G_STL: THE GEOSTATISTICAL TEMPLATE LIBRARY IN C^{++}

A REPORT

SUBMITTED TO THE DEPARTMENT OF PETROLEUM ENGINEERING

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Nicolas Remy

March 2001

I certify that I have read this report and that in my opinion it is fully adequate, in scope and in quality, as partial fulfillment of the degree of Master of Science in Petroleum Engineering.

Dr. Jef K. Caers
(Principal advisor)

Abstract

The development of geostatistics has been mostly acccomplished by application-oriented engineers in the past twenty years. The focus on concrete applications gave birth to a great many algorithms and computer programs designed to address very different issues, such as estimating or simulating a variable while possibly accounting for secondary information like seismic data, or integrating geological and geometrical data. At the core of any geostatistical data integration methodology is a well-designed algorithm.

Yet, despite their obvious differences, all these algorithms share a lot of commonalities one should capitalize on when building a geostatistics programming library, lest the resulting library is poorly reusable and difficult to expand.

Building on this observation, we design a comprehensive, yet flexible and easily reusable library of geostatistics algorithms in C++.

The recent advent of the generic programming paradigm allows us to elegantly express the commonalities of the geostatistical algorithms into computer code. Generic programming, also refered to as ”programming with concepts”, provides a high level of abstraction without loss of efficiency. This last point is a major gain over object-oriented programming which often trades efficiency for abstraction. It is not enough for a numerical library to be reusable, it also has to be fast.

Because generic programming is ”programming with concepts”, the essential step in the library design is the careful identification and thorough definition of these concepts shared by most of the geostatistical algorithms. Building on these definitions, a generic and expandable code can be provided.

To show the advantages of such a generic library, we use the GSTL to build two sequential simulation programs working on two very different types of grids: a surface with faults and an unstructured grid; without requiring any change to the GSTL code.

acknowledgements

First and foremost, I would like to express my appreciation to my advisor, Jef Caers, for giving me the opportunity to pursue this research and guiding me throughout this work. Working with him and Andre Journel in the geostatistics group has been an invaluable experience.

I am also very grateful to Bruno Levy, currently working at INRIA, the French agency for computer science research, for introducing me to the concept of Generic Programming. The discussions we had while he was at Stanford University were enlightening and most helpful. I would also like to express my gratitude to Arben Shtuka for his invaluable help and warm welcome during my stay at *gOcad*, France.

Finally, I would like to thank my wife for her constant support and belief in me.

Contents

Section 1

Introduction

The development of geostatistics has been mostly accomplished by application-oriented engineers in the past twenty years. The focus on concrete applications gave birth to a great many algorithms designed to address very different issues, such as estimating or simulating a variable while possibly accounting for secondary information like seismic data, or integrating geological and geometrical data.

In order for these algorithms to be tested and then applied to real cases, they have to be coded into a programming language. Making a computer executable available plays a capital role in popularizing an algorithm. However, despite the essential place of programmed algorithm in geostatistics, no programming library that implements the basic tools and algorithms of geostatistics exists (at least no such library is publicly available).

The main programming effort in geostatistics made publicly available is *GSLIB* [Deutsch and Journel, 1992], the Geostatistical Software Library. *GSLIB*, as its name suggests, is a collection of softwares, not a programming library: it provides a variety of computer executables which implement a broad family of algorithms, but it hardly provides a framework or tools for programming new softwares.

It was originally built with two goals in mind: the first one was to wide-spread the use of geostatistical algorithms developed at Stanford University. The second was to serve as a seed for research and new developments [Deutsch and Journel, 1992]. While *GSLIB* no doubt completed its first mission, adding new code or modifying the existing one has turned out really tedious. Most end-users either use *GSLIB* without making any change or have rewritten the programs to fit their own means (e.g. *gOcad*).

The purpose of this work is to propose a genuine programming library of geostatistical tools and algorithms.

It was designed with the following goals in mind:

- The new library should be usable both for research developments and direct applications. This means that the library should be flexible enough to serve a research clientele that requires a quick coding of new algorithms, as well serve a large Petroleum company willing to integrate easily a newly developed geostatistical application in their software platform.

- The new library should allow a fast reuse of existing code. This requires a thorough design of the library.

- The new library should be easily extendable. Expandability requires a library design that recognizes important concepts that are common to almost all geostatistical algorithms.

- We will propose a library that does not sacrifice reuseability for efficiency (in term of computing speed). Library optimization too often leads to incomprehensible code.

