which are stored in a `nMatrix` object, of dimension number of data locations by number of response values. The class is derived from class HandleId to simplify memory management by introducing *Handles* or *smart* pointers for this class object.

CONSTRUCTORS AND INITIALIZATION

The class has four constructors, including a copy constructor and a default constructor initializing the members to zero. An object might be initialized by giving the number of points, `np`, and the number of response values, `rd`, with default value one. The response values can then be initialized by using the `set`-function. The last constructor initializes the matrix of response values to the value of the `nMatrix` argument.

MEMBER FUNCTIONS

See the SYNTAX section.

FILES

SpatialData.C

EXAMPLE

See class IrregSpatialData and class GridData.

SEEALSO

class DataSet, class GridData, class IrregSpatialData

AUTHOR

Turid Follestad and Odd Egil Orøy, NR

2.1.2 GridData

NAME

GridData - a class for a set of data on a grid lattice

INCLUDE

```
include "SpatialData.h"
```

SYNTAX

```
class GridData: public SpatialData
{
 protected:
  RegGrid Grid;                       // grid lattice

  BooLean printCoord;                 // printing option
  void printWithCoord(Os out) const;  // prints data set
  void printWithGrid(Os out) const;   // prints data set

 public:
  GridData (int np, int d, int rd=1)
    : Grid(d), SpatialData(np, rd) { printCoord = dpFALSE; }
  GridData (const RegGrid& C, const nMatrix& R)
    : Grid(C), SpatialData(R){
      printCoord = dpFALSE;
      if (C.ngpoints() != R.getRdim())
        errorFP("GridData::GridData", "Number of coordinate points and "
                "response values unequal!\n"); }
  GridData (const GridData& gdata)
    : Grid(gdata.Grid), SpatialData(gdata) { printCoord = dpFALSE; }
  GridData ()
    : SpatialData() { printCoord = dpFALSE; };
  ~GridData () { };

  // redimensioning, setting the response values to zero:
  BooLean redim(int np, int d, int rd=1);

  BooLean ok ();          // returns dpTRUE if grid dim. and Resp not equal zero.
  void printCoordMatrix(BooLean b) { printCoord = b; } // sets printing option

  GridData& operator=(const GridData& sdata);

  // input and output:

  void setGrid (const RegGrid& C);    // sets grid lattice
  int getSpaceDim () const            // gets spatial dimension
    { return Grid.getDim(); }
  nMatrix getCoord () const           // gets coordinates as nobs by dim matrix
    { return Grid.coordMatrix(); }
  nVector getCoord (int no) const     // gets coordinate vector no. no
    { return Grid.coord(no); }
  RegGrid getGrid () const            // gets grid lattice
    { return Grid; }

  // range of coordinate values for dimension  d,
  // minc = minimum value, maxc = maximum value:
  void coordRange (int d, double& minc, double& maxc) const
    { minc=Grid.getMinval(d); maxc=Grid.getMaxval(d); }

  void cleanUp () { ;}                // cleanup, inherited from class DataSet
```

```
  String typeId() const { return "GridData"; }

  void scan (Is in);                  // reads from an input stream
  void print (Os out) const;          // prints to an output stream
};
```

KEYWORDS

data set, grid, grid lattice, spatial data

DESCRIPTION

The class is a representation of regularly spaced sets of spatial data, containing the specification of a grid lattice of spatial points, and one or more response variables. The response variables are stored in a matrix inherited from the base class `SpatialData`, and the grid specification in an object of type `RegGrid`.

The class is derived from class HandleId to simplify memory management by introducing *Handles* or *smart* pointers for this class object.

CONSTRUCTORS AND INITIALIZATION

The class has four constructors, including a copy constructor and a default constructor initializing all members to zero. An object might be initialized by giving the number of points, `np`, the spatial dimension, `d`, and the number of response values, `rd`, with default value one. The grid and response values can then be initialized by using the function `scan`, or the `set`-functions. The last constructor takes two arguments, a `RegGrid` object and a `np` by `rd` matrix of response values.

For the ordering of the reponse values to be in accordeance with the grid lattice definition, the matrix of response values should be ordered so that it corresponds to a coordinate matrix where the first coordinate dimension is first incremented:

```
Y(x(1,1,1,..)
Y(x(2,1,1,...)
    .
Y(x(nx,1,1,...)
Y(x(1,2,1,...)
Y(x(2,2,1,...)
    .
Y(x(nx,2,1,...)
    .
Y(x(nx,ny,1...)
Y(x(1,1,2,...)
Y(x(2,1,2,...)
    .
```

MEMBER FUNCTIONS

See also the SYNTAX section.

`print` - prints the contents of the object. The grid specification is printed first, followed by the matrix of response values. This is the format expected by the `scan`-function. Alternatively, if the member function `printCoordMatrix` is called with argument

dpTRUE, the coordinate matrix corresponding to the grid specification is computed, and the contents are printed with the same format as is done by the member function `print` in class `IrregSpatialData`.

The argument is of type class `Os`, from Diffpack, offering more flexibility than the standard `ostream` class. The actual arguments can be cout, an `ostream` object or a character string specifying a filename : "FILE=filename".

`printCoordMatrix` - see documentation of the member function `print`.

`scan` - reads the contents of the object from an input source. The argument of type class `Is` offers similar choices for input source as `Os` for output in `print`.

The input is expected to contain the grid specification formatted as explained in the documentation of the member function `scan` of class `RegGrid`, followed by the `np` by `rd` matrix of response values. The matrix should be organized so that it corresponds to a coordinate matrix where the coordinate value of the first spatial dimension is incremented first, then the second and so on.

FILES

GridData.C

EXAMPLE

```
#include <SpatialData.h>

main(int argc, char* argv[])
{
  int np = atoi(argv[1]);               // number of observations
  int d = atoi(argv[2]);                // spatial dimension

  // initialize data:

  GridData data(np,d);                  // regularly spaced data
  data.scan("FILE=griddata.dat");       // reads data from a file

  // range of coordinates for dimension 1:
  double cmin,cmax;
  data.coordRange(1,cmin,cmax);         // gets coordinate range

  // print to a file:

  if (data.ok()){
    data.printCoordMatrix(dpTRUE);      // coordinates to be printed as matrix
    data.print("FILE=outdata.dat");     // prints data set
  }
}
```

SEEALSO

class DataSet, class IrregSpatialData, class SpatialData

AUTHOR

Turid Follestad and Odd Egil Orøy, NR

2.1.3 IrregSpatialData

NAME

IrregSpatialData - a class for a set of irregularly spaced data

INCLUDE

```
include "SpatialData.h"
```

SYNTAX

```
class IrregSpatialData: public SpatialData
{
 protected:
  int dim;                          // number of spatial dimensions for the data
  nMatrix Coord;                    // nobs x dim matrix of coordinate values

 public:
  IrregSpatialData (int np, int d, int rd=1)
    : Coord(np,d), SpatialData(np, rd) { dim = d; }
  IrregSpatialData (const IrregSpatialData& sdata)
    : Coord(sdata.Coord), SpatialData(sdata) { dim = sdata.dim; }
  IrregSpatialData ()
    : Coord(), SpatialData() { dim=0; };
  IrregSpatialData (const nMatrix& D, int np, int d, int rd=1);
  IrregSpatialData (const nMatrix& C, const nMatrix& R);
  ~IrregSpatialData () { };

  IrregSpatialData& operator=(const IrregSpatialData& sdata);

  // redim. and set all coordinates and response values to zero:
  BooLean redim(int np, int d, int rd=1);

  // input and output of coordinate values:
  int getSpaceDim () const { return dim; }    // spatial dimension
  nMatrix getCoord () const { return Coord; } // nobs by dim coordinate matrix
  nVector getCoord (int no) const { return Coord.row(no); }
                                        // coordinate vector number  no.
  void setCoord (const nMatrix& C);           // sets nobs x dim coordinate matrix

  // range of coordinate values for dimension  d,
  // maxc = maximum value, minc = minimum value:
  void coordRange (int d, double& minc, double& maxc) const
    { minc = Coord.col(d).minval(); maxc = Coord.col(d).maxval();}

  BooLean ok ();       // returns dpTRUE if Coord and Resp both not equal zero.
  void cleanUp () { testDuplicate();}         // test for duplicate data locations

  // functions for editing the data:

  void extract(const nIntVector& ind, DataSet& ds) const;
  void extract(int from, int to, DataSet& ds) const;
  void remove (const nIntVector& ind);
  void remove (int from, int to);
  void insert (const DataSet&, int from);

  String typeId() const { return "IrregSpatialData"; }

  void scan (Is in);                        // reads from an input stream
  void print (Os out) const;                // prints to an output stream
};
```

KEYWORDS

data set, spatial data

DESCRIPTION

The class is a representation of a general set of spatial data, containing the coordinates of a set of spatial points and one or more response variables. The coordinates and the response variables are stored in two matrices, the response values in the base class `SpatialData`. The class is derived from class HandleId to simplify memory management by introducing *Handles* or *smart* pointers for this class object.

CONSTRUCTORS AND INITIALIZATION

The class has five constructors, including a copy constructor and a default constructor initializing all members to zero. An object might be initialized by giving the number of data locations, `np`, the spatial dimension, `d`, and the number of response values, `rd`, with default value one. The coordinates and response values can then be initialized by using the function `scan`, or the `set`-functions. The other two constructors take the coordinates and response values as arguments, in two matrices or a single matrix. When using one matrix, the first `d` columns should contain the coordinate values, and the following columns the response values.

MEMBER FUNCTIONS

See also the SYNTAX section.

`cleanUp` - tests for duplicate data locations. If duplicates, that is points with similar coordinate values, are found, a warning is issued, and a file called "Duplicatefile" is made. This file contains the indices of the duplicate points.

In the following functions for editing the data, inherited from the base class `DataSet`, the actual `DataSet` reference arguments are expected to be of type class `IrregSpatialData`.

`extract` - extracts observations indexed by `ind`, or from `from` to `to` (observations number `from` and `to` are included), returning a new `DataSet`.

`remove` - removes observations indexed by `ind`, or from `from` to `to` (observations number `from` and `to` are included).

`insert` - inserts data set after observation number `from-1`.

`scan` - reads the coordinates and response values from an input stream. The first `d` columns are expected to contain the coordinate values, and the following `rd` columns the corresponding response values. If the number of points on the input stream is greater than the initialized value, only the number of lines corresponding to the specified dimensionality of the member matrices are read.

`print` - prints the coordinate values and the response values in the same order as explained above for the `scan` function.

FILES

IrregSpatialData.C

EXAMPLE

```
#include <SpatialData.h>

main(int argc, char* argv[])
{
  int np = atoi(argv[1]);              // number of observations
  int dim = atoi(argv[2]);             // spatial dimension
  int nresp = atoi(argv[3]);           // number of response variables

  // data set:

  IrregSpatialData testd(np,dim,nresp); // data set object
  ifstream inf("dataset.dat",ios::in);
  testd.scan(inf);                     // reads data from a file
  inf.close();

  // cleanup:

  testd.cleanUp();                     // removes duplicate data locations
  np = testd.getNobs();                // number of data locations after cleanup

  // removing the first and last observations:
  nIntVector iv(2);                    // initializes vector of indices
  iv(1) = 1;                           // first observation
  iv(2) = np;                          // last observation

  testd.remove(iv);                    // removes indicated data
  np = testd.getNobs();                // new number of data locations

  // extraction and insertion:

  // extracts first half of the spatial data set:
  IrregSpatialData testd2(np/2,dim,nresp);
  testd.extract(1,np/2,testd2);

  // adds observations to  testd, after the last of the existing obervations:
  testd.insert(testd2,np+1);

  testd.print("FILE=testdata.dat");     // prints the new dataset to a file
}
```

SEEALSO

class DataSet, class RegGrid, class SpatialData

AUTHOR

Turid Follestad and Odd Egil Orøy, NR

2.2 Point sets

2.2.1 PointSet

NAME

PointSet - a base class for sets of spatial points

INCLUDE

```
include "PointSet.h"
```

SYNTAX

```
class PointSet : public virtual HandleId
{
 protected:
  int n;                                  // number of points
  int dim;                                // spatial dimension

 public:
  PointSet(int np, int d) { n = np; dim = d;}
  PointSet() { n = 0; dim = 0;}
  virtual ~PointSet() { };

  // creating a copy of an instance of a derived class:
  PointSet* createCopy () const;

  // redim. of number of points and spatial dimension:
  virtual BooLean redim(int np, int dim) = 0;

  int getNpoints() const { return n; }    // gets number of points
  int getDim() const { return dim;}       // gets spatial dimension
  virtual char* typeId() const { return "PointSet";}

  virtual nVector coord(int i) const = 0;
  virtual nMatrix coordMatrix() const = 0;

  // functions for iteration over points:
  virtual nVector firstPoint() = 0;       // resets iteration, returns first point
  virtual nVector nextPoint() = 0;        // returns next point

  virtual void scan(Is is) = 0;           // reads from an input stream
  virtual void print(Os os) const = 0;    // prints to an output stream
};
```

KEYWORDS

coordinates, point set, spatial data

DESCRIPTION

The class is an abstract base class for representations of sets of spatial points. The class is derived from class HandleId to simplify memory management by introducing *Handles* or *smart* pointers for this class object.

CONSTRUCTORS AND INITIALIZATION

The class has one constructor that takes as arguments the number of points and the spatial dimension, and a default constructor initializing the members to zero.

MEMBER FUNCTIONS

See also the SYNTAX section.

`coord` - returns a vector of the coordinates of point number `i`. The numbering of the points will depend on the representation in the derived classes.

`coordMatrix` - returns a matrix of coordinates of all the points in the point set.

FILES

PointSet.C

EXAMPLE

```
#include <PointSet.h>

main(int argc, char* argv[])
{
  char* type = argv[1];              // type of point set, "grid" or "gen"
  int np, dim=2;                     // number of points and spatial dimension

  // initialize PointSet object:
  PointSet* pset;
  if (!strncmp(type,"grid",4)){      // grid lattice
    pset = new RegGrid(dim);
    pset->scan("FILE=grid.dat");
    np = pset->getNpoints();
  }
  else if (!strncmp(type,"gen",3)){  // irregularly spaced points
    pset = new GenPointSet;
    np = countNumbers("gen.points"); // counts number of entries on the file
    np = np/dim;                     // number of coordinate points
    pset->redim(np,dim);             // resets number of points and dimension
    pset->scan("FILE=gen.points");   // reads form file 'gen.points'
  }

  // print coordinate matrix to file 'coord.matr':
  pset->coordMatrix().print("FILE=coord.matr");
}
```

SEEALSO

class GenPointSet, class RegGrid

AUTHOR

Turid Follestad, NR

2.2.2 GenPointSet

NAME

GenPointSet - a class for a set of irregularly spaced points

INCLUDE

```
include "PointSet.h"
```

SYNTAX

```
class GenPointSet : public PointSet
{
 private:
  nMatrix C;                            // matrix of coordinates
  int current;                          // current point index
 public:
  GenPointSet(const GenPointSet& Ps)
    :C(Ps.C), PointSet(Ps.getNpoints(), Ps.getDim()) { }
  GenPointSet(const nMatrix& Pm)
    :C(Pm), PointSet(Pm.getRdim(),Pm.getCdim()) { }
  GenPointSet(int np, int d)
    :C(np,d),PointSet(np,d) { }
  GenPointSet()
    :C(),PointSet() { }

  char* typeId() const { return "GenPointSet";}

  // resets the number of points and the spatial dimension, reallocates coordinate
  // matrix and sets all coordinate values to zero:
  BooLean redim(int np ,int dim);

  void setCoord(const nMatrix& Pm) { C.setEqual(Pm);} // sets coordinate matrix

  nVector coord (int i) const           // gets coordinate vector for point no. i
    { return C.row(i); }
  nMatrix coordMatrix () const          // gets coordinate matrix
    { return C; }

  // functions for iteration over all points:

  nVector firstPoint() { current = 0; return nextPoint();}  // resets iteration
  nVector nextPoint() { current+=1; return coord(current);} // gets next point

  void scan(Is in) { C.scan(in);}           // reads from input stream
  void print(Os out) const { C.print(out);} // prints to output stream
};
```

KEYWORDS

coordinates, point set, spatial points

DESCRIPTION

The class is derived from the base class `PointSet`, and represents a general set of spatial points. The points are stored in a `np` by `d` matrix, where `np` is the number of points and

`d` the dimension of the point space. Each row represents the set of `d` coordinates for one point.

CONSTRUCTORS AND INITIALIZATION

The class has a copy constructor, a default constructor and two other constructors. One constructor takes two integer arguments, the number of points `np` and the dimension `d`. The other constructor has one argument: a `np` by `d` matrix of coordinates.

MEMBER FUNCTIONS

See also the SYNTAX section.

`scan` - reads from an input stream, assuming that the dimensions of the input matrix equals the number of points and the spatial dimension of the object. If the dimensions differ, the function `redim` should be called on beforehand.

`setCoord` - resets the coordinate matrix to `Pm`. It is assumed that the number of rows and columns of `Pm` corresponds to the number of points and the spatial dimension of the `GenPointSet` object. If not, the member function `redim` sould be called prior to this function.

FILES

PointSet.C

EXAMPLE

See class PointSet.

SEEALSO

class PointSet, class RegGrid

AUTHOR

Turid Follestad, NR

2.2.3 RegGrid

NAME

RegGrid - a class for a set of points on a grid lattice

INCLUDE

```
include "PointSet.h"
```

SYNTAX

```
class RegGrid : public PointSet
{
 protected:
  nIntVector ndiv;          // number of divisions in each dimension
  nVector delta;            // grid spacing in each dimension
  nVector minval;           // minimum coordinate values in each dimension
  nVector maxval;           // maximum coordinate values in each dimension

  nIntVector current;       // current point while iterating over points

  BooLean redim (int np, int gdim); // issues a warning and calls redim(int)

 public:
  RegGrid (int gdim);
  RegGrid (int gdim, const nMatrix& Gpar);
  RegGrid (const RegGrid& rgrid);
  RegGrid ();

  BooLean redim (int gdim); // redimensions to spatial dimension  gdim

  // set grid specification parameters, i = dimension, val = value:

  void setMinval (int i, double val) { minval(i) = val;}
  void setMaxval (int i, double val) { maxval(i) = val;}
  void setNdiv (int i, int val)
    { ndiv(i) = val; delta(i) = (maxval(i) - minval(i))/ndiv(i);
      n = ngpoints();}
  void setDelta (int i, double val)
    { delta(i) = val; ndiv(i) = int((maxval(i)-minval(i))/delta(i));
      n = ngpoints();}

  // get grid specification parameters,  i = dimension:

  int getNdiv (int i)      const { return ndiv(i);}
  double getDelta (int i)  const { return delta(i);}
  nVector getMinval ()     const { return minval;}
  double getMinval (int i) const { return minval(i);}
  nVector getMaxval ()     const { return maxval;}
  double getMaxval (int i) const { return maxval(i);}

  int ngpoints () const;    // returns the total number of grid points
  char* typeId() const { return "RegGrid";}

  // get coordinate vectors:

  nVector coord (nIntVector index) const;
  nVector coord (int no) const;
  nMatrix coordMatrix() const;  // coordinate matrix

  nVector nextPoint ();     // returns coordinates of next point
```

```
  nVector firstPoint ()     // resets iteration, returns coordinates of next point
    { for (int i=1;i<=dim;i++) current(i)=0; return nextPoint();}

  void index (int no, nIntVector& ind) const;
                            // transforms from  obs.no. to index-vector
  int obsNumber (const nIntVector& ind) const;
                            // transforms from index-vector to obs.no.
  void scan (Is is);
  void print (Os os) const;

  friend class GridData;
};
```

KEYWORDS

coordinates, grid, grid lattice, point set, spatial points

DESCRIPTION

The class is a representation of a set of spatial points on a grid lattice, and is derived from the abstract base class `PointSet`. The grid is specified by the spatial dimension, the grid spacing, or alternatively the number of divisions, and the maximum and minimum coordinate values of each dimension.

CONSTRUCTORS AND INITIALIZATION

The class has a copy constructor, a default constructor and two other constructors. When initializing the object by one integer, the spatial dimension, the object must be further initialized by reading the specifiction from an input stream by `scan`, or by using the member functions `setMinval`, `setMaxval` and `setNdiv` or `setDelta`. The last two functions assume that the members `minval` and `maxval` are already initialized. By calling `setNdiv`, the member `delta` will be computed using the current values of `minval` and `maxval`, and similarly, `ndiv` is computed by `setDelta`.

When initializing a `RegGrid` object by using the default constructor, the member `redim` must be called prior to the `set`-functions.

Alternatively, the data members may be initialized by calling the constructor with a `nMatrix` reference argument in addition to the dimension. The matrix must be arranged as follows:

```
First column  - number of divisions in each dimension

Second column - minimum values of each dimension

Third column  - maximum values of each dimension
```

The member `delta` will in this case be computed by the constructor.

MEMBER FUNCTIONS

See also the SYNTAX section.

`set`-functions - see the CONSTRUCTORS AND INITIALIZATION section.

`coord (nIntVector ind)` - returns the coordinate vector corresponding to the index vector `ind`, specifying the grid index for each dimension.

`coord (int no)` - returns the coordinate vector corresponding to point number `no`. The points are numbered so that the first index is incremented first, then the second and so on.

`coordMatrix` - returns a matrix of the coordinates of all points.

`nextPoint` - returns the coordinates of the next point. This functions is intended for use when iterating over all points.

`firstPoint` - resets the iteration and returns the first point.

`index` - transforms from point number, `no`, to index-vector `ind`.

`obsNumber` - returns point number according to the argument index vector.

`print` - prints the specification of the grid. The argument is of type class `Os`, offering more flexibility than the standard `ostream` class. The actual arguments can be cout, an `ostream` object or a character string specifying a filename : "FILE=filename".

`scan` - reads the specification of the grid. The argument of type class `Is` offers similar choices for input source as `Os` for output in `print`. The input ought to be formatted as in the following example, specifying a 14 by 16 grid with coordinate ranges [0,1]:

Grid: d=2 domain = [0,1]x[0,1] grid points = [1:14]x[1:16]

If the spatial dimension `d` is already initialized by a call to the constructor, the dimension will be changed to the value of `d` read from the input source.

FILES

RegGrid.C

EXAMPLE

```
#include <PointSet.h>

main()
{
  int dim = 2;                // spatial dimension
  RegGrid grid(dim);          // grid object

  nIntVector divisions(dim);  // number of divisions
  nVector maximum(dim);       // maximum values
  nVector minimum(dim);       // minimum values

  // grid lattice with 9 divisions (10 points) for both dimensions
  // and with minimum value 0 and maximum value 1 in both dimensions:
  for (int i=1;i<=dim;i++){
    divisions(i) = 9;
    maximum(i) = 1.0;
    minimum(i) = 0.0;
  }

  // initialize grid:
  for (i=1;i<=dim;i++){
    grid.setMaxval(i,maximum(i));
    grid.setMinval(i,minimum(i));
    grid.setNdiv(i,divisions(i));
         // based on the minimum and maximum values as initialized above
   }
```

```
  // compute and print coordinate matrix:
  nMatrix Pcoord = grid.coordMatrix();
  Pcoord.print("FILE=coord.matr");
}
```

SEEALSO

class PointSet, class GenPointSet

AUTHOR

Turid Follestad, NR

Chapter 3

Spatial model specification

3.1 Covariance and semivariogram models for one-dimensional fields

3.1.1 VarStructure

NAME

VarStructure - a class for a parametric semivariogram model

INCLUDE

```
include "VarStructure.h"
```

SYNTAX

```
class VarStructure : public virtual HandleId
{
 protected:
  Handle(PrmSVModel) model;            // parametric model
  nVector param;                       // parameters

  nVector upperbound;                  // upper bounds for parameter values
  nVector lowerbound;                  // lower bounds for parameter values

 public:
  VarStructure (const PrmSVModel& mod, int pn=3);
  VarStructure (char* mod, int pn=3);
  VarStructure (const VarStructure& cov);
  VarStructure ()
    :param(), upperbound(), lowerbound() { ;}

  virtual ~VarStructure () { };

  VarStructure& operator= (const VarStructure& vmodel);

  BooLean redim (int pn);              // redim. of parameter vector

  // get parameters and model specification:

  nVector getParam () const { return param;}
  double getParam (int i) const { return param(i);}
  int getNparam() const { return param.getLen();}
  char* getModel () const { return model->typeId(); }
  double getRange ()  const;           // returns practical range

  // returns upper and lower bounds for parameter i:
  double getUpper(int i) const {return upperbound(i);}
  double getLower(int i) const {return lowerbound(i);}

  // set parameter values and model specification:

  void setParam (const nVector& pval) { param.setEqual(pval);}
  void setParam (int i, double prm) { param(i) = prm;}

  void setModel(char* mod);            // changes parametric model
  void setRange(double pval);          // sets practical range

  // set upper and lower bounds for parameter i:
  void setUpper(int i, double upp) { upperbound(i) = upp;}
```

```
  void setLower(int i, double low) { lowerbound(i) = low;}

  BooLean ok () const;                 // returns dpFALSE if model==NULL

  // covariance and semivariogram values:

  double covValue (const nVector& x1, const nVector& x2) const;
  double covValue (double dist) const;

  double semivValue (const nVector& x1, const nVector& x2) const;
  double semivValue (double dist) const;

  // derivatives with respect to the range:

  double corrDeriv1 (double h);
  double corrDeriv2 (double h);

  // covariance matrix:
  void covarMatrix (SymMatrix& Covm, const PointSet& ps) const;
};
```

KEYWORDS

covariance, covariogram, semivariogram, structure of variation, variogram

DESCRIPTION

The class is a representation of the structure of variation of a spatial field, holding information about the type of parametric model and the parameter values. The parametric model is represented by a pointer to an object in the `PrmSVModel`-hierarchy. The parameters for most of the models implemented in this hierarchy, are the nugget effect, the partial sill (variance - nugget effect) and the practical range.

The class is derived from class HandleId to simplify memory management by introducing *Handles* or *smart* pointers for this class object.

CONSTRUCTORS AND INITIALIZATION

There are four constructors, including a copy constructor and a default constructor. By default, all parameters are initialized to zero. By using one of the other two constructors, the parametric model can be specified. Five models are implemented in the library, and a model is chosen by entering its name as a character string argument. Possible choices are:

`exponential`, `generalexp`, `spherical`, `gaussian`, `bessel` and `power`.

`Bessel` refers to a parametric model based on modified Bessel functions (Abrahamsen (1994), p.44), and `generalexp` to the exponential family of models, with default model equal to the "gaussian" type model.

For a definition and description of the different models, see Cressie (1991), pp.61 and Abrahamsen (1994), pp.42.

The models are implemented as subclasses derived from the class `PrmSVModel`, and a `VarStructure` object can alternatively be initialized by speciying an object of one of these classes. If the user wants to implement other models than the five listed above, a new, user defined subclass of `PrmSVModel` can be implemented, and an object of this class can be used to initialize the `VarStructure` object.

The parameter values might be set by the `set`-functions, or estimated by one of the classes in the `SpatialModel` hierarchy.

The members `lowerbound` and `upperbound` specify the lower and upper bounds for the parameters. These are needed when estimating the model parameters. They are initialized by zero and inifinity (1e300). Their values can be modified by the `setUpper` and `setLower` functions prior to estimation of the parameters.

REFERENCES:

Abrahamsen, P.: "A Review of Gaussian Random Fields and Correlation Functions", NR-Report no.878, 1994.