- The code should be understandable without too much computer science background.

The first important decision regards the selection of a programming language. C++ is retained for both computer science reasons and practical reasons. C++ is a high level programming language whose usage is now wide-spread. This is important to produce understandable code and reach as large a user-base as possible.

The second capital choice is to decide on a design for the library. As stated previously, the new library ought to be expandable and generic. This implies that it must recognize the key concepts that are recurring in geostatistical algorithms, and capitalize on them to produce a generic implementation of the algorithms.

The solution retained to obtain an abstract and generic programming code is often object-oriented programming. However, object-oriented programming is not the only possible solution. Chapter 2 introduces a more recent and probably less known programming paradigm, *Generic Programming*, and explains why it was preferred to the more classical object oriented approach.

Instead of working directly with actual data types ("classes" in C++), a generic algorithm works on abstractions or *concepts*, which are assumed to have precise properties. The key step in the design of *generic* algorithms is thus the precise identification of the minimal set of properties the algorithms need to assume in order to perform efficiently. This is the aim of Chapter 3. Chapter 4 then thoroughly describes the identified properties.

Chapter 5 is a reference manual that describes the algorithms and objects provided by the library. Finally, Chapter 6 gives an example of how to extend the library by implementing a new algorithm and how the library could be integrated into an existing software.

Section 2

Library Design

2.1 Generic Programming

Algorithms detail the procedure for solving a specific set of problems. In order to make the usage of these procedures as widespread as possible, the programming of algorithms should be generic. A generic code is achieved by removing from the algorithm's implementation any unnecessary information, i.e. any data-structure or object that the code relies on but is not essential to the algorithm itself.

Consider for example the sequential Gaussian simulation algorithm for a Gaussian variable [Ripley, 1987; Journel, 1989; Isaaks, 1990]. The core "idea" of sequential Gaussian simulation is to simulate a series of values by sequential drawing from Gaussian distributions whose parameters are determined through kriging. It can be summarized as follows:

1. define a path visiting all the nodes of the simulation grid

2. for each node **u** in the path:

(a) find the node's informed neighbors. The neighbors can be nodes from the original data set (n), or nodes simulated at previous iterations (l).

(b) estimate the Gaussian cumulative distribution $G^*(\mathbf{u}; \mathrm{y} \mid (\mathrm{n}+\mathrm{l}))$ at $\mathbf{u}$ conditional to the neighbors (n+l) by solving a kriging system. The mean of $G^*(\mathbf{u}; \mathrm{y} \mid (\mathrm{n}+\mathrm{l}))$ is the kriging estimate and its variance is the kriging variance.

(c) draw a realization from $G^*(\mathbf{u}; \mathrm{y} \mid (\mathrm{n}+\mathrm{l}))$ by Monte-Carlo simulation, and assign the simulated value to the node

An implementation of this algorithm for a Cartesian grid would unnecessarily restrict its potential domain of application. The sequential Gaussian simulation algorithm does not indeed require the grid to be Cartesian. As long as a path through all the grid nodes can be defined, this algorithm can be applied to any type of grid, be it Cartesian or unstructured, 1D, 3D or nD.

Similarly, the path defined at the beginning of the algorithm is usually taken random in practical applications. However this is not imposed by the algorithm, and one could choose a path that visits preferentially nodes close to the original set of data.

A truly generic implementation of the sequential simulation algorithm should therefore be independent of the type of the grid or the type of the path.

In modern computing, one of the most usual way to tend to this aim is to use object-oriented programming. In object-oriented programming, the genericness of the algorithms' implementation is provided through the use of *inheritance* and *dynamic binding*. The algorithm is written for abstract types (or objects), e.g. an "AbstractGrid", an "AbstractPath", and will work on objects that represent particular cases of these abstract objects: the algorithm would be defined in terms of "AbstractGrid", but will be used on "CartesianGrid" or "UnstructuredGrid" which are particular types of grid that *inherit* from "AbstractGrid".

This approach is most useful when the entities dealt with are similar but not identical, i.e. when they can be grouped into objects hierarchies. If this is not the case, forcing an object oriented approach, i.e. forcing a taxonomy of the entities dealt with, leads to awkward designs. The use of inheritance and dynamic binding also has a major drawback in scientific programming: it induces non-negligible run-time overhead which can badly hurt CPU performance. These points will be developed in more detail in section 2.2

Object-Oriented programming is not the only way of achieving a high level of abstraction however. Generic programming is a fairly new[1] programming paradigm that allows to elegantly abstract the program implementation from any unnecessary information. Instead of working directly with actual data types ("classes" in C++), a generic algorithm works on abstractions (often called *concepts*) which are assumed to have precise properties (the fewer the assumed properties, the more generic the implementation). A generic algorithm is thus made of two parts: an actual program code, and a list of all the assumed properties of the abstractions used. This list of properties is not C++ code[2], yet it is a integral part of the algorithm. These properties are the hypotheses of the algorithm. Omitting them is as damaging as omitting to state the hypotheses of a mathematical theorem.

To illustrate how this works, consider the simple case of finding the maximum of a set of elements. The set could be an array, a linked list, . . . , and its elements real numbers, strings, cars, . . . To find the maximum of this set, one only requires:

1. a method to go from one element of the set to another

2. an order relation is defined on the elements of the set, and given two elements, one

[1]Early research papers on generic programming are actually 20 years old, but no example of generic programming had come out of research groups before 1994. *STL*, the C++Standard Template Library, was the first example of generic programming to become important as it was included in the C++ standard library.

[2]Other languages like Ada actually have keywords for specifying the assumptions made on the abstractions used by the algorithm. C++ does not. This makes the task of defining the assumptions critical: since there is no compiler check, it is the programmer's burden to ensure that all the assumptions are clearly defined.

knows how to compare them

The algorithm would then be implemented as follows:

```
1  template<class iterator, class comparator>
2  iterator find_maximum(iterator first,
3                        iterator last,
4                        comparator greater){
5
6    // initialize iterator max_position, the iterator
7    // that points to the largest element found so far
8    iterator max_position = first;
9
10   // iterate through the container
11   for(iterator current=first++ ; current!=last; current++)
12     {
13       if ( greater(*max_position , *current) )
14         max_position = current;
15     }
16
17   return max_position;
18 }
```

The first line indicates that algorithm `find_maximum` refers to two concepts: *iterator* and *comparator*. The algorithm assumes these two concepts have the following properties:

iterator : It is the device used to go through the set. One can think of it as a generalized pointer. An iterator is a classical way to make the code independent of the container (set of elements) it is applied to. Different kinds of iterators are detailed in

[Austern, 1999]. The `find_maximum` algorithm assumes an iterator has the following properties:

- an iterator can be assigned to another (line 8: `max_position = first`)
- two iterators can be compared using `!=` (line 11: `current!=last`)
- operator `++` can be applied to an iterator, and it will move the iterator to the next position in the set of elements (line 11: `current++`)
- operator `*` can be applied to an iterator, and it will return the element the iterator is pointing to (line 13: `*current`)

comparator :

- a comparator has an operator `()` which takes two objects as argument and returns a type convertible to `bool`.
  For example: `greater(*max_position,*current)` (line 12).
  It returns "true" if the first argument is greater than the second.

The previous C++ code and its two sets of requirements form the generic `find_maximum` algorithm. Any C++ object that fulfills the 4 requirements of concept *iterator* is an eligible iterator for the algorithm and can be an input of `find_maximum`. Such an object is called a **model** of **concept** *iterator*. On the other hand, trying to use as an *iterator* an object which does not meet the four requirements of *iterator* will result in a compile-time or link-time error.

Type `double*` is a valid model of *iterator* because it has the four properties required by concept *iterator*. A call to

```
find_maximum(double* an_array,
             double* an_array+10,
             greater_doubles() )
```

will then find the maximum of the array `an_array` which contains 10 elements of type `double`. Here `greater_doubles` is a model of concept *comparator*, i.e. it takes two doubles as argument and returns a type convertible to boolean. Nothing prevents a model of a concept to be implemented in a generic way, that is to use concepts of its own. Type `greater_doubles` could for example be defined as follows:

```
template<class ordered_set_element>
class greater_generic{
  public:
    bool operator()(ordered_set_element& arg1,
                    ordered_set_element& arg2)
      {
        return arg1 > arg2;
      }
};

// define greater_doubles as the particular case:
// ‘‘ordered_set_element’’ is ‘‘double’’
typedef greater_generic<double> greater_doubles;
```

Where `ordered_set_element` is assumed to be a type for which operator `>` is valid, this operator returning a type convertible to `bool`. In this example a model of concept *comparator* is defined using another concept: `ordered_set_element`.