Cressie, N.: "Statistics for Spatial Data", Wiley, New York, 1991.

MEMBER FUNCTIONS

See also the SYNTAX section.

`redim` - redimensions the parameter vector. Should be followed by a call to `setModel` to reset the type of parametric model.

`covValue` - overloaded function that computes the covariance between values of a stochastic field in two spatial points, `x1` and `x2`, or between two points with intermediate Euclidean distance `dist`.

`semivValue` - overloaded function that computes the semivariogram values of a stochastic field for lag `dist`, or for the lag between the two points `x1` and `x2`.

`covarMatrix` - computes the covariance matrix for the points represented by the `PointSet` object entered as an argument, returning the result through the `SymMatrix` reference argument.

`corrDeriv1` - computes the first derivative of the parmetric correlation function with respect to the range, for two points with intermediate distance `h`.

`corrDeriv2` - computes the second derivative of the parmetric correlation function with respect to the range, for two points with intermediate distance `h`.

FILES

VarStructure.C

EXAMPLE

```
#include <VarStructure.h>
#include <PointSet.h>

main()
{
  int npoints = 50;              // number of points
  int dim = 2;                   // spatial dimension
  char* vmtype = "bessel";       // type of parmetric model

  VarStructure var(vmtype);      // variation structure object
  nVector param(3);
  param.scan("FILE=param.dat");  // reads the model parameters from a file
  var.setParam(param);           // sets the model parameters

  GenPointSet pset(npoints,dim);
  pset.scan("FILE=gen.points");  // reads a set of points from a file
```

```
  SymMatrix C(npoints);          // covariance matrix object
  var.covarMatrix(C,pset);       // computes the covariance matrix

  C.print("FILE=covar.matr");    // prints the covariance matrix to a file
}
```

SEEALSO

class MLSpatMod, class PrmSVModel, class SemivarModel

AUTHOR

Turid Follestad, NR

3.1.2 VecSimplest(VarStructure)

NAME

VecSimplest(VarStructure) - a very simple vector of VarStructure objects

INCLUDE

```
include "VecSimplest_VarStructure.h"
```

SYNTAX

```
#define Type VarStructure
#include <VecSimplest.h>
#undef Type
```

KEYWORDS

structure of variation, vector

DESCRIPTION

`VecSimplest(VarStructure)` is a class implementing a very simple vector of `VarStructure` objects, and is a specification of the parametric class `VecSimplest(Type)` in Diffpack, with parameter `Type` equal `VarStructure`. The index base is 1. The only operation available is subscripting.

See documentation of the parametric class `VecSimplest(Type)` implemented in Diffpack for a description of the interface of the class.

SEEALSO

class VarStructure, class VecSimplest(Type)

3.1.3 PrmSVModel

NAME

PrmSVModel - an abstract base class for parametric semivariogram and covariance models

INCLUDE

```
include "PrmSVModel.h"
```

SYNTAX

```
class PrmSVModel : public virtual HandleId
{
 public:
  PrmSVModel () { ;}
  virtual ~PrmSVModel () { ;}

  virtual double corrValue (double h, double range) const = 0;
  virtual double covValue (double h, const nVector& param) const;
  virtual double semivValue (double h, double range) const = 0;
  virtual double semivValue (double h, const nVector& param) const;

  virtual double corrDeriv1 (double h, double range) const;
  virtual double corrDeriv2 (double h, double range) const;

  virtual char* typeId() const { return "PrmSVModel"; }
};
```

KEYWORDS

correlation, correlogram, covariance, model, parameters, semivariogram, variogram

DESCRIPTION

The class is an abstract base class for definitions of isotropic parametric models for the covariance and semivariogram of a stochastic field. For the majority of the models implemented in the class hierarchy, second order stationarity is assumed, and for these models, the covariance and semivariogram functions are on the form

```
C(h) = nug * delta(h) + psill * rho(h;R),        (1)
```

and

```
gamma(h) = nug + psill * (1 - rho(h;R)).         (2)
```

Here, `nug` is the nugget effect, `psill` is the partial sill, `h` is the radial distance and `R` the (practical) range. The function `delta(h)` is the delta function and `rho(h;R)` the correlation function.

If the second order stationarity requirement is not met, the covariance function (1) do not exist, and the relationship in (2) is not valid. For an intrinsic stationary process, the semivariogram is in general defined as

```
gamma(h) = nug + c1 * g(h;c2),                   (3)
```

with parameters `nug`, `c1` and `c2`. The function `g(h;c2)` must satisfy some conditions, for (3) to be a valid semivariogram model. See Cressie (1991), pp.86, for further details. In the class hierarchy, the `power` model is of this type.

A new model can be defined by implementing the virtual member functions in a derived class.

The models implemented in the current version are (model names used in the program in paranthesis):

```
modified Bessel function model ("bessel"),
gaussian type model ("gaussian"),
power model ("power"),
exponential model ("exponential"),
general exponential model ("generalexp") and
spherical model ("spherical").
```

The class is derived from class HandleId to simplify memory management by introducing *Handles* or *smart* pointers for this class object.

REFERENCE:

Cressie, N.: "Statistics for Spatial Data", Wiley, New York, 1991.

CONSTRUCTORS AND INITIALIZATION

The class has no data members, and thus only a default constructor.

An object of a derived class can be created as a pointer to class `PrmSVModel`, by using the function `createPrmSVModel`.

MEMBER FUNCTIONS

GENERAL REMARK:

For all member functions, it is assumed that the range parameter is non-zero. This requirement is NOT checked for in the functions. It is assumed that the validity of the parameters is checked for before invoking any member function of this class.

`corrDeriv1` - function that is to return the first derivative of the correlation function `rho(h;R)`, if the model is second order stationary, with respect to the range parameter, `R`. By default, a warning is issued, stating that the correlation function for the model is not differentiable.

`corrDeriv2` - function that is to return the second derivative of the correlation function `rho(h;R)`, if the model is second order stationary, with respect to the range parameter, `R`. By default, a warning is issued, stating that the correlation function for the model is not twice differentiable.

`corrValue` - is to compute the correlation function `rho(h;R)` for points at radial distance `h`.

`covValue` - computes the covariance for points at radial distance `h`, and with model parameters `param`. The parameter vector is expected to be ordered as follows:

```
param(1) - nugget effect,
param(2) - partial sill,
param(3) - range,
```

and the covariance is computed by (1), computing `rho(h;R)` by calling the member function `corrValue`.

`semivValue` - overloaded function that computes the semivariogram values of a stochastic process for lag `h`, for a model with parameter vector `param`. By the default implementation, the semivariogram is computed by (3), and the parameter vector `param` is expected to be ordered as follows:

```
param(1) - nug,
param(2) - c1,
param(3) - c2.
```

For a second order stationary process, `c1` equals the partial sill and `c2` the range. If the nugget effect is zero and the parameter `c1` equal to one, the function requiring only one parameter, `c2` (the range of a second order stationary process) might be used. This virtual function is to compute the function `g(h;c2)` in (3), with the argument `range` equal to `c2`.

FILES

PrmSVModel.C

EXAMPLE

```
#include <PrmSVModel.h>

main(int argc, char* argv[])
{
  double param;                 // parameter: range, or power for general
                                // exponential model
  double exppower = 1.5;        // power for general exponential model
  double power = 1.1;           // power for power model
  char* mtype;                  // type of parametric model

  mtype = argv[1];              // read model

  double maxlag = 1.0;          // maximum lag
  int nlags = 10;               // number of lags
  double range = 0.3;           // range

  // initialize parametric model:
  PrmSVModel* model;
  param = range;                // parameter = range parameter

  if (!strncmp(mtype,"bessel",6))
    model = new BessPrmSVModel; // default order: 2
  else if (!strncmp(mtype,"gaussian",8))
    model = new GaussPrmSVModel;
  else if (!strncmp(mtype,"exponential",11))
    model = new ExpPrmSVModel;
  else if (!strncmp(mtype,"generalexp",10))
    model = new GenExpPrmSVModel(exppower);
  else if (!strncmp(mtype,"spherical",9))
    model = new SphPrmSVModel;
  else if (!strncmp(mtype,"power",5)){
    model = new PowPrmSVModel;
    param = power;
  }
  else
    model = NULL;
```

```
  nVector corrval(nlags), semival(nlags);
  double h;

  if (model!=NULL){
    for (int i=1;i<=nlags;i++){
      h = (i-1)*maxlag/(nlags-1);
      corrval(i) = model->corrValue(h,param);
                          // if power model: a warning message is issued
      semival(i) = model->semivValue(h,param);
    }
    cout << "Correlations:\n";
    corrval.print(cout);
    cout << endl;
    cout << "Semivariogram values:\n";
    semival.print(cout);
    cout << endl;
  }
  else
    cout << "Illegal parametric model! No values computed.\n";
}
```

SEEALSO

class VarStructure

AUTHOR

Turid Follestad, NR

3.1.4 BessPrmSVModel

NAME

BessPrmSVModel - a class for a parametric semivariogram and covariance model based on modified Bessel functions

INCLUDE

```
include "PrmSVModel.h"
```

SYNTAX

```
class BessPrmSVModel : public PrmSVModel
{
 private:
  void scaleBessel (double& z); // finds scaling factor
  void initOrder (double bo);   // initializes, order = bo

 protected:
  double gconst,                // constant, depending on  order
         order,                 // order of modified bessel function
         rfactor;               // scaling factor: corr(distance=range)=0.05

 public:
  BessPrmSVModel (double ord=2.0);

  double corrValue (double h, double range) const;
  virtual double semivValue (double h, double range) const;

  double corrDeriv1 (double h, double range) const;
  double corrDeriv2 (double h, double range) const;

  char* typeId() const { return "BessPrmSVModel"; }
};
```

KEYWORDS

correlation, correlogram, covariance, model, semivariogram, variogram, Bessel functions

DESCRIPTION

The class defines a parametric model for the covariance and semivariogram of a stochastic field, based on modified Bessel functions (Abrahamsen (1994), p.44). The correlation function `rho(h;R)`, specified in the documentation of the abstract base class `PrmSVModel`, is defined, in terms of the variable

```
z = S*h/R,
```

as

```
rho(z) = gconst * pow(z,n) * K (z).
                              n
```

Here, `gconst` is a constant, depending on the Bessel function order, `n`, `pow` is the power function and `K` is the modified Bessel function of order `n`. Further, `R` is the range, and `S` is a scaling factor, depending on `n`, determined so that `rho(R;R)` = 0.05. For `n` equal 2, `gconst` equals 0.5 and `S` is approximately 5.37.

REFERENCE:

Abrahamsen, P.: "A Review of Gaussian Random Fields and Correlation Functions", NR-Report no.878, 1994.

CONSTRUCTORS AND INITIALIZATION

The class has one constructor, taking the order of the modified Bessel function as the only argument. The current version accepts order 2 only.

MEMBER FUNCTIONS

See the documentation of class `PrmSVModel`.

FILES

PrmSVModel.C

EXAMPLE

See class PrmSVModel.

SEEALSO

class PrmSVModel, class VarStructure

AUTHOR

Turid Follestad, NR

3.1.5 ExpPrmSVModel

NAME

ExpPrmSVModel - a class for an exponential semivariogram and covariance model

INCLUDE

```
include "PrmSVModel.h"
```

SYNTAX

```
class ExpPrmSVModel : public PrmSVModel
{
 public:
  ExpPrmSVModel ()
    :PrmSVModel () { ;}

  double corrValue (double h, double range) const;
  virtual double semivValue (double h, double range) const;

  double corrDeriv1 (double h, double range) const;
  double corrDeriv2 (double h, double range) const;

  char* typeId() const { return "ExpPrmSVModel"; }
};
```

KEYWORDS

correlation, correlogram, covariance, model, variogram, semivariogram

DESCRIPTION

The class specifies an exponential parametric model for the covariance and semivariogram of a stochastic field. The correlation function `rho(h;R)`, specified in the documentation of the abstract base class `PrmSVModel`, is defined as

```
rho(h;R) = exp(-3*h/R).
```

Here, `R` is the range of the correlation function. The scaling factor, equal to 3, is computed so that `rho(R;R)` $= 0.05$.

CONSTRUCTORS AND INITIALIZATION

The class has no data members, and thus a default constructor only.

MEMBER FUNCTIONS

See the documentation of class `PrmSVModel`.

FILES

PrmSVModel.C

EXAMPLE

See class PrmSVModel.

SEEALSO

class PrmSVModel, class VarStructure

AUTHOR

Turid Follestad, NR

3.1.6 GaussPrmSVModel

NAME

GaussPrmSVModel - a class for a "gaussian" type semivariogram and covariance model

INCLUDE

```
include "PrmSVModel.h"
```

SYNTAX

```
class GaussPrmSVModel : public PrmSVModel
{
 public:
  GaussPrmSVModel ()
    :PrmSVModel () { ;}

  double corrValue (double h, double range) const;
  virtual double semivValue (double h, double range) const;

  double corrDeriv1 (double h, double range) const;
  double corrDeriv2 (double h, double range) const;

  char* typeId() const { return "GaussPrmSVModel"; }
};
```

KEYWORDS

correlogram, correlation, covariance, model, variogram, semivariogram

DESCRIPTION

The class defines a gaussian parametric model for the covariance and semivariogram of a stochastic field. The correlation function `rho(h;R)`, specified in the documentation of the abstract base class `PrmSVModel`, is defined as

$$\texttt{rho(h;R)} = \exp(-3*(h/R)^2).$$

Here, `R` is the range of the correlation function. The scaling factor, equal to 3, is computed so that `rho(R;R)` $= 0.05$.

CONSTRUCTORS AND INITIALIZATION

The class has no data members, and thus a default constructor only.

MEMBER FUNCTIONS

See the documentation of class `PrmSVModel`.

FILES

PrmSVModel.C

EXAMPLE

See class PrmSVModel.

SEEALSO

class PrmSVModel, class VarStructure

AUTHOR

Turid Follestad, NR

3.1.7 GenExpPrmSVModel

NAME

GenExpPrmSVModel - a class for the class of exponential semivariogram and covariance models

INCLUDE

```
include "PrmSVModel.h"
```

SYNTAX

```
class GenExpPrmSVModel : public PrmSVModel
{
  double order;        // power of exponential function

 public:
  GenExpPrmSVModel (double ord = 2.0)
    :PrmSVModel () { order = ord; }

  double corrValue (double h, double range) const;
  virtual double semivValue (double h, double range) const;

  double corrDeriv1 (double h, double range) const;
  double corrDeriv2 (double h, double range) const;

  char* typeId() const { return "GenExpPrmSVModel"; }
};
```

KEYWORDS

correlation, correlogram, covariance, model, variogram, semivariogram

DESCRIPTION

The class defines the class of exponential parametric models for the covariance and semivariogram of a stochastic field. The correlation function `rho(h;R)`, specified in the documentation of the abstract base class `PrmSVModel`, is defined as

$$\texttt{rho(h;R)} = \exp(-3*(h/R)^{p})$$

where the power `p` is an additional parameter. This parameter is specified at the construction of an object of this class. `R` is the range of the correlation function. The scaling factor, equal to 3, is computed so that `rho(R;R)` $= 0.05$.

CONSTRUCTORS AND INITIALIZATION

The class has one constructor, taking the power of the exponential function as argument. By default, the power is 2, and the model is similar to the model of class `GaussPrmSVModel`.

MEMBER FUNCTIONS

See the documentation of class `PrmSVModel`.

FILES

PrmSVModel.C

EXAMPLE

See class PrmSVModel.

SEEALSO

class PrmSVModel, class VarStructure

AUTHOR

Turid Follestad, NR

3.1.8 PowPrmSVModel

NAME

PowPrmSVModel - a class for power (including linear) semivariogram models

INCLUDE

```
include "PrmSVModel.h"
```

SYNTAX

```
class PowPrmSVModel : public PrmSVModel
{
 public:
  PowPrmSVModel ()
    :PrmSVModel () { ;}

  double corrValue (double h, double pow) const;
  virtual double semivValue (double h, double pow) const;

  char* typeId() const { return "PowPrmSVModel"; }
};
```

KEYWORDS

model, variogram, semivariogram

DESCRIPTION

The class defines power parametric models for the semivariogram of a stochastic field. The semivariogram model is defined as

$$\texttt{gamma(h)} = \texttt{nug} + \texttt{c1} * \texttt{h}^{\texttt{p}} \qquad (1)$$

The power `p` should be in the range [0,2>, for the model to be a valid semivariogram model. The semivariogram is computed by the definition (3) in the documentation of the abstract base class `PrmSVModel`.

CONSTRUCTORS AND INITIALIZATION

The class has no data members, and thus a default constructor only.

MEMBER FUNCTIONS

See also the documentation of class `PrmSVModel`.

`corrValue` - since the power model is not second order stationary, the correlation function do not exist, and this function will issue a warning message and return 0.0.

`semivValue` - computes the semvariogram value (1) for lag `h` for a model with power `pow`, and with nugget effect `nug` = 0.0 and `c1` = 1.0.

FILES

PrmSVModel.C

EXAMPLE

See class PrmSVModel.

SEEALSO

class PrmSVModel, class VarStructure

AUTHOR

Turid Follestad, NR

3.1.9 SphPrmSVModel

NAME

SphPrmSVModel - a class for a spherical semivariogram and covariance model

INCLUDE

```
include "PrmSVModel.h"
```

SYNTAX

```
class SphPrmSVModel : public PrmSVModel
{
 public:
  SphPrmSVModel ()
    :PrmSVModel () { ;}

  double corrValue (double h, double range) const;
  virtual double semivValue (double h, double range) const;

  double corrDeriv1 (double h, double range) const;
  double corrDeriv2 (double h, double range) const;

  char* typeId() const { return "SphPrmSVModel"; }
};
```

KEYWORDS

correlation, correlogram, covariance, model, variogram, semivariogram

DESCRIPTION

The class defines a spherical parametric model for the covariance and semivariogram of a stochastic field. The correlation function `rho(h;R)`, specified in the documentation of the abstract base class `PrmSVModel`, is defined as

```
                                        3
rho(h;R) = (1-1.5*h/R+0.5*(h/R) );  h<R,

rho(h;R) =  0;                      h>=R.
```

Here, `R` is the range of the correlation function.

CONSTRUCTORS AND INITIALIZATION

The class has no data members, and thus a default constructor only.

MEMBER FUNCTIONS

See the documentation of class `PrmSVModel`.

FILES

PrmSVModel.C

EXAMPLE

See class PrmSVModel.

SEEALSO

class PrmSVModel, class VarStructure

AUTHOR

Turid Follestad, NR

3.1.10 NonParSemivar

NAME

NonParSemivar - a class for representation and estimation of a non-parametric semivariogram.

INCLUDE

```
include "NonParSemivar.h"
```

SYNTAX

```
class NonParSemivar : public virtual HandleId
{
 protected:
  int nlags;           // number of lag values
  double lagsize;      // size of the lag interval
  nIntVector nval;     // number of pairs of data points in each lag interval
  nVector values;      // estimated semivariogram values
  String estType;      // classic or robust estimation

  Handle(SpatialData) sdata;  // coordinates and residuals

  BooLean estimated;   // dpTRUE if semivariogram is estimated
  void estimateRobust (nVector&, nIntVector&, int nvar=1);
                       // estimates semivariogram, robust
  void estimateClassic (nVector&, nIntVector&, int nvar=1);
                       // estimates semivariogram, classic

 public:
  NonParSemivar (const SpatialData& sd, double lagsz=0);
  NonParSemivar (const NonParSemivar& npsv);
  NonParSemivar ();
  ~NonParSemivar () { ;}

  // get data members:

  int getNofLags () const { return nlags; }       // number of lags
  double getLagsize () const { return lagsize; }  // lagsize
  nIntVector getNvalues () const { return nval; } // no. of pairs of points
  nVector getSvalues () const { return values; }  // semivariogram values
  void getData(SpatialData& sd) const             // data
    { sd = *(sdata.getPtr());}

  // set data members:

  void setLagsize (int lsz);                      // resets lagsize and nlags
  void resetData (const SpatialData& sd);         // resets spatial data set

  // reset and get estimation method (robust or classic):

  void setEstType(const String& str);
  String getEstType() const { return estType; }

  int findNofLags(int np);                        // computes number of lags

  // estimation of semivariogram and cross-semivariogram:

  void estimate (int nvar=1);
  void estimateCrossSv (int nvar1=1, int nvar2=2);
```

```
  BooLean ok();           // returns dpTRUE if object is properly initialized,
                          // and estimated == dpTRUE
  void print (Os ostr) const;                    // prints semivariogram

  friend class SemivarModel;
};
```

KEYWORDS

classical semivariogram, non-parametric semivariogram, robust semivariogram, semivariogram, variogram

DESCRIPTION

The class is a representation of the non-parametric semivariogram for a set of spatial data, where the spatial data are assumed to be realizations of an intrinsic stationary stochastic process. The spatial data represented by the `SpatialData` member object, are thus assumed to be realizations of a stochastic process having a constant trend, or residuals from a non-stationary process.

The class is used by the classes in the `SpatialModel`- and `SpatialPred`- hierarchies for solving spatial estimation- and prediction-problems.

The values of the semvariogram are estimated by the function `estimate`, and stored as a member object of the class. Two methods are implemented. These are the classical estimator, defined as

$$2*sv(hk) = \frac{1}{N(hk)} * \sum_{|xi-xj| \text{ in lag } hk} [\, (Z(xi)-Z(xj))^2 \,]$$

and the robust estimator suggested in Cressie (1991), p.75:

$$2*sv(hk) = \left[\frac{1}{N(hk)} * \sum_{|xi-xj| \text{ in lag } hk} (\sqrt{|Z(xi)-Z(xj)|})^4\right] /(0.457 + 0.494/N(hk))$$

Here, `Z` is the stochastic field, `xi` is the coordinate vector of point `i`, `hk` is lag number `k` and `N(hk)` is the number of pairs of points with intermediate distace in lag `hk`. A distance `d` is defined to be in lag `hk` if `d` is in the interval `[hk-1,hk>`.

By default, the robust estimator is computed, and to compute the classical estimator, the member function `estType` should be called with a `String` argument equal to "classic".

The class is derived from class HandleId to simplify memory management by introducing *Handles* or *smart* pointers for this class object.

REFERENCE:

Cressie, N.: "Statistics for Spatial Data", Wiley, New York, 1991.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors, including a copy constructor and a default constructor initializing the members to zero. The third constructor has two arguments, a `SpatialData` object, representing the data for which the variogram is to be estimated, and a double value specifying the lagsize of the variogram. If the default value for the lagsize is used, the function `findNofLags` will be invoked, and the lagsize computed from the number of lags returned from that routine.

The data and the lagsize may be reset by the functions `resetData` and `setLasgsize`.

MEMBER FUNCTIONS

`setLagsize` - resets the lagsize to the value of the actual argument, computes the corresponding value of `nlags` and performs necessary redimensioning of the member objects.

`estType` - resets the indicator of what estimator to be computed. A `String` argument of "classic" yields the classical estimator, and "robust" the robust estimator.

`findNofLags` - computes the number of lags as a function of the number of data points. The function is called when no value (other than zero) is specified for the lagsize in the constructor. The number of lags for a set of `n` observations, is computed by

```
int(n*log(n)/6);   n > 45

20;                n <=45.
```

`resetData` - resets the data for which the semivariogram is to be computed. The response values are usually residuals from trend surface estimation, and this function is used when estimating the trend and variogram parameters by iteration.

Note that the address of the member object is set equal to the address of the input object.

`estimate` - estimates the semivariogram values `values` and computes the corresponding value of `nval`, the number of pairs of points used at the different lags. If there are several response values, an integer argument can be supplied to get estimated values for response values other than the first one, which is the default. If the value of the integer argument is set to zero, the mean value of the semivariograms of all the response variables are computed, an option that might be useful when dealing with simulated data.

`estimateCrossSv` - estimates the cross-semivariogram, Var(Z1-Z2), for two response values Z1 and Z2; which ones can be indicated through the integer arguments. The semivariogram is estimated using the robust estimator.

`print` - prints the estimated semivariogram to an output stream. The output is ordered in three columns: the lag, the semivariogram values and the number of pairs of points in the lag interval.

FILES

NonParSemivar.C

EXAMPLE

```
#include <NonParSemivar.h>

main(int argc, char* argv[])
{
  int dim=atoi(argv[1]);
  int np=atoi(argv[2]);
  int nresp=atoi(argv[3]);

  IrregSpatialData data(np,dim,nresp); // irregularly spaced data
  data.scan("FILE=test.data");         // reads data

  double lagsize = 0.1;                // sets lagsize
  NonParSemivar sv(data,lagsize);      // initializes NonParSemivar object

  ofstream resf("sv.dat",ios::out);    // output stream

  // estimation of semivariogram for all response variables, by using
  // the robust semivariogram model (default):
  for (int i=1;i<=nresp;i++){
    sv.estimate(i);
    sv.print(resf);
  }

  // computing residuals:
  nVector trendval(np);
  trendval.scan("FILE=trend.val");     // reads trend values
  nMatrix Y(data.getResp());
  Y.setCol(1,Y.col(1)-trendval);       // computes residuals

  data.setResp(Y);                     // sets response variables equal residuals
  sv.resetData(data);                  // resets data of NonParSemivar object
  sv.estimate();                       // estimates for the default variable 1

  cout << "Semivariogram values for residuals for variable 1: \n"
    << sv.getSvalues() << "\n";
}
```

SEE ALSO

class SemivarModel, class VarStructure

AUTHOR

Turid Follestad, NR

3.2 Covariance models for multi-dimensional fields

3.2.1 VecFieldVar

NAME

VecFieldVar - an abstract base class for the parametric covariance structure for stochastic vector fields

INCLUDE

```
include "VecFieldVar.h"
```

SYNTAX

```
class VecFieldVar : public virtual HandleId
{
 protected:
  int veclen;                 // dimensionality of the vector field
 public:
  VecFieldVar(int l) { veclen = l;}
  VecFieldVar() { veclen = 0;}
  virtual ~VecFieldVar() { ;}

  int getVeclen() const { return veclen; }
                              // returns the vector field dimension

  virtual nMatrix covValue(const nVector& x1, const nVector& x2) const = 0;
  virtual void covarMatrix(SymMatrix& Cm, const PointSet& pset) const;
};
```

KEYWORDS

covariance, stochastic field, vector field

DESCRIPTION

The class is an abstract base class for representations of the parametric covariance structure for stochastic vector fields. The class is derived from class HandleId to simplify memory management by introducing *Handles* or *smart* pointers for this class object.

CONSTRUCTORS AND INITIALIZATION

The class has two constructors, including a default constructor. The integer argument specifies the dimensionality of the vector field. By default, this variable is set to zero.

MEMBER FUNCTIONS

`covValue` - computes the covariance and cross-covariance between the points `x1` and `x2`, returning the result as a matrix, not necessarily symmetric.

`covarMatrix` - computes the covariance matrix for the points represented by the `PointSet` argument, returning the result through the `SymMatrix` reference argument. The covariance matrix is ordered as follows:

```
covX1X1    covX1X2       .          .   covX1Xm
covX2X1    covX2X2       .          .   covX2Xm
   .          .          .          .      .
   .          .          .          .      .
   .          .          .          .      .
covXmX1    covXmX2       .          .   covXmXm
```

where `covXiXj` is the cross-covariance matrix for vector field dimensions `i` and `j`.

FILES

VecFieldVar.C

EXAMPLE

See class GenVecFieldVar.