Similarly, `find_maximum` can be applied to a STL list of characters without any change to its implementation, because the STL type `list<char>::iterator` has the four properties of *iterator*. The comparator could be `greater_generic<char>` because characters support comparison through operator `>` (type `char` is a model of concept `ordered_set_element`):

```
list<char> stl_list;

// initialize list ...

// find maximum of stl_list
list<char>::iterator max_position = find_maximum(stl_list.begin(),
                                                 stl_list.end(),
                                                 greater_generic<char>());
```

Given a set of requirements, one can write any model of the concepts and use them in any generic algorithm that needs these concepts; without requiring any change to the implementation of the algorithm.

2.2 Generic Programming is NOT Object-Oriented Programming

At first sight, there might seem to be little conceptual difference between generic and object-oriented programming. A concept could be thought of as an abstract object, and a model of a concept would simply be an object derived from the abstract object-concept.

How would the `greater_generic` functor[3] be implemented in an object-oriented way? The first step is to turn the concept `ordered_set_element` into an actual C++ data type, call it `OrderedSetElement_OBJECT`. The requirement of `ordered_set_element` was: a type for which operator `>` is valid, this operator returning a type convertible to `bool`:

```
class OrderedSetElement_OBJECT{
  public:
    virtual bool operator>(ordered_set_element_OBJECT& B) = 0;
};
```

Note that this object is not a strict equivalent to the `ordered_set_element` concept: the return type of `OrderedSetElement_OBJECT`'s operator `>` is a boolean, which is less general than the "type convertible to `bool`" required by `ordered_set_element`. This is however of lesser importance, and `OrderedSetElement_OBJECT` could certainly be modified so as to return a "type convertible to `bool`", probably at the expense of code simplicity. Using this abstract object, the object-oriented programming counterpart of `greater_generic` would be:

[3] a functor is simply an object that behaves like a function

```
class greater_OOP{
  public:
    bool operator()(OrderedSetElement_OBJECT& arg1,
                    OrderedSetElement_OBJECT& arg2)
      {
        return arg1 > arg2;
      }
};
```

To compare two real numbers, one would then derive a `real_number` class from `OrderedSetElement_OBJECT`, define the `>` operator and call `greater_OOP`.

Although the code of `greater_OOP` and `greater_generic` look quasi-identical, there is actually a key difference: `greater_OOP` allows to compare any two objects derived from `OrderedSetElement_OBJECT`, for example a string of characters and a real number, which has no meaning ! The generic implementation did impose the two arguments to be of the same type.

Object-Oriented programming and generic programming do not express the same ideas: inheritance, the medium of object-oriented programming, expresses the relationship between two types. Modeling (making a model out of a concept), the generic programming counterpart of inheritance, is a relationship between a set of types and a type: a concept is the set of all the types that meet the concept's requirements; a model is one of these types. One of these relationships can not emulate the other.

There is another major difference, though less conceptual, between generic programming and object-oriented programming (at least as implemented in C++). The genericness obtained through object-oriented programming is usually obtained at the cost of speed. The

use of virtual functions and dynamic binding indeed causes a runtime overhead which can badly hurt performance, essentially when the functions are simple (no time consuming operation is performed) and frequently called. A function which compares two elements like `greater_OOP` or `greater_generic` has to be very fast since it is likely to be used very often in the program.

In the case of generic algorithms, the compiler adapts the generic code to the particular types (models of the algorithm's concepts) requested. Schematically, the generic code is a template that the compiler uses to write a new implementation, replacing every occurrence of a concept by its model. This results in an algorithm potentially as fast as a hand-crafted algorithm, specific to a single type.

2.3 Library Design

Generic programming allows to elegantly attain a high level of abstraction. It has many advantages that make it an interesting choice of paradigm for implementing a library of geostatistics algorithms.

Its most obvious advantage is efficiency. Contrary to object-oriented programming, generic programming enables to write generic code while retaining the efficiency usually only achieved by a specific, hand-crafted implementation, such as the current *GSLIB* programs. It is indeed essential that a scientific computing library be as fast as possible, as long as no sacrifice to code readability and re-usability is made.