SEEALSO

class GenVecFieldVar, class VecCovFunc

AUTHOR

Turid Follestad, NR

3.2.2 GenVecFieldVar

NAME

GenVecFieldVar - a general class for representing the covariance structure of a two-dimensional stochastic vector field

INCLUDE

```
include "GenVecFieldVar.h"
```

SYNTAX

```
class GenVecFieldVar : public VecFieldVar
{
 private:
  VecCovFunc* cfunc;           // pointer to a parametric covariance model

 public:
  GenVecFieldVar(char* mod);
  GenVecFieldVar(VecCovFunc& vcf);
  GenVecFieldVar()
    : VecFieldVar() { cfunc = NULL;}
  GenVecFieldVar(GenVecFieldVar& gvfv);

  ~GenVecFieldVar() { if (cfunc!=NULL) delete cfunc; }

  // set parameters and parametric model:
  void setParam(const nVector& param);
  void setModel(char* mod);

  // get parameters:
  double getRange() const { return cfunc->getRange();}    // range
  double getNugget() const { return cfunc->getNugget();}  // nugget effect
  double getVar() const { return cfunc->getVar();}        // variance

  nMatrix covValue(const nVector& x1, const nVector& x2) const;
};
```

KEYWORDS

covariance, stochastic field, vector field

DESCRIPTION

The class, derived from class `VecFieldVar`, is a representation of the covariance structure of a vector field by a set of functions for the covariances of each vector component, and the cross-covariance. The only data member is a pointer to an object of class `VecCovFunc`.

In the current version, only two-dimensional vector fields are supported.

See also the documentation of class `VecCovFunc`.

CONSTRUCTORS AND INITIALIZATION

There are three constructors, including a default constructor. The only variable that needs to be specified, is what parametric covariance model is to be used. Two parametric models are implemented in the library, as subclasses of class `VecCovFunc`. These are the "gaussian" type model (class `GaussVecCovFunc`) and the modified Bessel function covariance model (class `BessVecCovFunc`). To initialize an object of class `GenVecFieldVar`, the name of the model ("gaussian" or "bessel") can be given. Alternatively, an object of one of the two classes derived from `VecCovFunc` can be given as argument to the constructor.

If the user wants to specify other models, a new subclass can be derived from `VecCovFunc`, and an object of this class can be used to initialize the `GenVecFieldVar` object.

MEMBER FUNCTIONS

See alse the SYNTAX section.

`covValue` - computes the covariance and cross-covariance values for the field between the points `x1` and `x2`, returning a `nvec` by `nvec` matrix, not necessarily symmatric. Here, `nvec` is the number of components of the vector field.

`setParam` - sets the parameter vector for the covariance model. The function invokes the member function `setParam` of class `VecCovFunc`, and the number of parameters will depend on which subclass of `VecCovFunc` that is specified.

`setModel` - changes the parametric model.

FILES

GenVecFieldVar.C

EXAMPLE

```
#include <GenVecFieldVar.h>
#include <BessVecCovFunc.h>
#include <GaussVecCovFunc.h>

main(int argc, char* argv[])
{
  char* model;
  model = argv[1];                 // type of covariance model
                                   // ("bessel" or "gaussian")
  // parameters:
  double variance = 1.0, range = 0.3;

  // coordinates:
  int dim = 2;                     // two-dimensional space
  int nvec = 2;                    // two-dimensional vector field

  char* coordfile = "coord.dat";
  int np = countNumbers(coordfile); // counts number of entries on the file
  np = np/dim;                     // number of data locations

  GenPointSet pset(np,dim);        // point set object
  pset.scan("FILE=coord.dat");     // reads coordinates from file 'coord.dat'

  // create an object of class GenVecFieldVar:

  GenVecFieldVar fieldvar(model);
```

```
  // set model parameters:

  nVector param(3);
  if (!strncmp(model,"bessel",6)){
    param(1) = variance;
    param(2) = range;
    param(3) = 2;                   // order of modified Bessel function
  }
  else{ // "gaussian"
    param.redim(2);
    param(1) = variance;
    param(2) = range;
  }
  fieldvar.setParam(param);

  // compute covariances:

  SymMatrix covMatr(np*nvec);  // covariance matrix
  fieldvar.covarMatrix(covMatr, pset);

  // print covariances to file 'covar.matr':
  covMatr.print("FILE=covar.matr");
}
```

SEEALSO

class VecCovFunc, class VecFieldVar

AUTHOR

Turid Follestad, NR

3.2.3 VecCovFunc

NAME

VecCovFunc - covariance functions for two-dimensional vector fields

INCLUDE

```
include "VecCovFunc.h"
```

SYNTAX

```
class VecCovFunc
{
 protected:
  double nugget;          // nugget effect
  double variance;        // variance (nugget + partial sill)
  double range;           // practical range
  double partial_sill;    // partial sill (variance - nugget)

 public:
  VecCovFunc()
    { range = 0.0; variance = 0.0; nugget = 0.0; partial_sill = 0.0; }
  VecCovFunc(double r, double var = 1.0, double nug = 0.0)
    { range = r; variance = var; nugget = nug; partial_sill = var - nug;}
  virtual ~VecCovFunc() { ;}

  double getRange() const { return range;}         // returns range
  double getVar() const { return variance;}        // returns variance
  double getNugget() const { return nugget;}       // returns nugget

  virtual void setRange(double r) { range = r;}    // sets range parameter
  void setNugget(double n)                         // sets nugget, given variance
    { nugget = n; partial_sill = variance - n; }
  void setVariance(double v)                       // sets variance, given nugget
    { variance = v; partial_sill = v - nugget; }
  virtual void setParam(const nVector& param) = 0; // sets all parameters

  // ** computation of covariances:

  virtual double covX(double hx, double hy) const = 0;
  virtual double covY(double hx, double hy) const = 0;
  virtual double covXY(double hx, double  hy) const = 0;
  virtual double covYX(double hx, double  hy) const = 0;
};
```

KEYWORDS

covariance, stochastic field, vector field

DESCRIPTION

The class is an abstract base class for representations of covariance functions for vector fields, in the current version restricted to two-dimensional fields. The covariance model is to have a maximum of three parameters: the range, the variance and the nugget effect.

In the derived classes implemented in the current version, the covariance functions `covX`, `covY`, `covXY` and `covYX` are derived from the covariance function

```
C(r) = c0 * delta(r) + c1 * rho(r;R)    (1)
```

for a scalar field, by differentiation. Here, `c0` is the nugget effect, `c1` is the partial sill and `R` is the range for the scalar field model. The function `rho(r;R)` is the correlation function, and `delta(r)` the delta-function.

For `C(r)` to be differentiable for `r`=0, the nugget effect has to be 0. The only parameters to be specified, is then the variance (equal to `c1` when `c0`=0), and the range `R`.

For a definition of the components of the covariance function for the vector field, see Høst (1994).

REFERENCE:

Høst, G.: "A Note on Two-Dimensional Non-Divergent Gaussian Random Vector Fields", NR-notat, STAT/05/94.

CONSTRUCTORS AND INITIALIZATION

The class has two constructors, a default constructor initializing all parameters to zero, and a constructor taking values for the range, the variance and the nugget effect as arguments. In the second constructor, the variance has default value one and the nugget effect zero.

MEMBER FUNCTIONS

`covX` - returns the covariance value for the first vector field components of two points with distances `hx` and `hy` in the x- and y- directions respectively.

`covY` - returns the covariance value for the second vector field components of two points with distances `hx` and `hy` in the x- and y- directions respectively.

`covXY` - returns the cross-covariance function between the first component in point one and the second component in point two, for two points with distances `hx` and `hy` in the x- and y- directions respectively.

`covYX` - returns the cross-covariance function between the second component in point one and the first component in point two, for two points with distances `hx` and `hy` in the x- and y- directions respectively.

FILES

VecCovFunc.C

EXAMPLE

```
#include <BessVecCovFunc.h>
#include <GaussVecCovFunc.h>
#include <matrix.h>

main(int argc, char* argv[])
{
```

```
  char* type;
  type = argv[1];                  // type of covariance model
                                   // ("bessel" or "gaussian")
  // parameters:
  double variance = 1.0, range = 0.3;

  // coordinates:
  int dim = 2;                     // two-dimensional space
  char* coordfile = "coord.dat";
  int np = countNumbers(coordfile); // counts the number of entries on the file
  np = np/dim;                     // number of locations

  nMatrix Coord(np,2);             // np locations in two dimensions
  Coord.scan("FILE=coord.dat");    // reads coordinates from file 'coord.dat'

  // create an object of a class in the VecCovFunc hierarchy:
  VecCovFunc* cov;
  if (!strncmp(type,"gaussian",8))
    cov = new GaussVecCovFunc(range, variance); // gaussian type covariance
  else if (!strncmp(type,"bessel",6))
    cov = new BessVecCovFunc(range, variance);  // Bessel function covariance

  // compute covariances:
  SymMatrix covXMatr(np);          // covariance matrices
  SymMatrix covYMatr(np);
  SymMatrix covXYMatr(np);
  nVector x1(dim), x2(dim);        // coordinates
  double x1dist, x2dist;

  for (int i=1;i<=np;i++){
    x1 = Coord.row(i);             // coordinates for location i
    for (int j=1;j<=i;j++){
      x2 = Coord.row(j);           // coordinates for location j
      x1dist = x1(1)-x2(1);
      x2dist = x1(2)-x2(2);
      covXMatr(i,j) = cov->covX(x1dist,x2dist);
      covYMatr(i,j) = cov->covY(x1dist,x2dist);
      covXYMatr(i,j) = cov->covXY(x1dist,x2dist);
    }
  }
  // print to standard output:
  covXMatr.print(cout);
  covYMatr.print(cout);
  covXYMatr.print(cout);
}
```

SEEALSO

class GenVecFieldVar

AUTHOR

Turid Follestad, NR

3.2.4 BessVecCovFunc

NAME

BessVecCovFunc - modified Bessel function covariance functions for gaussian vector fields

INCLUDE

```
include "BessVecCovFunc.h"
```

SYNTAX

```
class BessVecCovFunc : public VecCovFunc
{

 protected:
  double ord,                          // order of modified Bessel function
         rfactor,                      // scaling factor
         B, gconst;                    // constants, depending on the order

 public:
  BessVecCovFunc (double r, double v=1.0, double bo=2.0);
  BessVecCovFunc ()
    :VecCovFunc() { ;}

  // set parameters:
  void setParam(const nVector& param);
  void setBesselOrder(double bo) { initOrder(bo,this->range); }
  void setRange(double r) { range = r; setBesselOrder(this->ord); }

  // get order of modified Bessel functions:
  double getBesselOrder () const { return ord; }

  // computation of covariances:
  double covX (double hx, double hy) const;
  double covY (double hx, double hy) const;
  double covXY (double hx, double hy) const;
  double covYX (double hx, double hy) const;
};
```

KEYWORDS

covariance, modified Bessel functions, stochastic field, vector field

DESCRIPTION

The class contains functions for computing the covariance and cross covariance functions for a two dimensional non-divergent isotropic gaussian vector field, using a modified Bessel function covariance model. The components of the covariance function is computed by differentiation of the covariance function for a corresponding scalar field.

Let

$$K_n(z)$$

be the modified Bessel function of order `n`, and let

```
z = S*r/R.
```

Here, `S` is a scaling factor depending on the Bessel order, `r` is the radial distance and `R` is the range.

The correlation function `rho(r)` for the scalar field model, is defined as (Abrahamsen (1994), p.44):

```
rho(r) = gconst * pow(z,n) * K (z).
                              n
```

By differentiation, the components of the modified Bessel function covariance function for the vector field become

```
                          n+1   n-2                                 2
covX(r) = gconst*c1*(S/R)   * r   * (r*K   (z) - (S/R)*K   (z)*r ),
                                       n-1             n-2     y


                          n+1   n-2                                 2
covY(r) = gconst*c1*(S/R)   * r   * (r*K   (z) - (S/R)*K   (z)*r ),
                                       n-1             n-2     x
```

and

```
                                    n+2   n-2
covXY(r) = covYX(r) = gconst*c1*(S/R)   * r   * K   (z)* r * r .
                                                  n-2       x   y
```

Here, `n` and `c1` are the order of the modified Bessel function and the variance for the corresponding scalar field covariance function, and `gconst` is a constant depending on `n`. For `n`=2, the constant `gconst` equals 0.5, and `S` is approximately 5.37.

REFERENCE:

Abrahamsen, P.: "A Review of Gaussian Random Fields and Correlation Functions", NR-Report no.878, 1994.

CONSTRUCTORS AND INITIALIZATION

The class has two constructors, a default constructor, and a constructor taking values for the range and the variance for the corresponding scalar field and the order of the modified Bessel function for the scalar field model, as arguments. The variance has default value one, and in the current version, the order of the modified Bessel functions has to be 2.

MEMBER FUNCTIONS

See also the SYNTAX section.

`setParam` - sets the parameters of the covariance model. The elements of the vector `param` is expected to be ordered as follows: the variance, the range, and the order of the modfied Bessel functions.

`setBesselOrder` - resets the order of the modified Bessel function of the scalar model, given the range parameter.

`setRange` - resets the range parameter, given the order of the modified Bessel function of the scalar model.

`covX` - see the documentation of class `VecCovFunc`.

`covY` - see the documentation of class `VecCovFunc`.

`covXY` - see the documentation of class `VecCovFunc`.

`covYX` - see the documentation of class `VecCovFunc`.

FILES

BessVecCovFunc.C

EXAMPLE

See class VecCovFunc.

SEEALSO

class VecCovFunc, class GenVecFieldVar

AUTHOR

Turid Follestad, NR

3.2.5 GaussVecCovFunc

NAME

GaussVecCovFunc - "gaussian" type covariance functions for two-dimensional gaussian vector fields

INCLUDE

```
include "GaussVecCovFunc.h"
```

SYNTAX

```
class GaussVecCovFunc: public VecCovFunc
{
 public:
  GaussVecCovFunc(double r, double v = 1.0)
    : VecCovFunc(r,v,0.0) { cfactor = 6/(r*r);}
  GaussVecCovFunc(): VecCovFunc()
    { cfactor = 0.0;}

  void setRange(double r) { range = r; cfactor = 6/(r*r);}
  void setParam(const nVector& param);

  double covX(double hx, double hy) const;
  double covY(double hx, double hy) const;
  double covXY(double hx, double hy) const;
  double covYX(double hx, double hy) const;
};
```

KEYWORDS

covariance, gaussian field, stochastic field, vector field

DESCRIPTION

The class contains functions for computing "gaussian" type covariance and cross-covariance functions for a two dimensional non-divergent isotropic gaussian vector field. The components of the covariance function is computed by differentiation of the covariance function for a corresponding scalar field.

The correlation function `rho(r;R)` for the scalar field model, is defined as

$$\mathtt{rho(r;R)} = \exp(-3*(r/R)^2).$$

By differentiation, the components of the "gaussian" type covariance function for the vector field become

$$\mathtt{covX(r)} = c1 * A * (1 - 6 * (r/R)_y^2) * \mathtt{rho(r;R)},$$

$$\mathrm{covY}(r) = c1 * A * (1 - 6 * (r/R)^2) * \mathrm{rho}(r;R),$$
$$x$$

and

$$\mathrm{covXY}(r) = \mathrm{covYX}(r) = c1 * A * 6 * r * r/(R*R) * \mathrm{rho}(r;R).$$
$$x \quad y$$

Here `c1` and `R` are the variance and range of the corresponding scalar field, `r` is the radial distance, and `A` is a constant, depending on `R`, defined by

```
A = 6/(R*R).
```

CONSTRUCTORS AND INITIALIZATION

The class has two constructors, a default constructor, and a constructor taking values for the range and the variance for the corresponding scalar field as arguments. The variance has default value 1.0.

MEMBER FUNCTIONS

`setParam` - sets the two parameter values variance and range. The vector **`param`** is expected to have the variance as its first element.

`setRange` - resets the value of the range.

`covX` - see the documentation of class **`VecCovFunc`**.

`covY` - see the documentation of class **`VecCovFunc`**.

`covXY` - see the documentation of class **`VecCovFunc`**.

`covYX` - see the documentation of class **`VecCovFunc`**.

FILES

VecCovFunc.C

EXAMPLE

See class VecCovFunc.

SEEALSO

class VecCovFunc, class GenVecFieldVar

AUTHOR

Turid Follestad, NR

3.3 Trend models

3.3.1 Trend

NAME

Trend - an abstract base class for models of the trend of a spatial surface

INCLUDE

```
include "Trend.h"
```

SYNTAX

```
class Trend : public virtual HandleId
{
protected:
   nVector coef;                // vector of trend coefficients
   int order;                   // no. of coefficients in the trend function
public:
   Trend (int ord):
     coef(ord), order(ord) { ;} // coefficient values are set to zero
   Trend ():
     coef(), order(0) { ;}
   virtual ~Trend () { ;}

   // creating a copy of an instance of a derived class:
   Trend* createCopy() const;

   // get the number of and the values of the trend coefficients:
   int getOrder() const { return order; }
   nVector getCoef() const { return coef; }

   // set coefficient values:
   void setCoef (const nVector& cf) { coef = cf; }
   void setCoef (double cf) { for (int i=1;i<=order;i++) coef(i) = cf; }

   virtual void funcVec (nVector& f, const nVector& x) const = 0;
   virtual void funcMatr (nMatrix& F, const nMatrix& X) const;
   virtual double value (const nVector& x) const;

   virtual BooLean ok() const;  // returns dpFALSE if order == 0
   virtual char* typeId() const = 0;
};
```

KEYWORDS

trend, trend coefficients, trend function

DESCRIPTION

The class is an abstract base class for representations of the trend of a spatial surface. The trend model `m(x;b)` is supposed to be of the type

```
m(x;b) = f'(x)*b,                    (1)
```

where `f(x) = (f1(x),....,fq(x))` is a vector of general functions, and `b` is the vector of trend coefficients. Typically, `f1(x)==1`.

The class is derived from class HandleId to simplify memory management by introducing *Handles* or *smart* pointers for this class object.

CONSTRUCTORS AND INITIALIZATION

The class has two constructors including a default constructor, initializing the members to zero.

The integer argument indicates the order, or number of coefficients, of the trend model.

MEMBER FUNCTIONS

See also the SYNTAX section.

`redim` - resets the number of coefficients to the value of the integer argument.

`createCopy` - creates a copy of an object of a derived class.

`funcVec` - computes the vector `f(x)` as defined in (1). The coordinates are specified by the `nVector` argument `x`, and the resulting vector is returned by the `nVector` reference argument `f`.

`funcMatr` - computes the matrix `F` for a set of `n` points, where the `i`th row equals the vector `f(xi)` in (1) for the coordinate vector `xi`. The function argument is a matrix of coordinates `X` (nobs by dim), and the result is returned by the `nMatrix` reference argument `F` (nobs by order).

`value` - returns the trend value `m(x;b)` in (1), in the point `x`, by computing the inner product of the member `coef` and `f(x)`, computed by `funcVec`.

FILES

Trend.C

EXAMPLE

```
#include <PolTrend.h>
#include <PointSet.h>

main()
{
  int dim = 2, np = 50;      // spatial dimension, number of points
  double val = 3.0;          // value of trend coefficients
  int pord = 1;              // order of polynomial trend

  // initialize a pointer to a PolTrend object as a class Trend pointer:
  Trend* tr = new PolTrend(pord,dim);

  tr->setCoef(val);          // sets trend coefficients, all equal

  // set of points:
  GenPointSet X(np,dim);
```

```
  ifstream in("point.set",ios::in);
  X.scan(in);               // reads point set from a file

  // compute vector of trend values:
  nVector trendval(np);
  for (int i=1;i<=np;i++)
    trendval(i) = tr->value(X.coord(i)); // trend value in point xi

  trendval.print("FILE=trendval.dat");   // prints trend values to a file
}
```

SEEALSO

class GenTrend, class PolTrend

AUTHOR

Jon Helgeland, NR, modified by Turid Follestad, NR.

3.3.2 PolTrend

NAME

PolTrend - a class for a polynomial model of the trend of a spatial surface

INCLUDE

```
include "PolTrend.h"
```

SYNTAX

```
class PolTrend : public Trend
{
private:
   int polOrder;                              // order of polynomial trend
   int funcLen (int d) { return binCoef(d+polOrder,d);};
                                              // length of polynomial trend vector
public:
   PolTrend (int po, int dim)
    :Trend(binCoef(dim+po,dim))
      { polOrder=po;}
   PolTrend (const PolTrend& pt)
    :Trend(pt.order)
      { polOrder = pt.polOrder; coef = pt.coef;}
   PolTrend ()
    :Trend()
      { polOrder = 0;}

   // redimension the polynomial order (po) and spatial dimension (dim):
   BooLean redim(int po, int dim);

   // get polynomial order:
   int getPolOrder() const { return polOrder;}

   void funcVec (nVector& f, const nVector& x) const;
   char* typeId() const { return "PolTrend";}
};
```

KEYWORDS

polynomial trend, trend, trend function

DESCRIPTION

The class is a representation of a polynomial trend surface of general order.

CONSTRUCTORS AND INITIALIZATION

The class has a copy constructor and a constructor taking two arguments: the order of the polynomial, and the spatial dimension.

MEMBER FUNCTIONS

See the SYNTAX section and the documentation of the base class `Trend`.

FILES

PolTrend.C

EXAMPLE

See class Trend.

SEEALSO

class Trend

AUTHOR

Jon Helgeland, NR

3.3.3 GenTrend

NAME

GenTrend - a class for a general trend surface model

INCLUDE

```
include "GenTrend.h"
```

SYNTAX

```
class GenTrend : public Trend
{
 private:
  VecSimplest(TFP)* fp;          // vector of pointers to TrendFunction

 public:
  GenTrend(VecSimplest(TFP)* p)
    :Trend(p->size())
      {fp=p; }
  GenTrend(const GenTrend& gt)
    :Trend(gt.order)
      {fp=gt.fp; }
  GenTrend()
    :Trend() { fp = NULL;}
  ~GenTrend () { }

  void funcVec(nVector& f, const nVector& x) const;

  BooLean ok () const;           // returns false if order == 0 or fp == NULL
  char* typeId() const { return "GenTrend";}
};
```

KEYWORDS

general trend, trend, trend function

DESCRIPTION

The class represents a general model for the trend of a spatial surface. The vector `f(x)`, as defined in the documentation of the base class `Trend`, is represented by an array of pointers to class `TrendFunction` objects. Each `TrendFunction` object defines one element of the vector `f(x)`.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors, including a copy constructor and a default constructor. The input to the third constructor is an array of objects of type pointer to `TrendFunction`, transferred through an object of type `VecSimplest(TFP)`, where TFP stands for pointer to class `TrendFunction`. Note that the address of the actual argument is taken, and no new memory is allocated.

MEMBER FUNCTIONS

See the SYNTAX section and the documentation of the base class `Trend`.

FILES

GenTrend.C

EXAMPLE

```
#include <Trend.h>
#include <GenTrend.h>
#include <PointSet.h>

// trend function f1(x) = sqrt(x'x):

class TrendFunc1 : public TrendFunction
{
public:
  double evaluate(const nVector& x) const
    { return x.normE();}
};

// trend function f2(x) = x(1)*x(1):

class TrendFunc2 : public TrendFunction
{
public:
  double evaluate(const nVector& x) const
    { return x(1)*x(1);}
};


main()
{
  int ord = 2;
  int np = 50, dim = 2;
  VecSimplest(TFP) tfunc(ord);

  // initializes VecSimplest(TFP) using the trend functions implemented
  // in TrendFunc1 and TrendFunc2:

  tfunc(1) = new TrendFunc1;
  tfunc(2) = new TrendFunc2;

  GenTrend gtr(&tfunc);                  // GenTrend object

  // trend coefficients:
  nVector trcoef(ord);
  trcoef(1) = 2.0;
  trcoef(2) = 3.0;
  gtr.setCoef(trcoef);                   // sets trend coefficients

  // point set:
  GenPointSet X(np,dim);
  X.scan("FILE=point.set");

  // vector of trend values, f'(x)*b:
  nVector trendval(np);
  for (int i=1;i<=np;i++)
    trendval(i) = gtr.value(X.coord(i));  // trend value in the point xi

  // regressor matrix:
```

```
  nMatrix F(np,ord);
  nMatrix Xmatr = X.coordMatrix();        // coordinate matrix
  gtr.funcMatr(F,Xmatr);                  // computes regressor matrix F
  F.print("FILE=F.dat");
}
```

SEEALSO

class Trend, class TrendFunction

AUTHOR

Jon Helgeland, NR, modified by Turid Follestad, NR.

3.3.4 TrendFunction

NAME

TrendFunction - an abstract base class for general trend functions

INCLUDE

```
include "TrendFunction.h"
```

SYNTAX

```
class TrendFunction
{
 public:
  virtual double evaluate(const nVector&) const = 0;
};
```

KEYWORDS

general trend, trend, trend function

DESCRIPTION

The class is an abstract base class for classes representing general trend functions. An array of `TrendFunction` objects is used to represent the trend in the class `GenTrend` derived from class `Trend`.

CONSTRUCTORS AND INITIALIZATION

No constructors.

MEMBER FUNCTIONS

`evaluate` - virtual function that returns the value of the trend function (with coefficients scaled to one) in a point with coordinates represented by the `nVector` argument. The function is called by the member function `funcVec` in class `GenTrend`.

FILES

Trend.C

EXAMPLE

```
/************************************************************************/
/* Definition of a derived class of TrendFunction, specifying the trend   */
/* function model                                                         */
/*                                                                        */
/*    f(x) = norm(x).                                                     */
/*                                                                        */
/************************************************************************/
#include <TrendFunction.h>
#include <math.h>

class TrendFunc1 : public TrendFunction
{
 public:
  double evaluate(const nVector& x) const
    { return x.normE();}
};
```

SEEALSO

class GenTrend, class Trend

AUTHOR

Jon Helgeland, NR

3.3.5 TrendMat

NAME

TrendMat - a class for a trend model represented by an arbitrary regressor matrix

INCLUDE

```
include "TrendMat.h"
```

SYNTAX

```
class TrendMat : public Trend
{
  nMatrix F;                    // regressor matrix

 public:
  TrendMat (nMatrix X)
    :Trend(X.getCdim()), F(X)
      { }
  TrendMat (const TrendMat& tm)
    :Trend(tm.order), F(tm.F)
      { coef = tm.coef; }
  TrendMat ()
    :Trend(), F()
      { }

  // virtual functions inherited from class Trend that are not valid for
  // class TrendMat, and issue warnings if they are invoked:
  void funcVec (nVector& f, const nVector& x) const; // empty function
  double value (const nVector& x) const;            // returns 0.0

  // returns regressor matrix:
  // the first function is inherited from the base class Trend
  void funcMatr (nMatrix& F, const nMatrix& X) const { F = this->F; }
  void funcMatr (nMatrix& F) const { F = this->F; }

  char* typeId() const { return "TrendMat";}
};
```

KEYWORDS

regressor matrix, trend

DESCRIPTION

The class, derived from class `Trend`, is a representation of a trend model by the regressor matrix `F` of a spatial model. For a Gaussian random prossess, let

```
E(Z(x)) = f'(x)*beta.
```

Let `xi` be one of `n` data locations. Then, `f(xi) = (f1(xi),...,fq(xi))`, is the `i`th row of the `n` by `q` regressor matrix F. Here, `q` is the number of regressors. The vector `beta` is the vector of trend coefficients.

CONSTRUCTORS AND INITIALIZATION

To create an object of the class, the regressor matrix `F` must be supplied to the constructor. The other to constructors are a copy and a default constructor.

MEMBER FUNCTIONS

See also the SYNTAX section.

`funcMatr` - virtual function, inherited from base class **`Trend`**, that returns the regressor matrix. The **`nMatrix`** argument **`X`** in the argument list of the base class virtual function is redundant, and the function is overloaded. But the function with two arguments is needed to make it possible for the function to be invoked for a class **`Trend`** pointer.

FILES

None.

EXAMPLE

```
#include <TrendMat.h>

main()
{
  int ntr = 2, np = 20;     // 20 data locations and 2 regressors.

  nMatrix X(np,ntr);        // regressor matrix
  X.scan("FILE=Xmatr.dat"); // reads regressor matrix from file

  TrendMat tr(X);           // creates TrendMat object

  tr.funcMatr(X);           // gets regressor matrix
  X.print(cout);            // prints regressor matrix to standard output
}
```

SEEALSO

class Trend

AUTHOR

Turid Follestad, NR

3.3.6 VecSimplest(Trend)

NAME

VecSimplest(Trend) - a very simple vector of pointers to Trend objects

INCLUDE

```
include "VecSimplest_Trend.h"
```

SYNTAX

```
typedef Trend* TP;

#define Type TP
#include <VecSimplest.h>
#undef Type
```

KEYWORDS

trend, vector

DESCRIPTION

`VecSimplest(TP)` is a class implementing a very simple vector of pointers to `Trend` objects, and is a specification of the parametric class `VecSimplest(Type)` in Diffpack, with parameter `Type` equal `Trend*`. The index base is 1. The only operation available is subscripting.

See documentation of the parametric class `VecSimplest(Type)` implemented in Diffpack for a description of the interface of the class.

SEEALSO

class Trend, class VecSimplest(Type)

3.3.7 VecSimplest(TrendFunction)

NAME

VecSimplest(TFP) - a very simple vector of pointers to TrendFunction objects

INCLUDE

```
include "VecSimplest_TrendFunction.h"
```

SYNTAX

```
typedef TrendFunction* TFP;

#define Type TFP
#include <VecSimplest.h>
#undef Type
```

KEYWORDS

trend, trend function, vector

DESCRIPTION

`VecSimplest(TFP)` is a class implementing a very simple vector of pointers to `TrendFunction` objects, and is a specification of the parametric class `VecSimplest(Type)` in Diffpack, with parameter `Type` equal `TrendFunction*`. The index base is 1. The only operation available is subscripting.

See documentation of the parametric class `VecSimplest(Type)` implemented in Diffpack for a description of the interface of the class.

SEEALSO

class TrendFunction, class VecSimplest(Type)

Chapter 4

Estimation of parameters of spatial models

4.1 Abstract base class

4.1.1 SpatialModel

NAME

SpatialModel - an abstract base class for estimation of the parameters of a spatial model

INCLUDE

```
include "SpatialModel.h"
```

SYNTAX

```
class SpatialModel
{
 protected:
  Handle(SpatialData) sdata;              // data
 public:
  SpatialModel (const SpatialData& d)
    { sdata.rebind(d.createCopy()); }
  SpatialModel () { ;}
  virtual ~SpatialModel () {;}            // Handle deletes data object

  virtual void estimate() = 0;            // estimates the parameters
  virtual BooLean modelFitted() const = 0; // returns dpTRUE if model is fitted
  virtual void getModelEst(SpatialModelDescr&) = 0; // gets estimated model
};
```

KEYWORDS

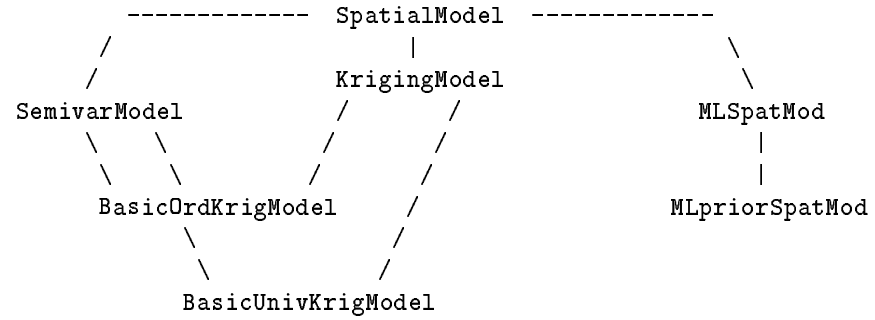
spatial model, parameter estimation

DESCRIPTION

This class is an abstract base class for classes implementing estimation of parameters for the trend and structure of variation in spatial models.

In the class hierarchy, two types of methods are implemented. These are weighted and generalized least squares estimation, and maximum likelihood estimation.

The class hierarchy is organized as follows:



The class `SemivarModel` estimates the semivariogram parameters for a constant trend model or a residual model, using a non-parametric semivariogram estimate and weighted least squares. The other classes estimate the trend parameters as well. The classes derived from the abstract class `KrigingModel` use generalized least squares, while the model parameters are estimated by maximum likelihood by the classes `MLSpatMod` and `MLpriorSpatMod`. In class `MLpriorSpatMod`, prior information on the parameters are included.

The models and methods are further specified in the documentation of each class.

The member `SpatialData` is referenced through the use of smart pointers, i.e. handles, for simplified memory management. A handle consists of a pointer to the object and some interface functions.

CONSTRUCTORS AND INITIALIZATION

The class has one constructor taking an argument of type `SpatialData`. This object is supposed to contain the coordinates and response values of the spatial data set to estimate the parameters from. The object is copied.

MEMBER FUNCTIONS

`estimate` - virtual function that is to estimate the model parameters. The resulting parameters are stored in the derived classes, and can be returned by calling the function `getModelEst`.

`modelFitted` - virtual function that is to return dpTRUE if the model is fitted and dpFALSE if it is not.

`getModelEst` - returns the estimated model parameters through the `SpatialModelDescr` reference argument.

FILES

None.

EXAMPLE

```
void f(SpatialModel& spmod)
{
  // ...
  spmod.estimate();                // estimates the parameters of a model
  SpatialModelDescr model;         // creates an object to hold the parameters
  spmod.getModelEst(model);        // gets the estimated model parameters

  // print the parameters to a file:
  ofstream outfile("model",ios::out);
  model.trend->getCoef().print(outfile);
  outfile << endl;
  model.vmodel->getParam().print(outfile);
  outfile << endl;
  // ...
}
```

SEEALSO

class SpatialModelDescr

AUTHOR

Turid Follestad and Jon Helgeland, NR

4.1.2 SpatialModelDescr

NAME

SpatialModelDescr - a class holding the parameters of a spatial model

INCLUDE

```
include "SpatialModelDescr.h"
```

SYNTAX

```
class SpatialModelDescr
{
 public:
  Handle(Trend) trend;            // trend model
  Handle(VarStructure) vmodel;    // model for structure of variation

  SpatialModelDescr() {;}
  SpatialModelDescr(const prm(Trend)& prm, char* vmod)
    { init(prm,vmod);}
  ~SpatialModelDescr() {;}        // Handles take care of deallocation

  void init(const prm(Trend)& prm, char* vmod);
  void print(Os os);
};
```

KEYWORDS

semivariogram, spatial model, structure of variation, trend

DESCRIPTION

The class holds the parameters of a spatial model. It has two data members: a handle (smart pointer) to class `Trend` and a handle (smart pointer) to class `VarStructure`. The class is used by the virtual member function `getModelEst` of the `SpatialModel` hierarchy, to get the values of the estimatied parameters. The user only needs to create an instance of the class by the default constructor, and enter this object as argument to the `getModelEst` function. Since the data members are public, the user will have direct access to the estimated parametric models.

The members are referenced through the use of smart pointers, i.e. handles, for simplified memory management. A handle consists of a pointer to the object and some interface functions.

CONSTRUCTORS AND INITIALIZATION

The class has two constructors, a default constructor and a constructor taking two arguments: a `prm(Trend)` object holding the parameters needed to create an instance of a class in the `Trend` hierarchy, and a character string indicating the model for the structure of variation.

MEMBER FUNCTIONS

`init` - initializes the data members. The first argument specifies the parameters needed to create an object of a class in the `Trend` hierarchy. The second, of type char*, specifies the type of parametric model for the structure of variation.

FILES

SpatialModelDescr.C

EXAMPLE

See class SpatialModel

SEEALSO

class SpatialModel, class Trend, class VarStructure

AUTHOR

Turid Follestad, NR

4.2 Estimation by weighted and generalized least squares

4.2.1 SemivarModel

NAME

SemivarModel - a class for estimation of semivariogram parameters

INCLUDE

```
include "SemivarModel.h"
```

SYNTAX

```
class SemivarModel: public virtual SpatialModel
{
 protected:
  Handle(NonParSemivar) semivar;  // non-parametric semivariogram
  Handle(VarStructure) vmodel;    // model for structure of variation

  BooLean estimated;              // indicator of whether the estimation is OK

  double wlsEst (const nVector& prm);   // weighted least squares function
  void estPar();                  // estimates the semivariogram parameters

 public:
  SemivarModel(const SpatialData&, const VarStructure&, const NonParSemivar&,
               int minl=30);
  SemivarModel(const SpatialData&, const VarStructure&, double lsz=0,
               int minl=30);
  virtual ~SemivarModel() { if (minim!=NULL) delete minim;}

  void initPar(const nVector& par);     // initializes semivariogram parameters

  void estimate();                      // estimates the parameters

  // reset non-parametric semivariogram:
  void reset (const SpatialData& sd)               // resets data
    { semivar->resetData(sd); estimated = dpFALSE;}
  void reset (int lsz)                             // resets lagsize
    { semivar->setLagsize(lsz); estimated = dpFALSE; }

  BooLean modelFitted() const { return estimated;}

  nVector getParam() { return vmodel->getParam();} // gets estimated parameters
  void getModelEst(SpatialModelDescr&);            // gets estimated model
};
```

KEYWORDS

covariance, parameter estimation, semivariogram, variogram

DESCRIPTION

The class implements estimation of the parameters of the variogram for a set of residuals. The estimation is carried out using the approximated weighted least squares method

proposed by Cressie (1985).

The estimated parameters are those that minimize the function

```
                                                          2
f(h) =  sum      [ N(hj) *{ (npsv(hj)/sv(hj;theta) - 1} ]
      all lags hj
```

where `hj` is lag number j, `N(hj)` is the number of pairs of points in lag `j`, and `npsv(hj)` and `sv(hj;theta)` are the non-parametric an parametric semivariograms for lag `hj` and parameter vector `theta`.

Values of the non-parametric semivariogram corresponding to a number of pairs of points less than a predefined minumum number, will be ignored in the weighted least squares fitting. By default, this number is 30. (Conf. Journel and Huijbregts (1978), p.194.) Another value of this minimum number can be specified in the constructor.

Some protected members are referenced through the use of smart pointers, i.e. handles, for simplified memory management. A handle consists of a pointer to the object and some interface functions.

REFERENCES:

Journel, A.G., and Huijbregts, Ch.J.: "Mining Geostatistics", Academic Press, London, 1978.

Cressie, N.: "Fitting Variogram Models by Weighted Least Squares", Math.Geol.**17**, 1985, pp.563-586.

CONSTRUCTORS AND INITIALIZATION

The class has two constructors. Both take a `SpatialData` and a `VarStructure` argument, initializing the spatial data set and the model of the semivariogram or covariogram structure. The initial values for the estimation of the semivariogram parameters must be specified within the `VarStructure` argument to the constructor.

The parameters are estimated from a non-parametric semivariogram estimate. This can be estimated in advance, and the corresponding `NonParSemivar` object can then be specified as a constructor argument. Alternatively, using the other constructor, the lagsize of the semivariogram to be estimated could be specified. When using the default lagsize (0), the lagsize is computed by the program, as described in the documentation of class `NonParSemivar`. The integer argument `minl`, with default value 30, specifies the minimum number of pairs of points to be allowed in each lag interval for the weighted least squares fitting.

MEMBER FUNCTIONS

See also the SYNTAX section. Only public member functions are documented below.

`initPar` - re-initializes the semivariogram parameters to the parameters of the `nVector` argument. The parameters should be (in that order), the nugget effect, the partial sill, and the range.

`estimate` - estimates the semivariogram parameters by first estimating a non-parametric semivariogram, and then fitting a parmetric model by weighted least squares. The initial values for the parameters are assumed to be the parameters of the `VarStructure` object of the constructor, if no other values are specified by a call to the `initPar` member function.

If the `SemivarModel` object is initialized using a `NonParSemivar` object for which the semivariogram is already estimated, no new estimation of the non-parametric semivariogram is performed. The estimation status of the `NonParSemivar` object is checked by its member function `ok`.

`getModelEst` - virtual function inherited from `SpatialModel` that gets the estimated model parameters. Since the only parameters estimated in this class is the vector of semivariogram parameters, it is easier to use the member `getParam` to get the estimated parameter vector. But `getModelEst` is needed to get access to the estimated parameters when the member `estimation` is invoked from a pointer to the base class `SpatialModel`.

`modelEstimated` - returns dpTRUE if the semivariogram parameters are estimated.

`reset(const SpatialData&)` - resets the data used to estimate the non-parametric semivariogram.

`reset(int lsz)` - resets the lagsize of the non-parametric semivariogram to `lsz`. The number of lags will also be changed by this member function.

FILES

SemivarModel.C

EXAMPLE

```
#include <SemivarModel.h>
#include <NonParSemivar.h>

main(int argc, char* argv[])
{
  int np = atoi(argv[1]);              // number of observations
  int d = atoi(argv[2]);               // spatial dimension

  // data:

  IrregSpatialData data(np,d);         // irregularly spaced data
  data.scan("FILE=dataset.dat");       // reads data from a file

  // parametric semivariogram model, modified Bessel function type:

  VarStructure vmod("bessel");         // structure of variation object
  nVector param(3);
  param.scan("FILE=initpar.dat");      // reads initial values for parameters
  vmod.setParam(param);                // sets the parameters of the object

  // estimation of a non-parametric semivariogram:

  NonParSemivar npsv(data);            // non-parametric semivariogram object
  npsv.estimate();                     // estimates non-parmetric semivariogram
  npsv.print("FILE=npsemivar.dat");    // prints non-parmetric semivariogram

  // parameter estimation:

  SemivarModel modest(data,vmod,npsv); // object for parameter estimation
                                       // -number of lags is computed from data
  modest.estimate();                   // estimates the parameters

  // output of estimated parameters:

  param = modest.getParam();           // gets parameter vector
  param.print("FILE=param.dat");       // prints estimated parameters to a file
}
```

SEEALSO

class SpatialModel, class NonParSemivar, class VarStructure

AUTHOR

Turid Follestad and Jon Helgeland, NR

4.2.2 KrigingModel

NAME

KrigingModel - an abstract base class for estimation of the parameters used in optimal linear spatial prediction, or kriging

INCLUDE

```
include "KrigingModel.h"
```

SYNTAX

```
class KrigingModel: public virtual SpatialModel
{
 public:
  KrigingModel(const SpatialData& sd)
    :SpatialModel(sd)
      { knownVarStructure = dpFALSE; glsq = NULL;}
  virtual ~KrigingModel()
    { if (glsq!=NULL) delete glsq;}

  virtual void estimateTrend() = 0;
  void knownVar(BooLean known) { knownVarStructure = known;}
};
```

KEYWORDS

estimation, kriging, parameters, spatial model

DESCRIPTION

The class is an abstract base class for classes implementing estimation of the parameters in the kriging model. The class is derived from the base class `SpatialModel`.

The classes in the `KrigingModel` hierarchy contains routines for estimating the trend and semivariogram parameters of a basic stochastic model for a spatial surface, z(x). The surface z(x) is viewed as a realization of the stochastic process `Z(x)`, with model

```
Z(x) = m(x,b) + e(x).
```

Here, `b` is a vector of trend coefficients, `m(x,b)` is a deterministic trend function, and `e(x)` are spatially dependent gaussian residuals. It is assumed that the residual stochastic process is second order stationary and isotropic, which means that the mean and covariance of `e(x)` are given by:

```
E(e(x)) = 0,
                                          2
Cov(e(x1),e(x2)) = C(|x1-x2|) = sigma *rho(|x1-x2|),
```

where `C(h)` and `rho(h)` are isotropic covariance and correlation functions.

Two types of models are implemented as subclasses of `KrigingModel`, an ordinary and an universal kriging model.

REFERENCES:

Cressie, N.: "Statistics for Spatial Data", Wiley, New York, 1991.

Journel, A.G. and Huijbregts, Ch.J.: "Mining Geostatistics", Academic Press, London, 1978.

Ripley, B.D.: "Spatial Statistics", Wiley, New York, 1981.

CONSTRUCTORS AND INITIALIZATION

The class has one constructor, initializing the `SpatialData` object representing the observed spatial data.

MEMBER FUNCTIONS

`knownVar` - specifies whether the parameters of the semivariogram, or the corresponding covariance structure, are known. By default, it is supposed that the parameters are unknown. If `known` = dpTRUE, the known parameters must be specified when an object of one of the derived classes is initialized.

FILES

None.

EXAMPLE

```
#include <BasicOrdKrigModel.h>
#include <BasicUnivKrigModel.h>
#include <PolTrend.h>

main(int argc, char* argv[])
{
  int np = atoi(argv[1]);                 // number of points
  int d = atoi(argv[2]);                  // spatial dimension
  int ptrorder = atoi(argv[3]);           // order of polynomial trend

  // read data:
  IrregSpatialData data(np,d);            // irregularly spaced data
  data.scan("FILE=dataset.dat");          // reads data from a file

  // specification of variogram model:
  char* vmtype = "exponential";           // type of parametric model
  nVector initPar(3);                     // initial parameter vector
  initPar.scan("FILE=vparam.dat");        // reads initial values from a file
  VarStructure vmodel(vmtype);            // object for structure of variation
  vmodel.setParam(initPar);               // sets parameters
  vmodel.setUpper(1,0.0);                 // upper bound of parameter 1 is 0,
                                          // equal to lower bound

  // specification of trend model:
  PolTrend ptr(ptrorder,d);               // polynomial trend object
  int nparam = ptr.getOrder();            // number of trend coefficients
```

```
  // spatial model estimation object:
  KrigingModel* modestim;
  if (nparam>1){ // universal kriging model
    modestim = new BasicUnivKrigModel(data,ptr,vmodel);
  }
  else{ // ordinary kriging model
    modestim = new BasicOrdKrigModel(data,vmodel);
  }

  // parameter estimation:
  modestim->estimate();                    // estimates parameters
  SpatialModelDescr model;
  modestim->getModelEst(model);            // gets estimated parameter values
  model.print("FILE=estparam.dat");         // prints estimated parameters

  delete modestim;                         // deletes estimation object
}
```

SEEALSO

class Kriging, class SpatialData, class SpatialModel

AUTHOR

Turid Follestad and Jon Helgeland, NR

4.2.3 BasicOrdKrigModel

NAME

BasicOrdKrigModel - a class for estimation of the parameters of the ordinary kriging model

INCLUDE

```
include "BasicOrdKrigModel.h"
```

SYNTAX

```
class BasicOrdKrigModel: public KrigingModel, public SemivarModel
{
 protected:
  double trend;          // trend coefficient
  BooLean fitted;        // indicator of whether the parameters are estimated

  void setupGLsq();      // QR-decomposition of the regressor matrix
  void estimateTrend();  // estimates trend for a given variogram model

 public:
  BasicOrdKrigModel(const SpatialData& sd, const VarStructure& vmodel,
                    const NonParSemivar& nps, int minl=30)
    :KrigingModel(sd), SemivarModel(sd,vmodel,nps,minl),
    SpatialModel(sd)
      { fitted = dpFALSE;}

  BasicOrdKrigModel(const SpatialData& sd , const VarStructure& vmodel,
                    double lsz=0, int minl=30)
    :KrigingModel(sd), SemivarModel(sd,vmodel,lsz,minl),
    SpatialModel(sd)
      { fitted = dpFALSE;}

  void estimate();              // estimation of model parameters
  double trendCoefVar();        // variance of estimated trend coefficient

  // indicator of whether model is fitted:
  BooLean modelFitted() const { return fitted;}

  // output of estimated values:

  void getModelEst(SpatialModelDescr&);   // gets model parameters
  void printModelParam(Os out) const;     // prints to an output stream
};
```

KEYWORDS

kriging, ordinary kriging, parameter estimation, spatial model

DESCRIPTION

The class is derived from the abstract base class `KrigingModel`, and contains a routine for estimation of the trend and semivariogram parameters of the ordinary kriging model. The model is the special case of the model defined in the documentation of class `KrigingModel`, with trend function `m(x,b) = b0`, where `b0` is a constant. Thus, the model is defined as

```
Z(x) = b0 + e(x).
```

where the residuals `e(x)` are gaussian and spatially dependent, with first and second order moments as defined in the documentation of class `KrigingModel`. In this case, the semivariogram of the stochastic process `Z(x)` and the residual stochastic process `e(x)` are identical.

The semivariogram parameters are estimated on the basis of a non-parametric semivariogram, by the method of the base class `SemivarModel`, and the trend parameters by generalized least squares.

Initial values of the semivariogram parameters are specified by the `VarStructure` constructor argument.

By calling the member function `knownVar`, inherited from `KrigingModel`, the semivariogram parameter estimation can be swithced off, so that only the trend parameters are estimated. The trend parameter estimation, performed by calling the member function `estimate`, will then be based on the initial values of the semivariogram parameters, as specified by the constructor.

REFERENCES:

Cressie, N.: "Statistics for Spatial Data", Wiley, New York, 1991.

Journel, A.G. and Huijbregts, Ch.J.: "Mining Geostatistics", Academic Press, London, 1978.

Ripley, B.D.: "Spatial Statistics", Wiley, New York, 1981.

CONSTRUCTORS AND INITIALIZATION

The class has two constructors, both taking arguments specifying the observed data set, of type `SpatialData`, and the structure of variation, a `VarStructure` object. An initial guess of the semivariogram parameters should be specified as a part of this object. The trend of an ordinary kriging model is a constant, and no initialization is needed.

The parametric semivariogram is estimated on the basis of a non-parametric semivariogram. This can be estimated on beforehand, and specified by an `NonParSemivar` object in the constructor. By specifying the lagsize only, the non-parametric semivariogram is estimated by the parameter estimation routine. By using the default lagsize (0), it is indicated that the lagsize should be computed by the program. This is done as described in the documentation of class `NonParSemivar`. The integer argument `minl`, with default value 30, specifies the minimum number of pairs of points to be allowed in each lag interval for the weighted least squares fitting of the semivariogram.

MEMBER FUNCTIONS

`estimate` - estimates the model parameters for the trend and the semivariogram. The results are stored in the class, and are returned when calling the function `getModelEst`. First, the semivariogram parameters are estimated by the `estimate` member function of the base class `SemivarModel`. Second, the trend parameters are computed by generalized least squares. If the semivariogram estimation is switched off (by `knownVar`), the trend parameters are estimated using the initial semivariogram parameter values. If the semivariogram parameters are estimated separately, by calling the base class member function `SemivarModel::estimate`, the estimated values will be used.

`estimateTrend` - estimates the trend parameter for a given semivariogram model.

`trendCoefVariance` - retuns the variance of the estimated trend coefficient.

`modelFitted` - returns dpTRUE if the model is fitted and dpFALSE if it is not.

`getModelEst` - returns the estimated model parameters through the `SpatialModelDescr` reference argument.

`printModelParam` - prints the estimated model parameters to an output stream.

FILES

BasicOrdKrigModel.C

EXAMPLE

See class KrigingModel

SEEALSO

class GenLeastSquaresQR, class KrigingModel, class NonParSemivar, class SemivarModel, class SpatialModelDescr, class Trend, class VarStructure

AUTHOR

Turid Follestad abd Jon Helgeland, NR

4.2.4 BasicUnivKrigModel

NAME

BasicUnivKrigModel - a class for estimation of the parameters of the universal kriging model

INCLUDE

```
include "BasicUnivKrigModel.h"
```

SYNTAX

```
class BasicUnivKrigModel: public KrigingModel, public SemivarModel
{
 protected:
  nMatrix F;             // design matrix for trend
  BooLean fitted;        // indicator of whether the parameters are estimated
  Handle(Trend) trend;   // pointer to trend model

  void setupGLsq();      // QR-decomposition of the regressor matrix
  void residuals(nMatrix&);   // computes residuals for variogram estimation
  void estimateTrend();  // estimates the trend for a given variogram model

 public:
  BasicUnivKrigModel(const SpatialData& sd, const Trend&,
                     const VarStructure& vmodel, double lsz=0,
                     int minl=30);

  void estimate();                       // estimation of parameters
  void trendCoefVar(SymMatrix& tv);      // covariance of trend coefficients

  // indicator of whether model is fitted:
  BooLean modelFitted() const { return fitted;}

  // output of estimated values:

  void getModelEst(SpatialModelDescr&);  // gets model parameters
  void printModelParam(Os out) const;    // prints to an output stream
};
```

KEYWORDS

kriging, parameter estimation, spatial model, universal kriging

DESCRIPTION

The class is derived from the abstract base class **KrigingModel**, and contains a routine for estimation of the trend and semivariogram parameters of the universal kriging model. The trend function of the model, as described in the documentation of the class **KrigingModel**, is assumed to be of the form

```
m(x,b) = f'(x)b
```

where `f(x) = (f1(x),....,fq(x))` is a vector of arbitrary functions of the data location `x`, and `b` is the vector of trend coefficients. Typically, `f1(x) == 1`. Thus, the model can be written as

```
Z(x) = f'(x)b + e(x).
```

where the residuals `e(x)` are gaussian and spatially dependent, with first and second order moments as described in the documentation of class `KrigingModel`.

The semivariogram parameters and trend coefficients are estimated simultaneously by iterating over two steps until convergence. First, the trend coefficients are estimated by generalized least squares. At the first iteration, the initial values of the semivariogram parameters are used. Second, the residuals `e(x)` is computed using the estimated trend coefficients, and the semivariogram parameters are estimated by the method of class `SemivarModel`.

The iteration is stopped when the relative difference between the trend coefficients of two succesive iterations is less than 1e-4, or the number of iterations exceeds 10.

Initial values of the semivariogram parameters are specified by the `VarStructure` constructor argument.

By calling the member function `knownVar`, inherited from `KrigingModel`, the semivariogram parameter estimation can be swithced off, so that only the trend parameters are estimated. The estimation, performed by calling the member function `estimate`, will then be based on the initial values of the semivariogram parameters, as specified by the constructor.

The member `Trend` is referenced through the use of smart pointers, i.e. handles, for simplified memory management. A handle consists of a pointer to the object and some interface functions.

REFERENCES:

Cressie, N.: "Statistics for Spatial Data", Wiley, New York, 1991.

Journel, A.G. and Huijbregts, Ch.J.: "Mining Geostatistics", Academic Press, London, 1978.

Ripley, B.D.: "Spatial Statistics", Wiley, New York, 1981.

CONSTRUCTORS AND INITIALIZATION

The class has one constructor. The observed data are specified by the `SpatialData` reference argument, and the parametric models for the structure of variation and the trend function by the `VarStructure` and `Trend` reference arguments. An initial guess of the semivariogram parameters should be specified as a part of the `VarStructure` object.