A second and maybe more subjective advantage of generic programming is its conceptual similarity with mathematics. Mathematics is based on abstract concepts, which are assumed to have precise properties. A theorem will hold true for any specific case which verifies the theorem's hypotheses. Similarily, a generic algorithm can be applied to

an objects that satisfy its hypotheses, i.e. satisfy its concepts' requirements. It is actually possible to elegantly define mathematical algebraic structures like groups, rings or fields with generic programming [Barton and Nackman, 1994]. Expressing an algorithm in the generic programming way is thus more natural than adopting the object-oriented approach. This makes generic programming very suitable for implementing geostatistics algorithms.

However, the choice of generic programming as a guiding programming paradigm does not prevent the use of other paradigms like object-oriented programming. The only restriction is that genericness and efficiency must be maintained.

After defining the algorithms to be implemented, a critical task in the design of the new library is the careful identification of the most general set of requirements that allows the algorithms to perform efficiently. As underlined previously, a "generic" code is useless if the concepts used are not thoroughly defined. This is the last part of the library design.

Section 3

Concepts Identification

The first step in the design of GsTL is to analyze the algorithms to be implemented, and identify the minimum set of requirements that allow these algorithms to perform efficiently. Some of these requirements might be common to all algorithms, while others may be more particular to specific algorithms.

The goal of geostatistics is to study and characterize phenomenons that vary in space (and/or time). Geostatistics has two principal applications:

- estimation, i.e. the mapping of a spatially and/or timely dependent variable z, through regression techniques. Estimation often provides a single number, termed estimate, and an associated error variance.
- simulation, used to assess the uncertainty on a spatially and/or timely dependent variable z, quantified through a series of numbers or possible outcomes, allowing risk quantification.

These two applications of geostatistics are reviewed and detailed in the following sections with the purpose of identifying the key concepts of geostatistics.

3.1 Estimation

Consider a set $\mathbb{U}$ of locations in space or time. In practical applications, $\mathbb{U}$ is finite, of size N. Suppose that the value of z is known on a subset of $\mathbb{U}$. The aim is to estimate the values of z, interpreted as the realization of a regionalized random variable $Z(\mathbf{u})$, at any location $\mathbf{u}$ in $\mathbb{U}$ given the known z-values $\{z(\mathbf{u}_\alpha),\ \alpha = 1, \ldots, \mathrm{n}\}$.

For a given loss function L, the best estimate $z^*(\mathbf{u})$ of unknown value $z(\mathbf{u})$ is the estimate that minimizes the expected loss:

$$z^*(\mathbf{u}) = \underset{\hat{\mathrm{z}}}{\operatorname{argmin}}\ \mathrm{E}\Big\{\mathrm{L}\Big(\hat{\mathrm{z}}, \mathrm{Z}(\mathbf{u})\Big)\Big\}$$

Kriging is the name of a family of generalized linear least square regression algorithms [Krige, 1951; Goovaerts, 1997]. The estimate $Z^*(\mathbf{u})$ is modeled as a linear combination of the known z-values $\{z(\mathbf{u}_\alpha)\}$:

$$Z^*(\mathbf{u}) - \mathrm{m}(\mathbf{u}) = \sum_{\alpha=1}^{\mathrm{n}} \lambda_\alpha [\mathrm{Z}(\mathbf{u}_\alpha) - \mathrm{m}(\mathbf{u}_\alpha)] \tag{3.1}$$

where $m(\mathbf{u})$ and $m(\mathbf{u}_\alpha)$ are the expected values of $Z(\mathbf{u})$ and $Z(\mathbf{u}_\alpha)$.

Under the unbiasedness constraint:

$$E\Big(Z^*(\mathbf{u}) - \mathrm{Z}(\mathbf{u})\Big) = 0$$

minimizing the expected loss amounts to minimizing the error variance:

$$\sigma_E^2(\mathbf{u}) = \mathrm{Var}\Big(\mathrm{Z}^*(\mathbf{u}) - \mathrm{Z}(\mathbf{u})\Big) \tag{3.2}$$

Substituting $Z^*(\mathbf{u})$ in (3.2) by its expression (3.1) and setting to zero all the derivatives $\frac{\partial \sigma_E^2(\mathbf{u})}{\partial \lambda_\alpha}$ yields a system of linear equations whose solution is the weights $\lambda_\alpha,\ \alpha = 1, \ldots, n$. The system is of the form:

$$\begin{pmatrix} C(\mathbf{u}_1, \mathbf{u}_1) & \ldots & C(\mathbf{u}_1, \mathbf{u}_\mathrm{n}) \\ \vdots & \ddots & \vdots \\ C(\mathbf{u}_\mathrm{n}, \mathbf{u}_1) & \ldots & C(\mathbf{u}_\mathrm{n}, \mathbf{u}_\mathrm{n}) \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_n \end{pmatrix} = \begin{pmatrix} C(\mathbf{u}, \mathbf{u}_1) \\ \vdots \\ C(\mathbf{u}, \mathbf{u}_\mathrm{n}) \end{pmatrix}$$

where $C(\mathbf{u}_\mathrm{i}, \mathbf{u}_\mathrm{j})$ is the covariance between $Z(\mathbf{u}_\mathrm{i})$ and $Z(\mathbf{u}_\mathrm{j})$.