The parametric semivariogram is estimated on the basis of a non-parametric semivariogram. The lagsize can be specified by the constructor. By default (`lsz`=0) the lagsize is computed by the program, as described in the documentation of class `NonParSemivar`. The integer argument `minl`, with default value 30, specifies the minimum number of pairs of points to be allowed in each lag interval for the weighted least squares fitting of the semivariogram.

MEMBER FUNCTIONS

`estimate` - estimates the model parameters for the trend and the semivariogram by the method described above. If the semivariogram estimation is switched off (by `knownVar`),

the trend parameters are estimated using the initial semivariogram parameter values. The results are stored in the class, and are updated after each iteration. They are returned by calling the function `getModelEst`.

`estimateTrend` - estimates the trend parameters for a given semivariogram model.

`trendCoefVariance` - retuns the covariance matrix of the estimated trend coefficients.

`modelFitted` - returns dpTRUE if the model is fitted and dpFALSE if not.

`getModelEst` - returns the estimated model parameters through the `SpatialModelDescr` reference argument.

`printModelParam` - prints the estimated model parameters to an output stream.

FILES

BasicUnivKrigModel.C

EXAMPLE

See class KrigingModel

SEEALSO

class GenLeastSquaresQR, class KrigingModel, class NonParSemivar, class SemivarModel, class SpatialModelDescr, class Trend, class VarStructure

AUTHOR

Turid Follestad and Jon Helgeland, NR

4.3 Maximum liklihood estimation

4.3.1 MLSpatMod

NAME

MLSpatMod - a class for maximum likelihood estimation for spatial model parameters

INCLUDE

```
include "MLSpatMod.h"
```

SYNTAX

```
class MLSpatMod : public SpatialModel
{
 protected:
  // specification:
  int np;                       // number of data locations
  Handle(VarStructure) vstr;    // structure of variation
  Handle(Trend) trend;          // parametric model for trend
  double tol, lltol;            // tolerances: lltol for loglikelihood,
                                // tol for parameters.

  // scratch matrices and vectors:
  SymMatrix Sigma;              // covariance matrix for data
  TriangMatrix L;               // Cholesky factor of Sigma, Sigma=LLt
  nMatrix F;                    // regressor matrix
  nMatrix InfoMatrix;           // information matrix
  nVector initVParam;           // initial values of covariance parameters
  SymMatrix TrendCov;           // covariance matrix for trend coefficients
  nMatrix LinvZ;                // inv(L)*z; z = Y-F*beta
  double ztSz;                  // z*inv(Sigma)*z

  BooLean nugget;               // indicator of whether nugget effect should
                                // be estimated
  BooLean TrendEst;             // indicator of whether trend parameters
                                // should be estimated
  BooLean fitted;               // dpTRUE if the model is estimated

  // member functions:
  SymMatrix derivCorr ();            // derivatives of covariance matrix
  virtual void updateDeriv ();       // first and second derivatives
                                     // of loglikelihood function
  virtual nVector estimateTrend ();  // trend parameters
  void Fmatr();                      // initializes regressor matrix

  void setup ();                     // decomposes covariance matrix
  double traceAB (nMatrix, nMatrix); // trace of (A*B)
  void traceS(double&, double&);     // trace of data covariance matrix

  // checking validity of new theta:
  BooLean validTheta(nVector& theta, const nVector& prev);

  double minimumDiffer();            // minimum coordinate difference
  void resetVarStr (nVector th);     // resets covariance parameters

  virtual double logl();             // loglikelihood function

  // first and second derivatives with respect to the covariance parameters:
```

```
  nVector firstDeriv;
  nMatrix secondDeriv;

 public:
  MLSpatMod (const Trend& tr, const VarStructure& vs,
               const SpatialData& dt, BooLean trendEst = dpTRUE,
               BooLean nug=dpFALSE, double lltol=1.0e-4, double mahatol=1e-2);
  MLSpatMod ();
  virtual ~MLSpatMod () { ;}

  void estimate ();                 // estimates model parameters

  // returns covariance and trend parameters:
  nVector getVarParam () const { return vstr->getParam(); }
         // nugget effect, partial sill and range
  nVector getTrendCoef () const { return trend->getCoef(); }
  void getModelEst(SpatialModelDescr&);   // gets all model parameters

  virtual nMatrix infoMatrix ();     // information matrix

  BooLean modelFitted() const { return fitted; }
  void print (Os os) const;          // prints estimated model parameters
};
```

KEYWORDS

covariance, maximum likelihood, multinormal distribution, spatial model

DESCRIPTION

The class implements the maximum likelihood estimation of the parameters in spatial regression, for the multinormal distribution. The estimation is performed according to the scoring algorithm suggested in Mardia and Marshall (1984). The covariance and trend parameters are estimated by iteration over two steps. First, the trend parameters `beta` are estimated by generalized least squares,

$$\mathtt{beta} = (\mathtt{F'S^{-1}F})^{-1}\,\mathtt{F'S^{-1}Y},$$

where `F` is the regressor matrix, `S` is the covariance matrix computed from a parametric covariance model, and `Y` is the observed data. In the second step, the covariance parameters `theta` are updated, using

$$\mathtt{theta}_i = \mathtt{theta}_{i-1} + \mathtt{Btheta}^{-1}\,\mathtt{Ltheta},$$

where `Btheta` is the theta-block of the information matrix `B`,

$$\mathtt{B} = \mathtt{diag(Bbeta,Btheta)},$$

and `Ltheta` is the vector of first derivatives of the loglikelihood function, with respect to the covariance parameters. The matrices `Bbeta` and `Btheta` are the expectations of the

second derivative matrices of the loglikelihood with respect to the trend cofficients (`beta`) and the covariance parameters (`theta`) respectively.

Optionally, the trend coefficients or the nugget effect of the covariance model can be kept constant during the estimation. This is specified by the initialization of an object of type `MLSpatMod`, as described below.

The stopping criterion for the iterative algorithm is a difference in loglikelihood for two successive iterations less than a tolerance, specified as a constructor argument, or a number of iterations greater than 15. By default, the tolerance for the loglikelihood is 1e-4. However, if the difference in trend coefficient or covariance parameter values, measured in Mahalanobis distance divided by the number of parameters, is greater than a tolerance, the program continues. This tolerance is 1e-2 by default. Other values can be specified at the initialization of a `MLSpatMod` object.

REFERENCE:

Mardia, K.V., and Marshall, R.J.: "Maximum likelihood estimation of models for residual covariance in spatial regression", Biometrica, `71`(1), 1984, pp.135-146.

CONSTRUCTORS AND INITIALIZATION

The class has a default constructor, initializing all members to zero, and a constructor with six arguments. These are the specification of the trend model, the covariance structure and the data followed by some optional parameters. The BooLean variable `nug` specifies whether (dpTRUE) or not (dpFALSE) the nugget effect is to be estimated, and `trendEst` whether (dpTRUE) or not (dpFALSE) the trend coefficients should be estimated. By default, the nugget effect is kept constant, equal to the initial value, and the trend coefficients are estimated.

The variables `lltol` and `mahatol` are the tolerances for the difference in loglikelihood and in Mahalanobis distance divided by the number of parameters for two successive iterations.

Initial values for the covariance parameters are needed. From these values, the initial values of the trend coefficients are computed by generalized least squares.

MEMBER FUNCTIONS

See also the SYNTAX section.

`estimate` - estimates the covariance parameters and trend coefficients by the method described above. The member objects specifying the trend and covariance structure are updated at each step. To get the estimated parameter values, use the member functions `getVarParam`, `getTrendCoef` or `getModelEst`. The information matrix for the estimated parameters can be computed by the function `infoMatrix`.

`estimateTrend` - estimates the trend coefficients and their corresponding covariance matrix, using the current values of the covariance parameters. The coefficients of the class `Trend` handle member and the `TrendCov` member matrix are updated.

`getModelEst` - gets the estimated parameters and stores them in a `SpatialModelDescr` object. This virtual function is inherited from the base class `SpatialModel`, and is needed to get access to the estimated parameters when the member `estimate` is invoked from a pointer to the base class `SpatialModel`. Alternatively, the parameters can be returned by the member functions `getVarParam` and `getTrendCoef`.

`infoMatrix` - returns the information matrix `B`, as defined above, for the estimated parameters.

`logl` - computes the value of the loglikelihood for the current parameters. The protected data members `LinvZ` and `ztSz` are updated by this function.

`print` - prints the estimated parameters and the information matrix to an output stream. If the member `infoMatrix` is not invoked on beforehand, the information matrix will be zero. The class `Os` is a generalization of the standard `ostream` class, and supports automatic conversion from an `ostream` object to an `Os` object.

`updateDeriv` - computes the first derivative vector and second derivative matrix of the loglikelihood function, with respect to the unknown covariance parameters. The updated derivatives are stored in the data members `firstDeriv` and `secondDeriv`.

FILES

MLSpatMod.C

EXAMPLE

```
#include <VarStructure.h>
#include <SpatialData.h>
#include <TrendMat.h>
#include <MLpriorSpatMod.h>

main(int argc, char* argv[])
{
  int np = 64, d=2;
  int ntr, model;
  ntr = atoi(argv[1]);               // number of regressors of trend model
  model = atoi(argv[2]);             // with (1) or without(0) prior

  BooLean nugEst = dpTRUE;           // nugget effect should be estimated
  BooLean trendEst = dpTRUE;         // trend coef. should be estimated

  double ltol=1e-4;                  // tolerance in loglikelihood
  double mtol=1e-2;                  // tolerance in parameter values

  // data:
  IrregSpatialData sdata(np,d);      // data observations
  sdata.scan("FILE=data.file");      // reads data from a file

  // model for structure of variation:
  char* name = "exponential";        // type of covariance model
  VarStructure vstr(name,3);         // structure of variation
  nVector param(3);                  // initial values of parameters
  param.scan("FILE=var.param");      // reads parameters from a file
  // set parameters of VarStructure object:
  vstr.setParam(1,param(1));         // nugget effect
  vstr.setParam(2,param(2)-param(1)); // partial sill
  vstr.setParam(3,param(3));         // range
  vstr.setLower(2,0.001);            // lower bound for partial sill

  // trend model:
  nMatrix F(np,ntr);                 // regressor matrix
  nVector trcoef(ntr);               // trend coefficients
  F.scan("FILE=Xmatr.dat");          // reads regressor matrix from a file
  trcoef.scan("FILE=trend.coef");    // reads trend coefficients from a file

  TrendMat trmodel(F);               // creates object of type TrendMat
  trmodel.setCoef(trcoef);           // sets trend coefficients

  // covariance matrix for trend coefficients (if model with prior)
  nMatrix trs(ntr,1);
```

```
  if ((model==1)&&(trendEst))
    trs.scan("FILE=trend.var");

  // creates estimation object:
  MLSpatMod* svest;
  if (model == 1){     // model with prior
    if (trendEst)
      svest = new MLpriorSpatMod(trmodel,vstr,sdata,trs,nugEst,ltol,mtol);
    else
      svest = new MLpriorSpatMod(trmodel,vstr,sdata,nugEst,ltol,mtol);
  }
  else if (model == 0) // model without prior
    svest = new MLSpatMod(trmodel,vstr,sdata,trendEst,nugEst,ltol,mtol);

  svest->estimate();                 // estimates parameters
  svest->infoMatrix();               // computes information matrix
  svest->print("FILE=ML.results");   // prints results to a file

  delete svest;                      // deletes MLSpatMod pointer
}
```

SEEALSO

class MLpriorSpatMod, class SpatialData, class SpatialModel, class Trend, class VarStructure

AUTHOR

Turid Follestad, NR

4.3.2 MLpriorSpatMod

NAME

MLpriorSpatMod - a class for maximum likelihood estimation for spatial model parameters, using prior information

INCLUDE

```
include "MLpriorSpatMod.h"
```

SYNTAX

```
class MLpriorSpatMod : public MLSpatMod
{
protected:
  nMatrix sigma0;      // covariance matrix (or vector of variances) for beta0
  nVector beta0;       // prior values for trend coefficients (beta)
  double detSigma0;    // determinant of sigma0
  BooLean uncorrBeta;  // incicator of correlated/uncorrelated beta's

  double rpow;         // power (s) for range prior, prop. to 1/(1+R)^s

  void updateDeriv (); // updates derivatives
  nVector estimateTrend ();  // estimates trend coefficients

  double logl();       // computes loglikelihood for current parameters

public:
  MLpriorSpatMod (const Trend& tr, const VarStructure& vs,
                  const SpatialData& dt, nMatrix trsigma,
                  BooLean nug=dpFALSE, double lltol=1.0e-4,
                  double mahatol=1e-2, double rpow = 2);
  MLpriorSpatMod (const Trend& tr, const VarStructure& vs,
                  const SpatialData& dt,
                  BooLean nug=dpFALSE, double lltol=1.0e-4,
                  double mahatol=1e-2, double rpow = 2);

  MLpriorSpatMod ()
    : sigma0(), beta0() { uncorrBeta = dpTRUE; detSigma0 = 0.0;}
  ~MLpriorSpatMod () { ;}

  nMatrix infoMatrix (); // computes information matrix
};
```

KEYWORDS

covariance, maximum likelihood, multinormal distribution, priors, spatial model

DESCRIPTION

The class implements the maximum likelihood estimation of the parmeters in spatial regression, using prior information, for the multinormal distribution.

The estimation is performed according to the scoring algorithm suggested in Mardia and Marshall (1984), as described in the documentation of the base class `MLSpatMod`, but adding

priors for the trend and covariance parameters to the loglikelihood function. The initial guess for the distribution of the trend coefficients is

```
beta ~ N(beta0,Sigma0)
```

where `beta0` is the expectation and `Sigma0` the covariance matrix of the trend coefficients. These values must be given initially.

The prior distribution for the variance, `sigma2` (= nugget effect plus partial sill), is assumed to be proportional to

```
1/sigma2,
```

and the prior distribution for the range, `R`, is proportional to

```
       k
1/(1+R) ,
```

where `k` can is specified by the user. The default is `k=2`.

To use another prior, the user can derive a new class from `MLSpatMod`, re-defining the virtual member functions `logl`, `updateDeriv` and `estimateTrend`.

See also the documentation of class `MLSpatMod`, and `CONSTRUCTORS AND INITIALIZATION` below.

REFERENCE:

Mardia, K.V., and Marshall, R.J.: "Maximum likelihood estimation of models for residual covariance in spatial regression", Biometrica, 71(1), 1984, pp.135-146.

CONSTRUCTORS AND INITIALIZATION

The class has a default constructor, initializing all members to zero, and two other constructors having seven and eight arguments. These are the specification of the trend model (class `Trend`), the covariance structure (class `VarStructure`) and the data (class `SpatialData`), followed by some optional parameters. The actual argument of type class `VarStructure` reference is expected to hold the initial values of the covariance parameters nugget effect, partial sill and range. The coefficient vector of the class `Trend` reference argument is expected to be initialized by the expectation `beta0` of the prior distribution of the trend coefficients. Initial values for the trend coefficient estimation are computed from the initial values of the covariance parameters.

The `nMatrix` argument `trsigma` of the first constructor specifies the covariance matrix `Sigma0` for the trend prior distribution, and this constructor should be used when the trend coefficients, as well as the covariance parameters, are to be estimated. If only the covariance parameters are unkown, the matrix `Sigma0` is not needed, and the second constructor is expected to be used. The covariance parameters are in this case estimated based on the initial values of the trend coefficients.

The BooLean variable `nug` specifies whether or not the nugget effect is to be estimated, and `lltol` and `mahatol` are the tolerances for the difference in loglikelihood and in Mahalanobis distance divided by the number of parameters for two successive iterations.

MEMBER FUNCTIONS

See also the base class `MLSpatMod`.

`infoMatrix` - returns the information matrix for the estimated parameters. The information matrix, `B`, is computed as for the ordinary maximum likelihood model in class `MlSpatMod`, but using the loglikelihood adjusted for the priors for the parameters.

FILES

MLpriorSpatMod.C

EXAMPLE

See class MLSpatMod

SEEALSO

class MLSpatMod, class SpatialData, class SpatialModel, class Trend, class VarStructure

AUTHOR

Turid Follestad, NR

Chapter 5

Spatial prediction

5.1 Abstract base class

5.1.1 SpatialPred

NAME

SpatialPred - an abstract base class for spatial prediction

INCLUDE

```
include "SpatialPred.h"
```

SYNTAX

```
class SpatialPred
{
public:
  virtual ~SpatialPred() { }
  virtual void predict(nMatrix&)=0;
};
```

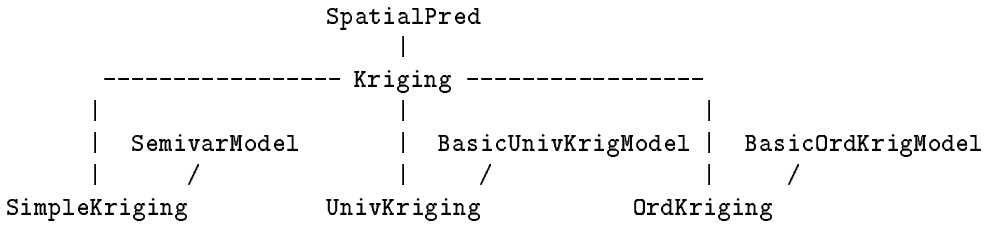
KEYWORDS

spatial model, spatial prediction

DESCRIPTION

The class is an abstract base class for classes implementing spatial prediction. The model parameters on which the predictions are based, can be estimated by using one of the classes in the `SpatialModel` hierarchy.

The classes in the hierarchy implements optimal linear prediction by the method of kriging, and the class hierarchy is organized as follows:



The class `Kriging` is an abstract class. Further specifications of the methods are given in the documentation of each of the derived classes.

CONSTRUCTORS AND INITIALIZATION

No constructors.

MEMBER FUNCTIONS

`predict` - virtual function performing spatial prediction, returning the result as a `nMatrix` reference argument.

FILES

None.

EXAMPLE

See class Kriging

SEEALSO

class Kriging, class SpatialModel

AUTHOR

Turid Follestad and Jon Helgeland, NR

5.2 Kriging

5.2.1 Kriging

NAME

Kriging - an abstract base class for linear spatial prediction by the method of kriging

INCLUDE

```
include "Kriging.h"
```

SYNTAX

```
class Kriging: public SpatialPred
{
 protected:
  BooLean stdev;   // indicator of whether prediction stdev. should be computed
  BooLean paramOK; // indicator of whether parameters should be estimated

  Handle(PointSet) ps;        // prediction domain
  Handle(SpatialData) pd;     // data observations
  Handle(VarStructure) pvmod; // parametric model for structure of variation

  // computation of vector of covariances:
  virtual nVector covarVector(const nVector& x);

 public:
  Kriging(const PointSet& ps);
  virtual ~Kriging() { ;}      // Handle deletes members

  // indicates whether prediction stdev. should be computed:
  void computeStdev(BooLean std) { stdev = std;}

  // indicates whether parameters is to be estimated:
  void setParamOK(BooLean b) { paramOK = b; }
};
```

KEYWORDS

kriging, linear spatial prediction, spatial prediction

DESCRIPTION

The class is an abstract base class for implementations of optimal linear spatial prediction by the method of kriging. The spatial model considered, is defined in the documentation of the class `KrigingModel`. The kriging predictor, `Zhat(x0)`, in a data location `x0`, is

$$\texttt{Zhat(x0)} = \texttt{m(x0,b)} + \texttt{k'(x0)*Sigma}^{-1}\texttt{(Zdata-mdata)}.$$

Here, `m(x0,b)` is the detetministic trend function with parameter vector `b`, `k(x0)` the vector of covariances of the residual `e(x)` between `x0` and the data locations for the observed data, and `Sigma` is the data covariance matrix. Further, `Zdata` is the vector of observed data, and `mdata` is a vector of the values of `m(x,b)` in the corresponding data locations.

The members are referenced through the use of smart pointers, i.e. handles, for simplified memory management. A handle consists of a pointer to the object and some interface functions.

REFERENCES:

Cressie, N.: "Statistics for Spatial Data", Wiley, New York, 1991.

Journel, A.G. and Huijbregts, Ch.J.: "Mining Geostatistics", Academic Press, London, 1978.

Ripley, B.D.: "Spatial Statistics", Wiley, New York, 1981.

CONSTRUCTORS AND INITIALIZATION

The class has one constructor taking an argument of type `PointSet`, representing the prediction domain. The `SpatialData` and `VarStructure` members must be initialized by the derived classes.

MEMBER FUNCTIONS

`computeStdev` - By this function, the user can specify whether the prediction standard deviance should be computed, in addition to the predicted values. The reason for giving this option, is that the computation of the standard deviances is the computationally most expensive part of the prediction routine. The default is dpTRUE.

`setParamOK` - specifies whether the parameters specifying the model are estimated on beforehand (`b`=dpTRUE). These parameter values should be specified in the constructor arguments. The default argument value is dpFALSE.

Protected functions to be used in computations in derived classes:

`covarVector` - computes the vector of covariance values between the observed data locations and the prediction point `x`.

FILES

Kriging.C

EXAMPLE

```
#include <UnivKriging.h>
#include <OrdKriging.h>
#include <SimpleKriging.h>
#include <PolTrend.h>

main(int argc, char* argv[])
{
  int trendorder = atoi(argv[1]);       // order of polynomial trend
  int trendKnown = atoi(argv[2]);       // if 0: trend parameters are unknown
                                        // else: trend parameters are known
  // data observations:
```

```
  int np = 50;                          // number of data locations
  int d = 2;                            // spatial dimension
  IrregSpatialData data(np,d);          // irregularly spaced data
  data.scan("FILE=dataset.dat");        // reads data from a file

  // prediction domain, irregularly spaced points:

  GenPointSet predpoints(np,d);         // general set of points
  predpoints.scan("FILE=point.set");    // reads point set from a file

  // model for structure of variation:

  VarStructure vmodel("bessel");        // "bessel" type covariance model
  nVector initparam(3);                 // initial parameter values
  initparam.scan("FILE=vparam.dat");    // reads parameters
  vmodel.setParam(initparam);           // sets parameters of vmodel

  // trend model:

  PolTrend trend(trendorder,d);         // polynomial trend model
                                        // - trend coefficients=0 by default

  // creation of an object of a class in the Kriging hierarcy,
  // using default options: parameters are to be estimated, and
  // prediction standard deviances are computed:

  Kriging* kriging;                     // pointer to class Kriging object

  if (!trendKnown){     // unknown trend parameters
    if (trendorder==0)  // constant trend -> ordinary kriging
      kriging = new OrdKriging (predpoints,data,vmodel);
    else                // trend not constant -> universal kriging
      kriging = new UnivKriging (predpoints,data,trend,vmodel);
  }
  else{                 // trend model is known -> simple kriging
    kriging = new SimpleKriging (predpoints,data,trend,vmodel);
  }

  nMatrix res(predpoints.getNpoints(),1);
  kriging->predict(res);                // prediction

  ofstream out("predres.dat",ios::out);
  res.print(out);                       // prints predicted surface to a file

  delete kriging;                       // deletes class Kriging pointer
}
```

SEEALSO

class KrigingModel, class PointSet, class SpatialPred

AUTHOR

Turid Follestad and Jon Helgeland, NR

5.2.2 SimpleKriging

NAME

SimpleKriging - a class for linear spatial prediction by the method of simple kriging

INCLUDE

```
include "SimpleKriging.h"
```

SYNTAX

```
class SimpleKriging: public Kriging, public SemivarModel
{
 private:
  Handle(Trend) ptrend;                // constant parametric trend model

  TriangMatrix SigmaL;                 // Cholesky factor of covariance matrix
  nIntVector pivL;                     // pivot vector, Cholesky factorization
  void setup();                        // decompose covariance matrix

  nVector fluctCoef(const nVector& z); // computes a prediction constant
  double fluctVar(const nVector& cv);  // variance of stochastic part

 public:
  SimpleKriging(const PointSet&, const SpatialData&, const Trend&,
                const VarStructure& vmodel, const NonParSemivar&, int minl=30);
  SimpleKriging(const PointSet&, const SpatialData&, const Trend&,
                const VarStructure& vmodel, double lsz=0.0, int minl=30);
  ~SimpleKriging();

 void predict(nMatrix& Y);            // computes predicted values
};
```

KEYWORDS

kriging, optimal prediction, simple kriging, spatial prediction

DESCRIPTION

The class contains a procedure for optimal linear spatial prediction by the method of simple kriging. Simple kriging is the special case of kriging where the coefficients of the trend function `m(x,b)` of the spatial model, as defined in the documentation of class `KrigingModel`, are known. The known trend coefficients should be specified in the `Trend` reference argument of the constructor. If the trend coefficients are known on beforehand, but estimated from the data, class `OrdKriging` should be used.

By default, the prediction routine, `predict`, first invokes the inherited member function `SemivarModel::estimate` to estimate the semivariogram parameters, and then the predicted surface is computed. If the semivariogram parameters are estimated before the creation of the `SimpleKriging` object, these can be specified at initialization by the parameters of the `VarStructure` object of the constructor. To avoid re-estimating these parameters, the member function `setParamOK`, inherited from class `Kriging`, should be invoked, setting its actual argument equal dpTRUE.

CONSTRUCTORS AND INITIALIZATION

The class has two constructors. The prediction domain and the observed spatial data are specified by objects in the `PointSet` and `SpatialData` hierarchies respectively. It is assumed that the `SpatialData` object contains only *one* response variable, if not, the prediction is performed for the first variable. The model for the structure of variation should also be specified, by a `VarStructure` object. If the semivariogram parameters are estimated on beforehand, the only additional parameter needed as input to the constructor, is the specification of the constant trend model, by an object in the `Trend` hierarchy. In this case, the prediction is based on the semivariogram parameters specified as a part of the `VarStructure` object.

If the semivarigram parameters are to be estimated prior to the spatial prediction, a non-parametric semivariogram needs to be specified. This can be done either by specifying a `NonParSemivar` object, corresponding to the observed data, or a lagsize (`lsz`). If the default lagsize (=0.0) is chosen, the lagsize is estimated from the number of observations, as described in the documentation of class `NonParSemivar`. The integer argument `minl`, with default value 30, specifies the minimum number of pairs of points to be allowed in each lag interval for the weighted least squares fitting of the semivariogram.

MEMBER FUNCTIONS

`predict` - computes predicted values and corresponding standard deviances in the spatial point set specified by the `PointSet` constructor argument. If the parameters are not known in advance, the routine will call the member function `estimate` inherited from `SemivarModel`, unless this function is explicitly called on beforehand.

The prediction standard deviances are not computed if the `computeStdev` function, inherited form class `Kriging`, is called with argument dpFALSE.

FILES

SimpleKriging.C

EXAMPLE

See class Kriging

SEEALSO

class Kriging, class KrigingModel, class PointSet, class SpatialData, class Trend, class VarStructure

AUTHOR

Turid Follestad, NR

5.2.3 OrdKriging

NAME

OrdKriging - a class for optimal linear prediction by the method of ordinary kriging

INCLUDE

```
include "OrdKriging.h"
```

SYNTAX

```
class OrdKriging: public Kriging, public BasicOrdKrigModel
{
 protected:
  nVector fluctCoef(const nVector& y); // computes a prediction constant
  double fluctVar(const nVector& cv);  // variance of stochastic part
  nVector trendVarC();                 // prediction variance constant

 public:
  OrdKriging(const PointSet&, const SpatialData&, const VarStructure& vm,
             const NonParSemivar& nps, int minl=30);
  OrdKriging(const PointSet&, const SpatialData&, const VarStructure& vm,
             double lsz=0.0, int minl=30);

  void predict(nMatrix& Y);            // computes predicted values
};
```

KEYWORDS

kriging, optimal prediction, ordinary kriging, spatial prediction

DESCRIPTION

The class contains a procedure for optimal linear spatial prediction by the method of ordinary kriging. It is derived from the classes `Kriging` and `BasicOrdKrigModel`, and by calling the member function `estimate` of the last class, the model parameters can be estimated.

Ordinary kriging is the special case of kriging where the trend function of the spatial model is an unknown constant, `b0`, as described in the documentation of class `BasicOrdKrigModel`. The kriging predictor is as defined in the documentation of class `Kriging`, with `m(x0,b) = b0`.

By default, the prediction routine, `predict`, first invokes the inherited member function `BasicOrdKrigModel::estimate` to estimate the parameters, and then the predicted surface is computed. If the semivariogram parameters are estimated before the creation of the `OrdKriging` object, these can be specified at initialization by the parameters to the `VarStructure` object. To avoid re-estimating these parameters, the member function `setParamOK`, inherited from class `Kriging`, should be invoked, setting its actual argument equal dpTRUE. If the trend coefficients are known, the method implemented in class `SimpleKriging` could be used.

CONSTRUCTORS AND INITIALIZATION

The class has two constructors. The prediction domain and the observed spatial data are specified by objects in the `PointSet` and `SpatialData` hierarchies respectively. It is assumed that the `SpatialData` object contains only *one* response variable, if not, the prediction is performed for the first variable. The model for the structure of variation should also be specified, by a `VarStructure` object.

If the semivariogram parameters are estimated on beforehand, the prediction is based on the semivariogram parameters specified as a part of the `VarStructure` object. If the semivariogram parameters are to be estimated by the `predict` member function, prior to the spatial prediction, a non-parametric semivariogram needs to be specified. This can be done either by specifying a `NonParSemivar` object, corresponding to the observed data, or a lagsize (`lsz`). If the default lagsize (=0.0) is chosen, the lagsize is estimated from the number of observations, as described in the documentation of class `NonParSemivar`. The integer argument `minl`, with default value 30, specifies the minimum number of pairs of points to be allowed in each lag interval for the weighted least squares fitting of the semivariogram.

MEMBER FUNCTIONS

`predict` - computes predicted values and corresponding standard deviances in the spatial point set specified by the `PointSet` constructor argument. If the parameters are not known in advance, the routine will call the member function `estimate` inherited from `BasicOrdKrigModel`, unless this function is explicitly called on beforehand. The prediction standard deviances are not computed if the `computeStdev` function, inherited form class `Kriging`, is called with argument dpFALSE.

Protected member functions, invoked from `predict`:

`fluctCoef` - computes the value `transp(z)*inv(Sigma)`, where the `nVector` argument `z` is the residuals (response values minus trend), and `Sigma` the covariance matrix for the data locations. This value is constant under the prediction. The argument is a vector of response values.

`fluctVar` - computes the part of the prediction variance corresponding to the stochastic part of the model, in the point giving raise to the covariance vector to be entered as an argument.

FILES

OrdKriging.C

EXAMPLE

See class Kriging

SEEALSO

class BasicOrdKrigModel, class Kriging, class NonParSemivar, class PointSet, class SpatialData, class VarStructure

AUTHOR

Turid Follestad and Jon Helgeland, NR

5.2.4 UnivKriging

NAME

UnivKriging - a class for optimal linear spatial prediction by the method of universal kriging

INCLUDE

```
include "UnivKriging.h"
```

SYNTAX

```
class UnivKriging: public Kriging, public BasicUnivKrigModel
{
 protected:
  nVector fluctCoef(const nVector& y);   // computes a prediction constant
  nMatrix trendVarC();                   // prediction variance constant
  double fluctVar(const nVector& cvec);  // variance of stochastic part

 public:
  UnivKriging(const PointSet&, const SpatialData&, const Trend&,
              const VarStructure& vmodel2, double lsz=0.0, int minl=30);

  void predict(nMatrix& Y);              // computes predicted values
};
```

KEYWORDS

spatial prediction, optimal prediction, kriging, universal kriging

DESCRIPTION

The class contains a procedure for optimal linear spatial prediction by the method of universal kriging. It is derived from the classes **Kriging** and **BasicUnivKrigModel**, and by calling the member function **estimate** of the last class, the model parameters can be estimated.

Universal kriging is optimal linear spatial prediction using a general trend function for the spatial model, as described in the documentation of class **BasicUnivKrigModel**. The kriging predictor is as defined in the documentation of class **Kriging**, with

```
m(x0,b) = f'(x0)b.
```

By default, the prediction routine, **predict**, first invokes the inherited member function **BasicUnivKrigModel::estimate** to estimate the parameters, and then the predicted surface is computed. If the semivariogram parameters are estimated before the creation of the **UnivKriging** object, these can be specified at initialization by the parameters to the **VarStructure** object. To avoid re-estimating these parameters, the member function **setParamOK**, inherited from class **Kriging**, should be invoked, setting its actual argument equal dpTRUE. If the trend coefficients are known, the method implemented in class **SimpleKriging** could be used.

CONSTRUCTORS AND INITIALIZATION

The class has one constructor. The prediction domain and the observed spatial data are specified by objects in the `PointSet` and `SpatialData` hierarchies, respectively. It is assumed that the `SpatialData` object contains only *one* response variable, if not, the prediction is performed for the first variable. The models for the structure of variation and trend should also be specified, by a `VarStructure` object and an object of one of the classes in the `Trend` hierarchy, respectively.

If the semivariogram parameters are estimated on beforehand, no additional constructor arguments are needed. In this case, the prediction is based on the semivariogram parameters specified as a part of the `VarStructure` object. If the semivariogram parameters are to be estimated by the `predict` member function, prior to the spatial prediction, the lagsize of the non-parametric semivariogram to be computed during the semivariogram parameter estimation, is needed. This can be specified by the double argument `lsz`. If the default lagsize (=0.0) is chosen, the lagsize is estimated from the number of observations, as described in the documentation of class `NonParSemivar`. The integer argument `minl`, with default value 30, specifies the minimum number of pairs of points to be allowed in each lag interval for the weighted least squares fitting of the semivariogram.

MEMBER FUNCTIONS

`predict` - computes predicted values and corresponding standard deviances in the spatial point set specified by the `PointSet` constructor argument. If the parameters are not known in advance, the routine will call the member function `estimate` inherited from `BasicUnivKrigModel`, unless this function is explicitly called on beforehand. The prediction standard deviances are not computed if the `computeStdev` function, inherited form class `Kriging`, is called with argument dpFALSE.

Protected member functions, invoked from `predict`:

`fluctCoef` - computes the value `transp(z)*inv(Sigma)`, where the `nVector` argument `z` is the residuals (response values minus trend), and `Sigma` the covariance matrix for the data locations. This value is constant under the prediction. The argument is a vector of response values.

`fluctVar` - computes the part of the prediction variance corresponding to the stochastic part of the model, in the point giving raise to the covariance vector to be entered as an argument.

FILES

UnivKriging.C

EXAMPLE

See class Kriging

SEEALSO

class BasicUnivKrigModel, class Kriging, class NonParSemivar, class PointSet, class SpatialData, class Trend, class VarStructure

AUTHOR

Turid Follestad and Jon Helgeland, NR

Chapter 6

Spatial simulation

6.1 Abstract base class

6.1.1 SpatialSim

NAME

SpatialSim - an abstract base class for spatial simulation

INCLUDE

```
include "SpatialSim.h"
```

SYNTAX

```
class SpatialSim{
 protected:
   RandomCont* rand;         // random number generator
   void initRand(int seed);  // initialization of random generator

 public:
   SpatialSim (int seed);
   SpatialSim () {rand=NULL;}
   virtual ~SpatialSim() { if (rand!=NULL) delete rand;}
   virtual void simulate(nMatrix&) = 0;
};
```

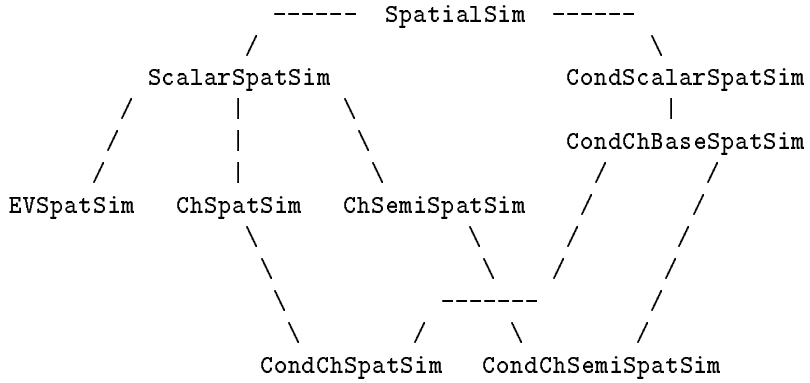
KEYWORDS

gaussian field, simulation, spatial model, stochastic field

DESCRIPTION

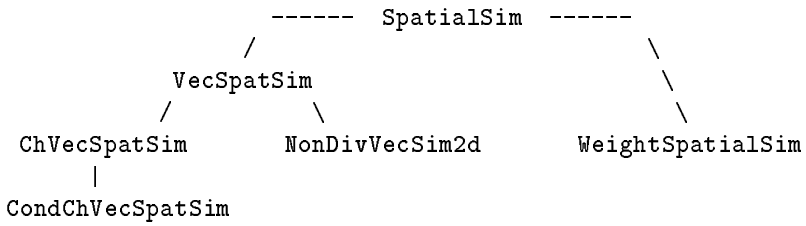
The class is an abstract base class for methods for simulation of spatial gaussian stochastic fields. In the hierarchy, methods for unconditional and conditional simulation are included. The methods are implemented for one-dimensional gaussian fields, and non-divergent two-dimensional gaussian vector fields.

The branch of the hierarcy implementing simulation of scalar fields, is organized as follows:



The classes `ScalarSpatSim` and `CondScalarSpatSim` are abstract base classes for unconditional and conditional simulation, and class `CondChBaseSpatSim` is an abstract base class for conditional simulation using the Cholesky decomposition.

The class hierarchy for the simulation of vector fields is



Here, `VecSpatSim` is an abstract base class. The classes `ChVecSpatSim`, `NonDivVecSim2d` and `WeightSpatialSim` all implements different methods for unconditional simulation, and in `CondChVecSpatSim`, a method for conditional simulation is implemented.

For a further description of the methods, see the documentation of the different classes.

CONSTRUCTORS AND INITIALIZATION

The class has a default constructor not allocating any memory, and a constructor taking an integer argument specifying the seed for the random number generator. If no non-zero value is specified, the seed will be taken to be the point of time.

MEMBER FUNCTIONS

`simulate` - virtual function simulating a stochastic field, returning the simulated values through the `nMatrix` reference argument.

FILES

SpatialSim.C

EXAMPLE

See class ScalarSpatSim

SEEALSO

class RandomCont, class RandStdNormal

AUTHOR

Turid Follestad, NR

6.2 Simulation of scalar fields

6.2.1 ScalarSpatSim

NAME

ScalarSpatSim - an abstract base class for linear spatial simulation of one-dimensional gaussian fields

INCLUDE

```
include "ScalarSpatSim.h"
```

SYNTAX

```
class ScalarSpatSim: public SpatialSim
{
 protected:
  Handle(PointSet) points;  // simulation domain

  // initialization and decomposition:
  virtual void setup(const Trend&, const VarStructure&)=0;

public:
  ScalarSpatSim(const PointSet&, int seed=0);
  ScalarSpatSim();
  ~ScalarSpatSim() {;}

  virtual void reset(const Trend&, const VarStructure&, const PointSet&,
             int seed=0);   // resets model parameters
};
```

KEYWORDS

gaussian field, stochastic field, simulation

DESCRIPTION

The class is an abstract base class for simulation of a one-dimensional gaussian stochastic field, `Z`, defined by

```
Z(x) = m(b,x) + e(x),
```

where `b` is a vector of trend coefficients, `m` is the trend function, and the residuals `e(x)` are `N(0,Sigma)`. The simulation is based on parametric models for the trend and structure of variation for the residuals. These models are defined by objects of type class `Trend` and class `VarStructure`.

The field is simulated in a general set of points or a grid lattice, initialized by the `PointSet` reference constructor argument. The member is referenced through the use of smart pointers, i.e. handles, for simplified memory management. A handle consists of a pointer to the object and some interface functions.

CONSTRUCTORS AND INITIALIZATION

There are two constructors, including a default constructor not allocating any memory. The simulation domain is determined by the `PointSet` argument. The integer argument specifies the seed for the random number generator. If no non-zero value is specified, the seed will be taken to be the point of time.

MEMBER FUNCTIONS

`reset` - virtual function that resets the trend, the structure of variation and the simulation domain. A non-zero value of the seed can be specified by the integer argument. By default, the point of time is taken as the seed value. The function is intended for use when an object is created using the default constructor. If the other constructor has been used, a specification of a seed for the random number generator will be ignored, and the random generator attached to the global random stream already created.

`setup` - virtual function that is to initialize the parameters of the first and second order moments of the distribution for the simulation. Two arguments, of type class `Trend` and class `VarStructure` are expected.

FILES

ScalarSpatSim.C

EXAMPLE

```
/************************************************************************/
/*                                                                      */
/* Simulation of a scalar spatial field by the Cholesky decomposition  */
/* method, a modfied Cholesky decomposition method, or by eigenvalue   */
/* decomposition of the covariance matrix.                              */
/*                                                                      */
/************************************************************************/
#include <ChSpatSim.h>
#include <ChSemiSpatSim.h>
#include <EVSpatSim.h>
#include <PolTrend.h>

main(int argc, char* argv[])
{
  int nsim = atoi(argv[1]);    // number of simulations
  int npoints = atoi(argv[2]); // number of points in simuation domain
  int dim = atoi(argv[3]);     // spatial dimension
  int seed = atoi(argv[4]);    // seed for random number generator
  char* method;                // simulation method:
                               // "EV"=eigenvalue decomposition
                               // "Chol"= ordinary Cholesky decomposition
                               // "ModCh"= modified Cholesky decomposition
  method = argv[5];

  // simulation domain:

  ifstream infile("point.set",ios::in);
  GenPointSet pset(npoints,dim);
  pset.scan(infile);
  infile.close();

  // structure of variation:
```

```
  char* vmtype = "exponential";
  nVector covparam(3);
  covparam(1) = 0.0;           // nugget effect
  covparam(2) = 1.0;           // partial sill
  covparam(3) = 0.2;           // range

  VarStructure vmodel(vmtype);
  vmodel.setParam(covparam);

  // constant trend:

  PolTrend ptr(0,dim);
  nVector trendval(1,1.0);     // constant trend = 1.0
  ptr.setCoef(trendval);

  // creating simulation object:

  ScalarSpatSim* sim;
  if (!strncmp(method,"Ch",2))
    sim = new ChSpatSim(ptr,vmodel,pset,seed);
  else if (!strncmp(method,"ModCh",5))
    sim = new ChSemiSpatSim(ptr,vmodel,pset,seed);
  else if (!strncmp(method,"EV",2))
    sim = new EVSpatSim(ptr,vmodel,pset,seed);

  // simulation:

  nMatrix res1(pset.getNpoints(),1);
  nMatrix res(pset.getNpoints(),nsim);

  if (sim!=NULL){
    for (int i=1;i<=nsim;i++){
      sim->simulate(res1);
      res.setCol(i,res1.col(1));
    }
  }

  ofstream outfile("simres",ios::out);
  res.print(outfile);
  outfile.close();

  if (sim!=NULL)
    delete sim;
}
```

SEEALSO

class PointSet, class SpatialSim, class Trend, class VarStructure

AUTHOR

Turid Follestad and Tove Andersen, NR

6.2.2 ChSpatSim

NAME

ChSpatSim - a class for linear spatial simulation of one-dimensional gaussian fields

INCLUDE

```
include "ChSpatSim.h"
```

SYNTAX

```
class ChSpatSim: public virtual ScalarSpatSim
{
 protected:
  nVector trendval;           // trend function values in the simulation domain
  TriangMatrix covmatL;       // Cholesky factor of covariance matrix
  void setup(const Trend&, const VarStructure&);

public:
  ChSpatSim(const Trend&, const VarStructure&, const PointSet&,
               int seed=0);
  ChSpatSim();

  void reset(const Trend&, const VarStructure&, const PointSet&,
            int seed=0);                          // resets model parameters
  TriangMatrix getCovL() const { return covmatL; } // returns Cholesky factor

  void simulate(nMatrix&);
};
```

KEYWORDS

Cholesky decomposition method, gaussian field, simulation, stochastic field

DESCRIPTION

The class is derived from class `ScalarSpatSim`, and implements simulation of a gaussian scalar field based on a Cholesky decomposition of the covariance matrix, `S=L*transp(L)`. For very smooth fields, this method might fail. In such cases, method `ChSemiSpatSim` or `EVSpatSim` could be used.

CONSTRUCTORS AND INITIALIZATION

There are two constructors, including a default constructor not allocating any memory. The parametric model is specified by arguments of type `Trend` and `VarStructure`, and the simulation domain is determined by the `PointSet` argument. The integer argument specifies the seed for the random number generator. If no non-zero value is specified, the seed will be taken to be the point of time.

MEMBER FUNCTIONS

`reset` - resets the object. See the documentation of class `ScalarSpatSim`.

`setup` - initializes the `Trend` and `VarStructure` members, and computes the Cholesky factor of the covariance matrix.

`simulate` - performes one simulation of the random field on the specified point set and according to the model parameters. The resulting vector of simulated values, ordered according to the numbering of the `nps` points of the point set, is returned as a `nps` by one matrix by the `nMatrix` reference argument.

FILES

ChSpatSim.C

EXAMPLE

See class ScalarSpatSim

SEEALSO

class ChSemiSpatSim, class EVSpatSim, class ScalarSpatSim, class PointSet, class Trend, class VarStructure

AUTHOR

Turid Follestad and Tove Andersen, NR

6.2.3 ChSemiSpatSim

NAME

ChSemiSpatSim - a class for linear spatial simulation of one-dimensional gaussian fields

INCLUDE

```
include "ChSemiSpatSim.h"
```

SYNTAX

```
class ChSemiSpatSim: public virtual ScalarSpatSim
{
 protected:
  nIntVector pivot;
  nVector trendval;
  TriangMatrix covmatL;    // pivoted L , LL' = P'SP
  void setup(const Trend&, const VarStructure&);

public:
  ChSemiSpatSim(const Trend&, const VarStructure&, const PointSet&,
                int seed=0);
  ChSemiSpatSim();

  void reset(const Trend&, const VarStructure&, const PointSet&,
             int seed=0);
  TriangMatrix getCovL() const { return covmatL; }
  nIntVector getPivot() const { return pivot; }

  void simulate(nMatrix&);
};
```

KEYWORDS

Cholesky decomposition method, gaussian field, stochastic field, simulation, smooth field

DESCRIPTION

The class is derived from class `ScalarSpatSim`, and implements simulation of a gaussian scalar field based on a Cholesky like decomposition of the covariance matrix, using symmetric pivoting. This method will work for very smooth fields, for which the ordinary Cholesky decomposition method fails. Alternatively, class `EVSpatSim` could be used in such cases.

The Cholesky decomposition is performed by the function `cholFactorSemi` of the matrix library in NUTILITY.

CONSTRUCTORS AND INITIALIZATION

There are two constructors, including a default constructor not allocating any memory. The parametric model is specified by arguments of type `Trend` and `VarStructure`, and the simulation domain is determined by the `PointSet` argument. The integer argument

specifies the seed for the random number generator. If no non-zero value is specified, the seed will be taken to be the point of time.

MEMBER FUNCTIONS

`reset` - resets the object. See the documentation of class `ScalarSpatSim`.

`setup` - initializes the `Trend` and `VarStructure` members, and computes the Cholesky factor of the covariance matrix, using symmetric pivoting.

`simulate` - performes one simulation of the random field on the specified point set and according to the model parameters. The resulting vector of simulated values, ordered according to the numbering of the `nps` points of the point set, is returned as a `nps` by one matrix by the `nMatrix` reference argument.

FILES

ChSemiSpatSim.C

EXAMPLE

See class ScalarSpatSim

SEEALSO

class ChSpatSim, class EVSpatSim, class PointSet, class ScalarSpatSim, class Trend, class VarStructure

AUTHOR

Turid Follestad and Tove Andersen, NR

6.2.4 EVSpatSim

NAME

EVSpatSim - a class for linear spatial simulation of one-dimensional gaussian fields

INCLUDE

```
include "EVSpatSim.h"
```

SYNTAX

```
class EVSpatSim: public virtual ScalarSpatSim
{
 protected:
  nMatrix Sigmaevec;         // eigenvectors
  nVector Sigmaeval;         // eigenvalues

  nVector trendval;          // trend function values in the simulation domain

  virtual void setup(const Trend&, const VarStructure&);

public:
  EVSpatSim(const Trend&, const VarStructure&, const PointSet&,
               int seed=0);
  EVSpatSim();

  void reset(const Trend&, const VarStructure&, const PointSet&,
               int seed=0); // resets trend parameters

  void simulate(nMatrix&);
};
```

KEYWORDS

eigenvalues, gaussian field, stochastic field, simulation

DESCRIPTION

The class is derived from class `ScalarSpatSim`, and implements simulation of a gaussian scalar field based on an eigenvalue decomposition of the covariance matrix. Eigenvectors corresponding to eigenvalues close to zero are omitted.

The simulated field, `y`, is computed from a vector, `u`, of `N(0,1)` random numbers, by

```
y = mu + G*Lambda u,
```

where `mu` is the expected value. `Lambda` is a diagonal matrix with the square root of the positive eigenvalues on the diagonal, and the columns of `G` are the corresponding eigenvectors.

CONSTRUCTORS AND INITIALIZATION

There are two constructors, including a default constructor. The parametric model is specified by arguments of type `Trend` and `VarStructure`, and the simulation domain is defined by the `PointSet` argument. The integer argument specifies the seed for the random number generator. If no non-zero value is specified, the seed will be taken to be the point of time.

MEMBER FUNCTIONS

`reset` - resets the object. See the documentation of class `ScalarSpatSim`.

`setup` - initializes the `Trend` and `VarStructure` members, and computes the eigenvalue decomposition of the covariance matrix.

`simulate` - performes one simulation of the random field on the specified point set and according to the model parameters. The resulting vector of simulated values, ordered according to the numbering of the `nps` points of the point set, is returned as a `nps` by one matrix by the `nMatrix` reference argument.

FILES

EVSpatSim.C

EXAMPLE

See class ScalarSpatSim

SEEALSO

class ChSpatSim, class ChSemiSpatSim, class PointSet, class ScalarSpatSim, class Trend, class VarStructure

AUTHOR

Turid Follestad, NR

6.2.5 CondScalarSpatSim

NAME

CondScalarSpatSim - an abstract base class for methods for conditional simulation of one-dimensional gaussian stochastic fields

INCLUDE

```
include "CondScalarSpatSim.h"
```

SYNTAX

```
class CondScalarSpatSim : public virtual ScalarSpatSim
{
 protected:
    Handle(SpatialData) data;   // data observations

  public:
    CondScalarSpatSim(const SpatialData& sd, const PointSet& pset, int seed=0)
      :ScalarSpatSim(pset, seed)
        { data.rebind(sd.createCopy());}
    CondScalarSpatSim() {;}
    ~CondScalarSpatSim() {;}
  };
```

KEYWORDS

conditional simulation, simulation, spatial model, stochastic field

DESCRIPTION

This class is an abstract base class for methods for simulation of one-dimensional gaussian stochastic fields, conditionally on data. The class is derived from class `ScalarSpatSim`. In addition to the simulation domain, contained in `ScalarSpatSim`, this class requires a spesification of the given data. This is done by specifying the `SpatialData` argument of the constructor.

CONSTRUCTORS AND INITIALIZATION

The class has two constructors, including a default constructor. The simulation domain is specified by an object in the `PointSet` hierarcy, and the `SpatialData` argument specifies the locations and values of the given data. The integer argument specifies the seed for the random number generator. If no non-zero value is specified, the seed will be taken to be the point of time.

MEMBER FUNCTIONS

None but those inherited from the base class `ScalarSpatSim`.

FILES

None.

EXAMPLE

```
/*************************************************************************/
/*  Conditional simulation of a one-dimensional field on a grid lattice,    */
/*                                                                          */
/*  An input file must be supported, with format (example)                  */
/*   -spatial dimension (2)                                                 */
/*   -name of file containing the data: the first d columns should contain  */
/*           the d coordinates, and the last column the value.              */
/*   -grid spec. (grid: d=2 domain = [0,1]x[0,1] grid points = [1:5]x[1:5]) */
/*   -covariance model (bessel)                                             */
/*   -covariance parameters - nugget, partial sill, range (0.0 1.0 0.3)     */
/*   -if 'bessel': order of modified Bessel function (2)                    */
/*   -order of polynomial trend (0)                                         */
/*   -trend coefficients (0.0)                                              */
/*   -number of simulations, seed and indicator for non-smooth (0) or       */
/*         smooth field (1)   (500 326 0)                                   */
/*************************************************************************/
#include <CondChSemiSpatSim.h>
#include <CondChSpatSim.h>
#include <PolTrend.h>

main(int argc, char* argv[])
{
  char* filename = argv[1];
  ifstream inputfile(filename,ios::in);

  // ******************* INITIALIZATION/SPECIFICATION ************

  // read data:
  int d;                              // spatial dimension
  inputfile >> d;

  char text[15];
  inputfile >> text;                  // name of file holding the data

  int fn = countNumbers("data.sim"); // counts number of elements on file
  int np = fn/(d+1);                  // number of data observations

  IrregSpatialData data(np,d);        // object for irregularly spaced data
  ifstream infile(text,ios::in);
  data.scan(infile);                  // reads data from file
  infile.close();

  // read point set:

  RegGrid pset(d);                    // simulation domain: a grid lattice
  pset.scan(inputfile);

  // structure of variation:

  char vmod[15];
  inputfile >> vmod;                  // covariance model
  VarStructure vstr(vmod);            // object for structure of variation
  nVector prmvec(vstr.getNparam());  // covariance parameters
  inputfile >> prmvec;
  vstr.setParam(prmvec);

  // trend model:

  int po;                             // polynomial order
```

```
  inputfile >> po;
  PolTrend ptrend(po,d);             // polynomial trend model
  nVector tval(ptrend.getOrder());   // trend coefficients
  inputfile >> tval;
  ptrend.setCoef(tval);

  // number of simulations, seed and specification of smooth/non-smooth field:

  int nsim, seed, choice;
  inputfile >> nsim;
  inputfile >> seed;
  inputfile >> choice; // if (choice = 0) the field is assumed to be non-smooth
                       // if (choice = 1) the field is assumed to be smooth

  // **** END OF INITIALIZATION/SPECIFICATION ************

  // simulator:

  CondScalarSpatSim* sim=NULL;       // simulation object

  if (choice==0) // non-smooth, use Cholesky decomposition method
    sim = new CondChSpatSim(vstr,ptrend,pset,data,seed);
  else           // smooth, use modified Cholesky decomposition method
    sim = new CondChSemiSpatSim(vstr,ptrend,pset,data,seed);

  nMatrix simmatr(pset.getNpoints(),nsim);

  // simulation loop:

  nMatrix res(pset.getNpoints(),1);

  for (int isim=1;isim<=nsim;isim++){
    sim->simulate(res);
    simmatr.setCol(isim,res.col(1));
  }

  // prints coordinate matrix and simulated values to a file:
  ofstream ostr("sim.res",ios::out);
  pset.coordMatrix().colConc(simmatr).print(ostr);
  ostr.close();

  delete sim;                        // deletes CondScalarSpatSim object
}
```

SEEALSO

class CondChBaseSpatSim, class PointSet, class ScalarSpatSim, class SpatialData

AUTHOR

Turid Follestad and Tove Andersen, NR

6.2.6 CondChBaseSpatSim

NAME

CondChBaseSpatSim - an abstract base class for conditional simulation of one-dimensional gaussian stochastic fields using Cholesky decomposition

INCLUDE

```
include "CondChBaseSpatSim.h"
```

SYNTAX

```
class CondChBaseSpatSim : public virtual CondScalarSpatSim
{
 protected:

  VarStructure varstr;       // structure of variation
  TriangMatrix covTrendL;    // Cholesky factor of cov.matrix for trend coef.
  Handle(Trend) trDistrib;   // trend model
  TriangMatrix ccL;          // Cholesky factor of data covariance matrix
  nIntVector sindex, pindex; // scratch vectors
  BooLean simtrend;          // indicator of trend simulation on/off

  nVector covVector(const nVector& x); // covariance vector
  void duplicates(PointSet& ps, const SpatialData& data);
       // check for duplicates between simulation domain and data locations
  virtual nVector simTrendCoef ();     // simulation of trend coefficients
  virtual nVector simConst (const nMatrix& zsim)=0; // constant under simulation
  virtual void uncondSim(nMatrix&)=0;  // unconditional simulation

 public:

  CondChBaseSpatSim(const VarStructure&, const Trend&, const SymMatrix& trVar,
                    const PointSet&, const SpatialData&, int seed=0);
  CondChBaseSpatSim(const VarStructure&, const Trend&,
                    const PointSet&, const SpatialData&, int seed=0);
  CondChBaseSpatSim();

  virtual void simulate (nMatrix&);
  virtual void simTrendCoefOn (BooLean btrend, const SymMatrix& trVar);
        // sets trend coefficient simulation status
};
```

KEYWORDS

Cholesky decomposition method, conditional simulation, spatial model, stochastic field

DESCRIPTION

The class is derived from class `CondScalarSpatSim`, and is an abstract base class for conditional simulation of one-dimensional gaussian stochastic fields, based on Cholesky decomposition of the covariance matrix. There are two derived classes: `CondChSpatSim` and `CondChSemiSpatSim`, similar to `ChSpatSim` and `ChSemiSpatSim` for unconditional simulation of non-smooth and very smooth fields.

The method is based on unconditional simulation and kriging, and the conditionally simulated field is computed by

```
Zcond(x) = Zncond(x) + k'(x)*inv(Sigma)*(Z0 - Z0sim),
```

where `Zncond` is the unconditionally simulated field, `Sigma` is the data covariance matrix, `Z0` is the vector of data observations, and `Z0sim` the (unconditionally) simulated values in the data locations. See Cressie (pp.207-209) for a derivation of the method.

The member `Trend` is referenced through the use of smart pointers, i.e. handles, for simplified memory management. A handle consists of a pointer to the object and some interface functions.

REFERENCE:

Cressie, N.: "Spatial Statistics", Wiley, 1991.

CONSTRUCTORS AND INITIALIZATION

The class has two non-default constructors. The parametric model is specified by arguments of type `VarStructure` and `Trend`. If the trend coefficients are to be simulated, a `SymMatrix` argument, holding the covariance matrix for the trend coefficients, should be specified. The member function `simTrendCoefOn` can be called to set trend coefficient estimation on (`btrend` = dpTRUE) or off (`btrend` = dpFALSE). By default, the trend coefficients are not simulated.

The simulation domain is determined by the `PointSet` argument, and the given data by the `SpatialData` argument. The integer argument specifies the seed for the random number generator. If no non-zero value is specified, the seed will be taken to be the point of time.

MEMBER FUNCTIONS

`simTrendCoefOn` - This function can be called to set trend coefficient estimation on (`btrend` = dpTRUE) or off (`btrend` = dpFALSE), an set the covariance matrix for the trend coefficients to the value of the `SymMatrix` reference argument.

`simulate` - simulates one realisation of the field according to the specified model parameters, conditionally on the data. The resulting vector of simulated values, ordered according to the numbering of the `nps` points of the point set, is returned as a `nps` by one matrix by the `nMatrix` reference argument.

FILES

CondChBaseSpatSim.C

EXAMPLE

See class CondScalarSpatSim

SEEALSO

class CondChSpatSim, class CondChSemiSpatSim, class CondScalarSpatSim

AUTHOR

Turid Follestad and Tove Andersen, NR

6.2.7 CondChSpatSim

NAME

CondChSpatSim - a class for conditional linear simulation of a one-dimensional spatial gaussian field, by the Cholesky decomposition method

INCLUDE

```
include "CondChSpatSim.h"
```

SYNTAX

```
class CondChSpatSim: public CondChBaseSpatSim, public ChSpatSim
{
 private:
  void init (const VarStructure&, const Trend&,
             const PointSet&, const SpatialData&, int seed=0);

 protected:
  nVector simConst (const nMatrix& zsim); // constant under simulation

  void uncondSim(nMatrix& matr)           // unconditional simulation
    {ChSpatSim::simulate(matr);}

 public:
  CondChSpatSim (const VarStructure&, const Trend&,
                 const SymMatrix& trVar, const PointSet&,
                 const SpatialData&, int seed=0);
  CondChSpatSim (const VarStructure&, const Trend&,
                 const PointSet&,
                 const SpatialData&, int seed=0);
  CondChSpatSim();
  ~CondChSpatSim();

  void simulate (nMatrix& matr)
    {CondChBaseSpatSim::simulate(matr);}

  void simTrendCoefOn (BooLean btrend, const SymMatrix& trVar)
    {CondChBaseSpatSim::simTrendCoefOn(btrend, trVar);}
};
```

KEYWORDS

Cholesky decomposition method, conditional simulation, gaussian field, spatial model, stochastic field

DESCRIPTION

The class is derived from the classes `CondChBaseSpatSim` and `ChSpatSim`, and is an implementation of the Cholesky decomposition method. For very smooth fields, this method might fail. In such cases, method `CondChSemiSpatSim` could be used.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors, including a default constructor. For a specification of the constructor arguments, see the documentation of class `CondChBaseSpatSim`.

MEMBER FUNCTIONS

See documentation of the classes `CondChBaseSpatSim` and `ChSpatSim`.

FILES

CondChSpatSim.C

EXAMPLE

See class CondScalarSpatSim

SEEALSO

class CondChBaseSpatSim, class CondChSemiSpatSim, class ChSpatSim

AUTHOR

Turid Follestad and Tove Andersen, NR

6.2.8 CondChSemiSpatSim

NAME

CondChSemiSpatSim - a class for conditional linear simulation of a one- dimensional spatial gaussian field, by a modified Cholesky decomposition method

INCLUDE

```
include "CondChSemiSpatSim.h"
```

SYNTAX

```
class CondChSemiSpatSim: public CondChBaseSpatSim, public ChSemiSpatSim
{
 private:
  void init (const VarStructure&, const Trend&,
             const PointSet&, const SpatialData&, int seed=0);

 protected:
  nIntVector pivcL; // pivot vector for data covariance matrix

  nVector simConst (const nMatrix& zsim); // constant under simulation

  void uncondSim(nMatrix& matr)           // unconditional simulation
    {ChSemiSpatSim::simulate(matr);}

 public:
  CondChSemiSpatSim (const VarStructure&, const Trend&,
                     const SymMatrix& trVar, const PointSet&,
                     const SpatialData&, int seed=0);
  CondChSemiSpatSim (const VarStructure&, const Trend&,
                     const PointSet&,
                     const SpatialData&, int seed=0);
  CondChSemiSpatSim();
  ~CondChSemiSpatSim();

  void simulate (nMatrix& matr)
    {CondChBaseSpatSim::simulate(matr);}

  void simTrendCoefOn (BooLean btrend, const SymMatrix& trVar)
    {CondChBaseSpatSim::simTrendCoefOn (btrend, trVar);}
};
```

KEYWORDS

Cholesky decomposition method, conditional simulation, gaussian field, spatial model, stochastic field, symmetric pivoting

DESCRIPTION

The class is derived from the classes `CondChBaseSpatSim` and `ChSemiSpatSim`. It implements conditional simulation of a gaussian scalar field, based on a Cholesky like decomposition of the covariance matrix, using symmetric pivoting. This method will work for very smooth fields, for which the ordinary Cholesky decomposition method (class `CondChSpatSim`) fails.

The Cholesky decomposition is performed by the function `cholFacorSemi`, of the matrix library in NUTILITY.

CONSTRUCTORS AND INITIALIZATION

The class has three constructors, including a default constructor. For a specification of the constructor arguments, see the documentation of class `CondChBaseSpatSim`.

MEMBER FUNCTIONS

See documentation of the classes `CondChBaseSpatSim` and `ChSemiSpatSim`.

FILES

CondChSemiSpatSim.C

EXAMPLE

See class CondScalarSpatSim

SEEALSO

class CondChBaseSpatSim, class ChSemiSpatSim

AUTHOR

Turid Follestad and Tove Andersen, NR

6.3 Simulation of vector fields

6.3.1 VecSpatSim

NAME

VecSpatSim - an abstract base class for spatial simulation of multidimensional gaussian fields

INCLUDE

```
include "VecSpatSim.h"
```

SYNTAX

```
class VecSpatSim: public SpatialSim
{
protected:
  Handle(PointSet) points;  // simulation domain
public:
  VecSpatSim(const PointSet& pset, int seed = 0)
    :SpatialSim(seed)
      { points.rebind(pset.createCopy()); }
  VecSpatSim()
    :SpatialSim() { ;}

  virtual void reset(const PointSet& ps)
    { points.rebind(ps.createCopy()); }
};
```

KEYWORDS

stochastic field, simulation, gaussian field

DESCRIPTION

The class is an abstract base class for simulation of a multidimensional gaussian stochastic field. The field is simulated in a general set of points or a grid lattice, initialized by the `PointSet` reference constructor argument. The member is referenced through the use of smart pointers, i.e. handles, for simplified memory management. A handle consists of a pointer to the object and some interface functions.

CONSTRUCTORS AND INITIALIZATION

There are two constructors, including a default constructor not allocating any memory. The simulation domain is determined by the `PointSet` argument. The integer argument specifies the seed for the random number generator. If no non-zero value is specified, the seed will be taken to be the point of time.

MEMBER FUNCTIONS

None.

FILES

None.

EXAMPLE

See class ChVecSpatSim, class CondChVecSpatSim, class NonDivVecSim2d

SEEALSO

class PointSet, class SpatialSim

AUTHOR

Turid Follestad, NR

6.3.2 ChVecSpatSim

NAME

ChVecSpatSim - a class for linear spatial simulation of gaussian vector fields by the Cholesky decomposition method

INCLUDE

```
include "ChVecSpatSim.h"
```

SYNTAX

```
class ChVecSpatSim: public VecSpatSim {
 protected:
  int nvec;             // dimensionality of the vector field
  BooLean semid;        // indicator of smooth/non-smooth field
  nIntVector pivot;     // pivot vector of modified Cholesky decomposition
  TriangMatrix covmatL; // (modified) Cholesky factor of covariance matrix
  nMatrix trendval;     // trend function values in the simulation domain

  virtual void setup(const VecSimplest(TP)&, const VecFieldVar&,
                     BooLean sd=dpFALSE);
  void reset(const PointSet& ps)
    { VecSpatSim::reset(ps); }

 public:
  ChVecSpatSim(const VecSimplest(TP)&, const VecFieldVar&,
               const PointSet&, int seed=0, BooLean sd=dpFALSE);
  ChVecSpatSim(const VecSimplest(TP)&, const TriangMatrix& CovL,
               const PointSet&, int seed=0, BooLean sd=dpFALSE,
                  const nIntVector* piv=NULL);
  ChVecSpatSim();
  ~ChVecSpatSim() {;}

  // reset model parameters:
  void reset(const VecSimplest(TP)&, const VecFieldVar&,
             const PointSet&, int seed=0, BooLean sd=dpFALSE);
  void reset(const VecSimplest(TP)&, const TriangMatrix& CovL,
             const PointSet&, int seed=0, BooLean sd=dpFALSE,
             const nIntVector* piv=NULL);

  TriangMatrix getCovL() const { return covmatL; } // Cholesky factor
  nIntVector getPivot() const { return pivot; }    // Cholesky factor pivot vector

  void simulate (nMatrix&);
  void divergence(const nMatrix& simr, nVector& div); // divergence of vector field
};
```

KEYWORDS

Cholesky decomposition method, gaussian field, simulation, stochastic field

DESCRIPTION

The class represents simulation of a gaussian stochastic vector field by the Cholesky decomposition method, and is derived from class `VecSpatSim`. The simulation is based on

parametric models for the trend and structure of variation of the stochastic field. Their parameters should be set by the user. The field is simulated in a general set of points or a grid lattice.

The simulation method is a generalization of the Cholesky decomposition method for one-dimensional fields. Let

```
Z = (U,V)'
```

be a length `2n` vector, where `U` and `V` are the two length `n` components of a two-dimensional vector field. The simulation is based on a Cholesky decomposition of the composite covariance matrix

```
                cov(U,U) cov(U,V)
Sigma_Z =
                cov(V,U) cov(V,V)
```

The simulated field is computed by

```
Zsim(x) = mu(x) + L*w(x),
```

where

```
w(x) ~ N   (0,Sigma_Z),
        2n
```

`L` is the Cholesky factor of `Sigma_Z` and

```
mu(x) = (mu (x), (mu (x))'.
           U        V
```

In the current version, the classes in the `VecFieldVar`-hierarchy, defining the covariance structure of the vector field, is only implemented for two-dimensional vector fields. The covariance models included, represented by the classes in the `VecCovFunc`-hierarchy, also assumes a non-divergent vector field. Other models might be implemented by the user, by deriving new classes from the base class `VecCovFunc`.

CONSTRUCTORS AND INITIALIZATION

There are three constructors, including a default constructor. The parametric model is specified by constant reference arguments of type `VecSimplest(TP)`, where TP is a pointer to class `Trend`, and `VecFieldVar`. Alternatively, if the covariance matrix is computed on beforehand, its Cholesky factor, and the corresponding pivot vector for a modified Cholesky decomposition, can be specified. The simulation domain is determined by the `PointSet` argument, and the integer argument specifies the seed for the random number generator. If no non-zero value is specified, the seed will be taken to be the point of time. If the simulated field is assumed to be very smooth, the parameter `sd` should be set to dpTRUE.

MEMBER FUNCTIONS

`reset` - resets the vector of trend pointers, the structure of variation and the simulation domain. An alternative specification of the structure of variation is the Cholesky factor of the covariance matrix, and the corresponding pivot vector for a modified Cholesky decomposition. A non-zero value of the seed can be specified by the integer argument. By default, the point of time is taken as the seed value. If the simulated field is assumed to be very smooth, the parameter `sd` should be set to dpTRUE. The function is intended for use when as object is created using the default constructor. If another constructor has been used, a specification of a seed for the random number generator will be ignored, and the random generator attached to the global random stream already created.

`simulate` - performes one simulation of the random `nv`-dimensional vector field in the specified point set and according to the model parameters. The resulting matrix of simulated values, ordered according to the numbering of the `nps` points of the point set, is returned as a `nps` by `nv` matrix by the `nMatrix` reference argument.

`divergence` - computes the estimated divergence for the simulated vector field. The implementation restricts the simulation domain to be a grid lattice in two dimensions. The arguments are a matrix of simulated values and a vector reference argument through which the resulting divergence in each of the grid points is returned.

FILES

ChVecSpatSim.C

EXAMPLE

```
#include <ChVecSpatSim.h>
#include <PolTrend.h>
#include <GenVecFieldVar.h>
#include <GaussVecCovFunc.h>

main(int argc, char* argv[])
{
  int nsim = atoi(argv[1]);
  int np = atoi(argv[2]);
  int dim = atoi(argv[3]);
  int veclen = atoi(argv[4]);            // should be 2
  int seed = atoi(argv[5]);

  // simulation domain:

  ifstream infile("gen.points",ios::in);
  GenPointSet pset(np,dim);
  pset.scan(infile);
  infile.close();

  // structure of variation:

  double range = 0.2, variance = 1.0;
  GaussVecCovFunc cfunc(range,variance); // "gaussian" type covariance function

  GenVecFieldVar var(cfunc);             // covariance structure

  // trend model:

  VecSimplest(TP) trend(veclen);

  PolTrend ptr1(0,dim);                  // first component: constant trend
```

```
  nVector tcoef(ptr1.getOrder(),0.0);
  ptr1.setCoef(tcoef);

  PolTrend ptr2(1,dim);                 // second component: first order
                                        // polynomial function
  tcoef.redim(ptr2.getOrder());
  for (int i=1;i<=ptr2.getOrder();i++)
    tcoef(i) = 0.0;
  ptr2.setCoef(tcoef);

  trend(1) = ptr1.createCopy();
  trend(2) = ptr2.createCopy();

  // creating simulation object, assuming a non-smooth field:

  ChVecSpatSim sim(trend,var,pset,seed);

  nMatrix res(pset.getNpoints(),veclen);
  ofstream outfile("simres",ios::out);

  // simulation:

  for (i=1;i<=nsim;i++){
    sim.simulate(res);  // simulation of one realisation

    outfile << "Simulation no. " << i << ":\n";
    outfile << "----------------------------\n";
    res.print(outfile);
    outfile << "----------------------------\n";
  }
  outfile.close();
}
```

SEEALSO

class GenVecFieldVar, class PointSet, class Trend, class VecFieldVar, class VecSpatSim

AUTHOR

Turid Follestad, NR

6.3.3 NonDivVecSim2d

NAME

NonDivVecSim2d - a class for spatial simulation of a two-dimensional non-divergent gaussian vector field

INCLUDE

```
include "NonDivVecSim2d.h"
```

SYNTAX

```
class NonDivVecSim2d: public VecSpatSim {
 protected:
  ScalarSpatSim* scalar;   // object for simulation of a scalar field
  RegGrid grid;            // simulation domain

  void reset(const PointSet& ps)
    { VecSpatSim::reset(ps); }

 public:
  NonDivVecSim2d(const Trend& tr, const VarStructure& vs,
                 const RegGrid& rg, int seed=0, BooLean sd=dpFALSE);
  NonDivVecSim2d()
    : grid(), VecSpatSim() { scalar = NULL;}

  ~NonDivVecSim2d() { if (scalar!=NULL) delete scalar;}

  void reset(const Trend& tr, const VarStructure& vs,
             const RegGrid& rg, int seed=0, BooLean sd=dpFALSE);

  void simulate (nMatrix&);
};
```

KEYWORDS

gaussian field, stochastic field, simulation, non-divergent field, velocity field

DESCRIPTION

The class represents simulation of a non-divergent two-dimensional gaussian vector field, as defined in Høst (1994). The simulation is based on parametric models for the trend and structure of variation of a scalar field, and the vector field is obtained by numerical differentiation of the simulated scalar field.

Let `Y` be the simulated scalar field, simulated by one of the methods in the `ScalarSpatSim`-hierarchy. The `U`- and `V`-components of the vector field are then computed by

```
U(x_i ,x_j) = - (Y(x_i,x_(j+1)) - Y(x_i,x_j))/delta(x2)
```

and

```
V(x_i ,x_j) = (Y(x_(i+1),x_j) - Y(x_i,x_j))/delta(x1).
```

The field is simulated on a grid lattice.

REFERENCE:

Høst, G.: "A Note on Two-Dimensional Non-Divergent Gaussian Random Vector Fields", NR-notat, STAT/05/94.

CONSTRUCTORS AND INITIALIZATION

There are two constructors, including a default constructor not allocating any memory. The parametric model of the corresponding scalar field is specified by constant reference arguments of type `Trend` and `VarStructure`. The simulation domain is indicated by the `RegGrid` argument, and the integer argument specifies the seed for the random number generator. If no non-zero value is specified, the seed will be taken to be the point of time. If the simulated scalar field is assumed to be very smooth, the parameter `sd` should be set to dpTRUE. In this case, the scalar field is simulated by the method of class `ChSemiSpatSim`. By default, the method of class `ChSpatSim` is used.

MEMBER FUNCTIONS

`reset` - function that resets the trend and the structure of variation of the scalar field, as well as the simulation domain. A non-zero value of the seed can be specified by the integer argument. By default, the point of time is taken as the seed value. The function is intended for use when as object is created using the default constructor. If another constructor has been used, a specification of a seed for the random number generator will be ignored, and the random generator attached to the global random stream already created.

`simulate` - performes one simulation of the random two-dimensional non-divergent vector field on the specified grid and according to the model parameters for the scalar field. The resulting matrix of simulated values, ordered according to the numbering of the `nps` points of the grid, is returned as a `nps` by `2` matrix by the `nMatrix` reference argument.

FILES

NonDivVecSim2d.C

EXAMPLE

```
#include <NonDivVecSim2d.h>
#include <PolTrend.h>

main(int argc, char* argv[])
{
  int d = 2;
  char* vmod = argv[1];     // model for structure of variation
  int nsim = atoi(argv[2]); // number of simulations
  int seed = atoi(argv[3]); // seed for random generator

  // read point set (a grid lattice):

  RegGrid pset(d);
  ifstream infile("grid.spec",ios::in);
```

```
  pset.scan(infile);
  infile.close();

  // structure of variation:

  VarStructure vstr(vmod);
  nVector prmvec(vstr.getNparam());
  infile.open("init.par",ios::in);
  prmvec.scan(infile);
  vstr.setParam(prmvec);
  infile.close();

  // trend model:

  PolTrend ptrend(0,d);
  nVector tval(1);
  tval(1) = 0.0;
  ptrend.setCoef(tval);

  // creating object for simulation:

  NonDivVecSim2d sim(ptrend,vstr,pset,seed);

  nMatrix simmatr(pset.getNpoints(),2*nsim);
  nMatrix res(pset.getNpoints(),2);

  for (int isim=1;isim<=nsim;isim++){
    cout << "simulering " << isim << endl;
    sim.simulate(res);

    simmatr.setCol(2*isim-1,res.col(1));
    simmatr.setCol(2*isim,res.col(2));
  }

  ofstream ostr("sim.res",ios::out);
  pset.coordMatrix().colConc(simmatr).print(ostr);
        // printing coordinates and simulated values for all realisations
  ostr.close();
}
```

SEEALSO

class RegGrid, class VecSpatSim, class Trend, class VarStructure

AUTHOR

Turid Follestad, NR

6.3.4 CondChVecSpatSim

NAME

CondChVecSpatSim - a class for conditional spatial simulation of gaussian vector fields

INCLUDE

```
include "CondChVecSpatSim.h"
```

SYNTAX

```
class CondChVecSpatSim : public ChVecSpatSim
{
 private:
  Handle(SpatialData) sdata;    // data observations
  TriangMatrix trVarL;          // Cholesky factor for covariance matrix
                                // for trend coefficients
  VecSimplest(TP) trDistrib;    // trend model
  Handle(VecFieldVar) vstr;     // structure of variation
  VecSimple(nIntVector) mind;   // scratch vector

  nMatrix Cvec;                 // each row is a covariance vector
  TriangMatrix ccL;             // Cholesky factor of data covariance matrix
  nIntVector pivL;              // pivot vector for Cholesky factor
  VecSimple(nIntVector) sindex;
  nIntVector pindex;            // scratch vectors
  BooLean simtrend;             // indicator of trend simulation on/off

  void init (const VecFieldVar&, const VecSimplest(TP)&,
             PointSet&, const SpatialData&,
             int seed=0, BooLean sd=dpFALSE);

 protected:
  nVector covWithData (const nVector& x, int vind); // covariance vector
  nMatrix covWithDataM ();      // covariance vectors, stored rowwise
  nVector simConst (const nMatrix& zsim);           // simulation constant
  nVector simTrendCoef ();      // simulation of trend coefficients

  // check for duplicates between simulation domain and data locations:
  void duplicates (PointSet& ps, const SpatialData& data);

  SymMatrix covarCond () const; // data covariance matrix

  void setup(const VecSimplest(TP)& tp, const VecFieldVar& v,
             BooLean sd=dpFALSE)
    { ChVecSpatSim::setup(tp,v,sd); }

 public:
  CondChVecSpatSim (const VecFieldVar&, const VecSimplest(TP)&,
                    const SymMatrix& trVar, PointSet&,
                    const SpatialData&, int seed=0, BooLean sd=dpFALSE);
  CondChVecSpatSim (const VecFieldVar&, const VecSimplest(TP)&,
                    PointSet&, const SpatialData&,
                    int seed=0, BooLean sd=dpFALSE);

  void simulate (nMatrix& Y);

  // set trend coefficient simulation status:
  void simTrendCoefOn (BooLean btrend, const SymMatrix& trVar);

  // get indicator of whether trend coefficients are set to be simulated:
```

```
  BooLean trendCoefSimulated () const { return simtrend; }
};
```

KEYWORDS

Cholesky decomposition method, conditional simulation, gaussian field, spatial model, stochastic field

DESCRIPTION

The class is derived from the class `ChVecSpatSim`, and is an implementation of conditional simulation of a gaussian vector field by the Cholesky decompsition method.

The method is based on unconditional simulation and kriging, and is a generalization of the method used for one-dimensional fields in class `CondChBaseSpatSim`. For a two-dimensional field `Z`,

```
Z = (U,V)',
```

the conditionally simulated field is computed by

$$\texttt{Ucond(x)} = \texttt{Uncond(x)} + \texttt{k'(x)*inv(Sigma)*(Z0 - Z0sim)},_{\texttt{U}}$$

and

$$\texttt{Vcond(x)} = \texttt{Vncond(x)} + \texttt{k'(x)*inv(Sigma)*(Z0 - Z0sim)}.$$

Here, `Uncond` and `Vncond` are the components of the unconditionally simulated field, `Sigma` is the covariance matrix, `Z0` is the vector of observed data, and `Z0sim` the (unconditionally) simulated values in the data locations. The vectors

$$\texttt{k (x) and k (x)}$$

are the vectors of covariances between the `U`- (`V`-) component of the field in the location `x`, and the `U`- and `V`-components of the data.

REFERENCE:

Cressie, N.: "Spatial Statistics", Wiley, 1991, pp. 207-209.

CONSTRUCTORS AND INITIALIZATION

The class has two constructors. The parametric model is specified by constant reference arguments of type `VecSimplest(TP)`, where TP is a pointer to class `Trend`, and `VecFieldVar`. The simulation domain is determined by the `PointSet` argument, and the integer argument specifies the seed for the random number generator. If no non-zero value is specified, the seed will be taken to be the point of time. If the simulated field is assumed to be very smooth, the parameter `sd` should be set to dpTRUE.

If the trend coefficient is to be simulated, a **SymMatrix** argument, holding the covariance matrix for the trend coefficients, should be specified. The member function **simTrendCoefOn** can be called to set trend coefficient estimation on (**btrend** = dpTRUE) or off (**btrend** = dpFALSE). By default, the trend coefficients are not simulated.

The field is simulated conditionally on the data specified by the **SpatialData** reference argument. If there are conditions on some, but not all, components of the vector field, the value of the unconstrained components should be set to -9999 in the **SpatialData** object. For example, if a two-dimensional field is to be simulated conditionally on a vertical component equal zero at **x2=b**, the response value matrix, **Y**, here shown with the corresponding coordinate matrix, **X**, should be defined as

```
    X                Y
x11  b           -9999 0
x12  b           -9999 0
 .   .             .   .
 .   .             .   .
 .   .             .   .
x1n  b           -9999 0
```

MEMBER FUNCTIONS

simTrendCoefOn - This function can be called to set trend coefficient estimation on (**btrend** = dpTRUE) or off (**btrend** = dpFALSE), with covariance matrix for the trend coefficients as specified by the **SymMatrix** reference argument.

simulate - performes one simulation of the random **nv**-dimensional vector field on the specified point set, according to the model parameters and conditionally on the data. The resulting matrix of simulated values, ordered according to the numbering of the **nps** points of the point set, is returned as a **nps** by **nv** matrix by the **nMatrix** reference argument.

FILES

CondChVecSpatSim.C

EXAMPLE

```
/*************************************************************************/
/*  Simulation of a two-dimensional vector field on a grid lattice,      */
/*  under the condition of zero vertical component at the upper and      */
/*  lower boundaries of the simulation domain.                           */
/*                                                                       */
/*  An input file must be supported, with format (example)               */
/*   -spatial dimension (2)                                              */
/*   -grid spec. (grid: d=2 domain = [0,1]x[0,1] grid points = [1:5]x[1:5]) */
/*   -covariance model (bessel)                                          */
/*   -covariance parameters - partial sill, range (1.0 0.3)              */
/*   -if 'bessel': order of modified Bessel function (2)                 */
/*   -order of polynomial trend (0)                                      */
/*   -trend coefficients (0.0)                                           */
/*   -number of simulations and seed (500 326)                           */
/*   -non-smooth (0) or smooth field (1)  (0)                            */
/*************************************************************************/
```

```
#include <CondChVecSpatSim.h>
#include <GenVecFieldVar.h>
#include <PolTrend.h>

// declaration of function to compute data set from condititions:
IrregSpatialData condDataVert0(RegGrid&);

main(int argc, char* argv[])
{
  // reads name of specification file:
  char* filename = argv[1];
  ifstream inputfile(filename,ios::in);

  // ******************* INITIALIZATION/SPECIFICATION ************
  const int nvec = 2;
  int d, ival;
  inputfile >> d; // spatial dimension

  // reads specification of the simulation domain, a grid lattice:
  RegGrid pset;
  pset.scan(inputfile);

  // computes data from conditions. The vertical component of the vector field
  // is to be zero at x2=smallest value and x2=largest value
  IrregSpatialData tdata = condDataVert0(pset);

  // definition of structure of variation:

  char vmod[15];
  inputfile >> vmod;
  nVector prmvec(2);
  inputfile >> prmvec;
  if (strcmp(vmod,"bessel")==0){  // if modified Bessel function
    inputfile >> ival;            // covariance model:
    prmvec = prmvec.concat(ival); // reads order of Bessel function
  }
  GenVecFieldVar vstr(vmod);      // covariance structure of vector field
  vstr.setParam(prmvec);          // sets model parameters

  // trend model: polynomial trend of order  ival  for both components
  //              of the vector field

  inputfile >> ival;
  PolTrend ptrend(ival,d);        // polynomial trend model object
  nVector tval(ptrend.getOrder());
  tval.scan(inputfile);           // reads model parameters
  ptrend.setCoef(tval);           // sets model parameters

  VecSimplest(TP) trvec(d);
  for (int i=1;i<=d;i++)          // sets trend model for each component
    trvec(i) = new PolTrend(ptrend);                // equal to  ptrend

  // number of simulations and seed:
  int nsim, seed;
  inputfile >> nsim >> seed;

  // smooth or non-smooth field:
  int sdval;
  inputfile >> sdval;
  if ((sdval!=0)&&(sdval!=1))
    fatalerrorFP("main","Indicator should be in {0,1}\n");
  BooLean sdvalb;
  if (sdval==1) sdvalb=dpTRUE;    // smooth field, modified Cholesky
                                  // decomposition is to be used
  if (sdval==0) sdvalb=dpFALSE;   // non-smooth field

  // **** END OF INITIALIZATION/SPECIFICATION ************
```

```
  // creates simulation object:
  CondChVecSpatSim sim(vstr,trvec,pset,tdata,seed,sdvalb);

  // simulation loop:

  nMatrix simmatr(pset.getNpoints(),2*nsim);
  nMatrix res(pset.getNpoints(),2);

  for (int isim=1;isim<=nsim;isim++){
    cout << "simulation no. " << isim << endl;
    sim.simulate(res);
    simmatr.setCol(2*isim-1,res.col(1));
    simmatr.setCol(2*isim,res.col(2));
  }

  ofstream ostr("sim.res",ios::out);

  nMatrix mm=pset.coordMatrix();
  mm=mm.colConc(simmatr);
  mm.print(ostr);
  ostr.close();
}

/*************************************************************************/
/*  Function to create an IrregSpatialData object specifying the data    */
/*  for the simulation of a two-dimensional vector field, under the      */
/*  condition of zero vertical component at the upper and lower          */
/*  boundaries of the simulation domain.                                 */
/*************************************************************************/

IrregSpatialData condDataVert0(RegGrid& rg)
{
  int nvec=2; // assumes the dimensionality of the vector field to be 2
  int nc=0, d=rg.getDim();
  int i;
  nc = 2*(rg.getNdiv(1)+1);

  nMatrix X(nc,d);
  nMatrix Y(nc, nvec, -9999.0);
  Y(1,2)=0.0;

  X.setRow(1,rg.firstPoint());
  int k=1;
  for (i=2;i<=rg.getNdiv(1)+1;i++){
    k++;
    X.setRow(k,rg.nextPoint());
    Y(k,2)=0.0;
  }

  double min1=rg.getMinval(1);
  double max1=rg.getMaxval(1);
  double delta1=rg.getDelta(1);
  double min2=rg.getMinval(2);
  double max2=rg.getMaxval(2);
  double delta2=rg.getDelta(2);

  for (i=1;i<=rg.getNdiv(1)+1;i++){
    k++;
    X(k,1) = min1+(i-1)*delta1;
    X(k,2) = max2;
    Y(k,2) = 0.0;
  }

  IrregSpatialData data(X,Y);
  return data;
}
```

SEEALSO

class ChVecSpatSim, class PointSet, class SpatialData, class Trend, class VecFieldVar

AUTHOR

Turid Follestad, NR

6.3.5 WeightSpatialSim

NAME

WeightSpatialSim - a class for simulation of gaussian stochastic vector fields

INCLUDE

```
include "WeightSpatialSim.h"
```

SYNTAX

```
class WeightSpatialSim : public SpatialSim
{
 public:
  WeightSpatialSim(const RegGrid& grid, int in_m, nMatrix Weights,
                   int seed=0);
  ~WeightSpatialSim() { ;}
  void simulate(nMatrix& Z);
};
```

KEYWORDS

stochastic field, simulation, weights

DESCRIPTION

`WeightSpatialSim` is a class for simulation of two-dimensional gaussian stochastic fields over a rectangular grid. A moving-average technique is used for the simulation.

The result `Z(i,j)` at each point `(i,j)` is a sum over independent gaussian (0,1) variables simulated for each gridpoint in a area around `(i,j)`, multiplicated with given weights. In the current version, the weights have to be set on beforehand, and are supplied through the constructor.

CONSTRUCTORS AND INITIALIZATION

The class has one constructor. It takes as input a specification of a grid lattice through a `RegGrid` reference argument, the size of the area around the point `(i,j)` which the summation is to be made over, a `nMatrix` holding the given weights, and a seed for the random number generator. If no non-zero value for the seed is specified, the seed will be taken to be the point of time.

In the grid specification, the parameters needed are the dimension (the current version is implemented for two-dimensional grids only) and the number of divisions in each dimension. The minimum and maximum values of the grid will not be used, since the stochastic process is assumed to be stationary.

MEMBER FUNCTIONS

`simulate` - simulates the stochastic process by summation over independent gaussian variables in the area around the point `(i,j)`, multiplicated by the given weights. The result is returned through the `nMatrix` reference argument, as a `m` by `n` matrix, `m` and `n` indicating the number of grid points in the two dimensions.

FILES

WeightSpatialSim.C

EXAMPLE

```
#include <WeightSpatialSim.h>

main()
{
  int d=2, in_m=1, rand;

  // weights:
  nMatrix W(2*in_m+1,2*in_m+1);
  W(1,1)=0.1;
  W(1,2)=0.1;
  W(1,3)=0.1;
  W(2,1)=0.1;
  W(2,2)=0.2;
  W(2,3)=0.1;
  W(3,1)=0.1;
  W(3,2)=0.1;
  W(3,3)=0.1;

  // specification of grid lattice:
  RegGrid grid(d);
  ifstream griddescr("grie.spec",ios::in);
  grid.scan(griddescr);
  griddescr.close();

  // seed:
  cout << "Give integer: " ;
  cin >> rand ;

  // simulation:
  WeightSpatialSim newfield(grid, in_m, W, rand);
  nMatrix res(grid.getNdiv(1)+1,grid.getNdiv(2)+1);
  newfield.simulate(res);

  // output of simulated values:
  ofstream outfile("simres",ios::out);
  res.print(outfile);
  outfile.close();
}
```

SEEALSO

class RegGrid, class SpatialSim

AUTHOR

Andre Teigland, NR. Modified by Turid Follestad, NR

Appendix A

Classes and functions for internal usage

A.1 Overview

In this appendix, some classes and functions for internal usage in the classes of the libraries, are documented. Most of these classes and functions are used to simplify the creation of objects in some of the class hierarchies. They are created from templates implemented in the module Diffpack by Hans Petter Langtangen, SINTEF/UiO. The templates are described briefly below.

Let **X** be the base class of a class hierarcy:

```
class X
{
  // ...
};
```

A class **prm(X)** is defined, holding the parameters needed to create an object of all subclasses in the **X**-hierarchy:

```
class prm(X)
{
  char* classname;     // name of subclass
  // ... parameters needed to create subclass
  void fill(...);      // set parameters
};
```

A pointer to an object of a class in the **X**-hierarchy can be initialized from these parameters by a function **createX**, declared as

```
X* createX (const prm(X)& pm).
```

A list of the names of all classes that can be created, is generated by the function

```
const char** hierX ().
```

A template for the declaration and definition of the **createX**- and **hierX**-functions for a class **X** can be automatically generated by the script "Mkcreate":

```
Mkcreate X.
```

Two files, "createX.h" and "createX.C" are created, and the user should edit the .C-file.

In the following sections, all parameter classes and create-functions in the NSPACE package are documented.

In section 3 of the appendix, modified Bessel functions used in one of the parametric semivariogram models, are documented, and the minimization procedure used in the weighted least squares estimation of the parameters of the semivariogram, is documented briefly in section A.4.

A.2 Creation of an instance of a derived class in a class hierarchy

A.2.1 createSpatialData

NAME

createSpatialData - create function for objects in the `SpatialData` hierarchy

INCLUDE

```
include "createSpatialData.h"
```

SYNTAX

```
extern const char** hierSpatialData (); // table of classes that can be created
extern SpatialData* createSpatialData (const prm(SpatialData)& pm);
```

SEEALSO

class SpatialData, class prm(SpatialData)

AUTHOR

Turid Follestad, NR, from a template written by Hans Petter Langtangen, SINTEF/UiO

A.2.2 prm(SpatialData)

NAME

prm(SpatialData) - parameters for the `SpatialData` hierarchy

INCLUDE

```
include "SpatialData.h"
```

SYNTAX

```
class prm(SpatialData){
 public:
  String classname;        // name of derived class

  int np;                  // number of points
  int dim;                 // spatial dimension
  int respdim;             // number of response variables

  prm(SpatialData)() :classname()
    { np = dim = respdim = 0;}

  void fill(const char* cn, int n, int d, int rd=1)
    { classname = cn; np = n; dim = d; respdim = rd;}

  void scan(Is istr)
    { istr >> classname; istr >> np; istr >> dim; istr >> respdim;}
  void print(Os ostr) const
    { ostr << "Classname:\t" << classname << '\n';
      ostr << "Number of points:\t" << np << '\n';
      ostr << "Spatial dimension:\t" << dim << '\n';
      ostr << "Number of response variables:\t" << respdim << '\n';
    }
};
```

KEYWORDS

initialization, parameters, spatial data

DESCRIPTION

The class contains the parameters needed to create an instance of an object of a class in the `SpatialData` hierarchy.

CONSTRUCTORS AND INITIALIZATION

The class has one constructor, a default constructor. By default, the parameters are all set to zero.

Further initialization is done by the `fill` - function, or could be done manually, since all members are public.

MEMBER FUNCTIONS

`fill` - function for initializing the data members. The members might as well be initialized manually, since all members are public. The arguments are a character string specifying the classname, the number of spatial points, `np`, the spatial dimension, `d`, and the number of response values, `rd`.

`scan` - reads the parameter values from an input stream. The parameter values are expected to be ordered as follows:

```
classname

number_of_points   spatial_dimension   number_of_resp._variables
```

`print` - prints classname and parameters to an output stream.

FILES

SpatialData.C

SEEALSO

class SpatialData, createSpatialData

AUTHOR

Turid Follestad, NR

A.2.3 createPointSet

NAME

createPointSet - create function for objects in the `PointSet` hierarchy

INCLUDE

```
include "createPointSet.h"
```

SYNTAX

```
extern const char** hierPointSet (); // table of classes that can be created
extern PointSet* createPointSet (const prm(PointSet)& pm);
```

SEEALSO

class PointSet, class prm(PointSet)

AUTHOR

Turid Follestad, NR, from a template written by Hans Petter Langtangen, SINTEF/UiO

A.2.4 prm(PointSet)

NAME

prm(PointSet) - parameters for the `PointSet` hierarchy

INCLUDE

```
include "PointSet.h"
```

SYNTAX

```
class prm(PointSet)
{
 public:
  char* classname;    // name of derived class

  int npoints;        // number of points
  int dim;            // spatial dimension
  nMatrix pdata;      // grid specification in case of "RegGrid",
                      // coordinate matrix in case of "GenPointSet"

  prm(PointSet)() : pdata()
    { classname = NULL; dim = 0; npoints = 0; }
  prm(PointSet)(char* cn, int d, int np, const nMatrix& data);

  void fill(char* cn, int dim, int np, const nMatrix& data);

  void scan(Is is);
  void print(Os os) const;
};
```

KEYWORDS

initialization, parameters, point set, spatial points, parameters

DESCRIPTION

The class contains the parameters needed to create an instance of an object of a class in the `PointSet` hierarchy.

CONSTRUCTORS AND INITIALIZATION

The class has two constructors, a default constructor and a constructor taking four arguments. The arguments are the classname, the spatial dimension, the number of points, and a matrix. The `nMatrix` argument is expected to hold the coordinate matrix if classname equals `GenPointSet`, and a set of parameters if classname equals `RegGrid`. In the case of a regular grid, the input parameter matrix is expected to be ordered as in the constructor argument for this class:

```
First column - number of divisions in each dimesion

Second column - minimum values of each dimension
```

```
Third column - maximum values of each dimension
```

MEMBER FUNCTIONS

`fill` - function for initializing the data members. The members might as well be initialized manually, since all members are public.

`scan` - reads parameter values from an input stream. The input values are expected to be ordered as follows:

```
classname

number of points

dimension

parameter ("RegGrid") or coordinate matrix ("GenPointSet")
```

`print` - prints classname and parameters to an output stream.

FILES

PointSet.C

SEEALSO

class GenPointSet, class PointSet, class RegGrid, createPointSet

AUTHOR

Turid Follestad, NR

A.2.5 createPrmSVModel

NAME

createPrmSVModel - create function for objects in the `PrmSVModel` hierarchy

INCLUDE

```
include "createPrmSVModel.h"
```

SYNTAX

```
extern const char** hierPrmSVModel (); // table of classes that can be created
extern PrmSVModel* createPrmSVModel (const prmSVModel& pm);
```

SEEALSO

class PrmSVModel

AUTHOR

Turid Follestad, NR, from a template written by Hans Petter Langtangen, SINTEF/UiO

A.2.6 prm(PrmSVModel)

NAME

prmSVModel - parameters for the `PrmSVModel` hierarchy

INCLUDE

```
include "PrmSVModel.h"
```

SYNTAX

```
class prmSVModel{
 public:
  char* classname;              // name of derived class
  double order;                 // order of modified Bessel function
                                // or general exponential function

  prmSVModel () { classname = NULL; order = 0.0;}
  void fill (char* name, double ord = 2.0)
    { classname = name; order = ord; }
};
```

KEYWORDS

covariance, initialization, parameters, semivariogram, variogram

DESCRIPTION

The class contains the parameters needed to create an instance of an object of a class in the `PrmSVModel` hierarchy, and is used by the function `createPrmSVModel`.

CONSTRUCTORS AND INITIALIZATION

The class has one constructor, a default constructor. Initialization is done by the `fill` - function, or could be done manually.

MEMBER FUNCTIONS

`fill` - initializes the data members. The member `order` is needed for only some of the derived classes in the `PrmSVModel` hierarchy.

FILES

PrmSVModel.C

SEEALSO

class PrmSVModel, createPrmSVModel

AUTHOR

Turid Follestad, NR

A.2.7 createVecFieldVar

NAME

createVecFieldVar - create function for objects in the `VecFieldVar` hierarchy

INCLUDE

```
include "createVecFieldVar.h"
```

SYNTAX

```
extern const char** hierVecFieldVar (); // table of classes that can be created
extern VecFieldVar* createVecFieldVar (const prm(VecFieldVar)& pm);
```

SEEALSO

class VecFieldVar, class prm(VecFieldVar)

AUTHOR

Turid Follestad, NR, from a template written by Hans Petter Langtangen, SINTEF/UiO

A.2.8 prm(VecFieldVar)

NAME

prm(VecFieldVar) - parameters for the `VecFieldVar` hierarchy

INCLUDE

```
include "VecFieldVar.h"
```

SYNTAX

```
class prm(VecFieldVar)
{
 public:
  char* classname;       // name of derived class
  int vlen;              // dimensionality of parameter vector
  char* genmodtype;      // name of subclass of VecCovFunc in case of a
                         // GenVecFieldVar object
  double range;          // practical range
  double variance;       // variance (nugget + partial sill)
  double nugget;         // nugget
  double beorder;        // order of modified Bessel function

  prm(VecFieldVar)()     // default constructor
    {
      classname = NULL;
      vlen = 0;
      genmodtype = NULL;
      range = 0;
      variance = 0;
      nugget = 0;
      beorder = 0;
    }

  void fill(char* cname, int vl, char* gmtype, double r=0, double v=0,
                  double n=0, double beo=0);
};
```

KEYWORDS

covariance, initialization, parameters, vector field

DESCRIPTION

The class contains all parameters needed to initialize an instance of a class in the `VecFieldVar` hierarchy.

CONSTRUCTORS AND INITIALIZATION

The class has one constructor, a default constructor initializing all members to zero.

MEMBER FUNCTIONS

`fill` - function for initialization of the data members. The members might as well be initialized manually, since all members are public. The function initializes the parameters of a `GenVecFieldVar` instance, and the arguments are the classname (`cn`), the dimensionality of the vector field (`vl`), the name of a subclass in the `VecCovFunc` hierarchy (`gmtype`), and the covariance parameters. These are the range (`r`), the variance (`v`) and the nugget effect (`n`).

FILES

VecFieldVar.C

SEEALSO

class VecFieldVar, createVecFieldVar

AUTHOR

Turid Follestad, NR

A.2.9 createVecCovFunc

NAME

createVecCovFunc - create function for objects in the `VecCovFunc` hierarchy

INCLUDE

```
include "createVecCovFunc.h"
```

SYNTAX

```
extern const char** hierVecCovFunc (); // table of classes that can be created
extern VecCovFunc* createVecCovFunc (const prm(VecCovFunc)& pm);
```

EXAMPLE

SEEALSO

class VecCovFunc, class prm(VecCovFunc)

AUTHOR

Turid Follestad, NR, from a template written by Hans Petter Langtangen, SINTEF/UiO

A.2.10 prm(VecCovFunc)

NAME

prm(VecCovFunc) - parameters for the `VecCovFunc` hierarchy

INCLUDE

```
include "VecCovFunc.h"
```

SYNTAX

```
class prm(VecCovFunc)
{
 public:
  char* classname;           // name of derived class
  double range;              // practical range
  double variance;           // variance (nugget + partial sill)
  double nugget;             // nugget effect
  double beorder;            // order of modified Bessel function

  prm(VecCovFunc)()          // default constructor
    { classname = NULL; range = 0.0; variance = 0.0; nugget = 0.0;}

  void fill(char* cname, double r, double v, double n, double beo=0.0)
    { classname = cname; range = r; variance = v; nugget = n; beorder=beo;}

  void scan(Is in);          // reads from input stream
  void print(Os out) const;  // prints to output stream
};
```

KEYWORDS

covariance, vector field, parameters, initialization

DESCRIPTION

The class contains all parameters needed to initialize an instance of a class in the `VecCovFunc` hierarchy.

CONSTRUCTORS AND INITIALIZATION

The class has a default constructor, initializing all members to zero.

MEMBER FUNCTIONS

`fill` - function for initialization of the data members. The members might as well be initialized manually. The parameters are the classname (`cn`), the range (`r`), the variance (`v`), the nugget effect (`n`) and optionally the order (`beo`) of the modified Bessel functions used in class `BessVecCovFunc`.

FILES

VecCovFunc.C

SEEALSO

class VecCovFunc, createVecCovFunc

AUTHOR

Turid Follestad, NR

A.2.11 createTrend

NAME

createTrend - create function for objects in the `Trend` hierarchy

INCLUDE

```
include "createTrend.h"
```

SYNTAX

```
extern const char** hierTrend (); // table of classes that can be created
extern Trend* createTrend (const prm(Trend)& pm);
```

EXAMPLE

See class Trend.

SEEALSO

class Trend, class prm(Trend)

AUTHOR

Turid Follestad, NR, from a template written by Hans Petter Langtangen, SINTEF/UiO

A.2.12 prm(Trend)

NAME

prm(Trend) - parameters for the `Trend` hierarchy

INCLUDE

```
include "Trend.h"
```

SYNTAX

```
class prm(Trend){
 public:
  char* classname;        // name of derived class

  VecSimplest(TFP) tfvec; // general trend functions
  nMatrix Fmatr;          // regressor matrix

  int order;              // number of coefficients
  int polOrder;           // order of polynomial trend
  int dim;                // dimension

  prm(Trend)() : tfvec(), Fmatr()
    { classname = "PolTrend"; order = 1; polOrder = 0; dim = 2;}

  void fill(char* cn, int ord, VecSimplest(TFP) vtf)
    { classname = cn; order = ord; polOrder = 0; dim = 0;}
  void fill(char* cn, nMatrix F)
    { classname = cn; order = F.getCdim(); Fmatr = F; polOrder = 0; dim = 0;}
  void fill(char* cn, int ord, int po, int d)
    {
      classname = cn; order = ord; dim = d; polOrder = po;
      if (strncmp(classname,"PolTrend",8)==0)
        ord = binCoef(d+po,d);
    }

  void scan(Is istr);
  void print(Os ostr) const;
};
```

KEYWORDS

trend, parameters, initialization

DESCRIPTION

The class contains the parameters needed to create an instance of an object of a class in the `Trend` hierarchy.

CONSTRUCTORS AND INITIALIZATION

The class has one constructor, a default constructor. By default, the parameters are set to correspond to a polynomial trend of (polynomial) order zero, i.e. a constant trend, and the dimension is initialized to 2.

Further initialization is done by the `fill` - functions, or could be done manually.

MEMBER FUNCTIONS

`fill` - overloaded function for initializing the data members. The members might as well be initialized manually, since all members are public. The arguments of the three functions are specified to initialze objects of type `GenTrend`, `PolTrend` and `TrendMat` respectively.

`scan` - reads parameter values from an input stream. The member `tfvec` will be allocated with the indicated size, but the components of the vector must be initialized manually. The member `Fmatr` must be initialized manually. The parameter values are expected to be ordered as follows:

```
classname

spatial dimension

order

polynomial order
```

The values not needed for one particular choice of classname, can be given arbitrary values. For the polynomial trend the value of `order` will be computed from the value of `polOrder`.

`print` - prints classname and parameters to an output stream.

FILES

Trend.C

EXAMPLE

See class Trend.

SEEALSO

class Trend, createTrend

AUTHOR

Turid Follestad, NR

A.3 Modified Bessel functions

A.3.1 Bessel functions

NAME

modified Bessel functions

INCLUDE

```
include "bessel_functions.h"
```

SYNTAX

```
// modified Bessel functions of first kind:
double besselI0 (double ba);       // order 0
double besselI1 (double ba);       // order 1

// modified Bessel functions of second kind:
double besselK0 (double ba);       // order 0
double besselK1 (double ba);       // order 1
double besselK (int n, double ba); // order n>=2
```

KEYWORDS

Bessel functions, modified Bessel functions

DESCRIPTION

These functions are modified Bessel functions of first (I) kind of order 0 and 1, and of second kind (K) of general order. They are used to compute the parmeteric semivariogram based on modified Bessel functions.

The functions are adapted from Press, Flannery, Teukolsky and Vetterling: "Numerical Recipes in C", Cambridge University Press, 1988.

FILES

bessel_functions.C

AUTHOR

Adapted from "Numerical Recipes in C", by Turid Follestad, NR.

A.4 Minimization procedure for semivariogram parameter estimation

A.4.1 Minimizer

NAME

Minimizer - minimization procedures

INCLUDE

```
include "Minimizer.h"
```

SYNTAX

```
class Minimizer
{
public:
   Minimizer(int, PS, PSFV, const nVector& low, const nVector& upp);
   Minimizer(int, PS, PSFV);
   ~Minimizer(){delete [] istate; }
   int find_opt(nVector&, double, int);
   void restart_mode_on(){restart_mode = 1;}
                    // restart minimizer when iterating
   void set_scale(int i, double s){scale(i) = s;}   // sets scaling values for
                                                    // the parameters
   double get_scale(int i) const{return scale(i) ;} // gets scalig values

   void verbose_mode_off(){ verbosity=0;}   // no output during optimization
   friend OPF optfunc_;
};
```

KEYWORDS

estimation, minimization, optimization, parameters, semivariogram estimation

DESCRIPTION

The class contains routines for minimization of a function used to estimate the parameters of a spatial semivarigram in the class `SemivarModel`. The minimization is performed by invoking the FORTRAN NAG-routine E04UCF.

CONSTRUCTORS AND INITIALIZATION

The class has two constructors, taking three and five arguments. The integer argument indicates the length of the parameter vector to be optimized, PS is a pointer to class `SemivarModel`, and PSF a pointer to a `SemivarModel` member function taking a const `nVector` reference argument. The two `nVector` arguments in the first constructor indicates lower and upper bounds for the parameter vector.

MEMBER FUNCTIONS

`find_opt` - finds optimum for the parameter vector, with initial values entered as a `nVector` argument. The double argument is supposed to be a tolerance, and the integer indicates the maximum number of iterations.

FILES

Minimizer.C

SEEALSO

class SemivarModel

AUTHOR

Jon Helgeland, NR

Index