Combining the weights $\lambda_1, \ldots, \lambda_n$ according to (3.1) provides the best linear least-squares estimate $Z^*(\mathbf{u})$.

Many variants of kriging have been developed, but all rely on the same concepts.

Three types of kriging can be differentiated depending on the model used for $m(\mathbf{u})$:

Simple kriging: the mean is known and constant for all locations in $\mathbb{U}$:

$$\forall \mathbf{u} \in \mathbb{U} \quad \mathrm{m}(\mathbf{u}) = \mathrm{m}$$

The kriging problem is then to find (λ_α) such that:

$$Var\Big(\sum_{\alpha=1}^{n} \lambda_\alpha [Z(\mathbf{u}_\alpha) - \mathrm{m}] - [\mathrm{Z}(\mathbf{u}) - \mathrm{m}] \Big) \quad \text{is minimum}$$

Ordinary kriging: the mean is unknown but is locally constant. The kriging problem then becomes to find (λ_α) such that:

$$\begin{cases} Var\Big(\sum_{\alpha=1}^{n} \lambda_\alpha [Z(\mathbf{u}_\alpha) - \mathrm{m}] - [\mathrm{Z}(\mathbf{u}) - \mathrm{m}] \Big) \quad \text{is minimum} \\ \\ \sum_{\alpha=1}^{n} \lambda_\alpha = 1 \end{cases}$$

The constraint $\sum_{\alpha=1}^{n} \lambda_\alpha = 1$ filters the mean m out of the first condition, hence alleviating the need for knowing m:

$$Var\Big(\sum_{\alpha=1}^{n} \lambda_\alpha [Z(\mathbf{u}_\alpha) - \mathrm{m}] - [\mathrm{Z}(\mathbf{u}) - \mathrm{m}] \Big) = \mathrm{Var}\Big(\sum_{\alpha=1}^{\mathrm{n}} \lambda_\alpha \mathrm{Z}(\mathbf{u}_\alpha) - \mathrm{Z}(\mathbf{u}) \Big)$$

if $\sum_{\alpha=1}^{n} \lambda_\alpha = 1$.

Kriging with Trend: the mean is unknown and varies smoothly with location:

$$m(\mathbf{u}) = \sum_{\mathrm{k}=0}^{\mathrm{K}} \mathrm{a_k}(\mathbf{u}) \mathrm{f_k}(\mathbf{u})$$

where a_k are unknown but locally constant and f_k are known functions of **u**. The kriging system at location **u** is then given by:

$$\begin{cases} Var\Big(\sum_{\alpha=1}^{n}\lambda_\alpha[Z(\mathbf{u}_\alpha)-\mathrm{m}(\mathbf{u}_\alpha)]-[\mathrm{Z}(\mathbf{u})-\mathrm{m}(\mathbf{u})]\Big) \quad \text{is minimum} \\ \sum_{\alpha=1}^{n}\lambda_\alpha = 1 \\ \sum_{\alpha=1}^{n}\lambda_\alpha(\mathbf{u})\mathrm{f_k}(\mathbf{u}_\alpha)=\mathrm{f_k}(\mathbf{u}) \quad \forall \mathrm{k}\in[1,\mathrm{K}] \end{cases}$$

Kriging can also be made to account for secondary information by extending equation (3.1). Suppose n_v secondary variables $S_i(\mathbf{u})$, $i = 1, \ldots, n_v$ are to be accounted for, equation (3.1) becomes:

$$\begin{aligned} Z^*(\mathbf{u})-\mathrm{m}(\mathbf{u}) &= \sum_{\alpha=1}^{n}\lambda_\alpha[Z(\mathbf{u}_\alpha)-\mathrm{m}(\mathbf{u}_\alpha)] \\ &\quad +\sum_{i=1}^{n_v}\sum_{\alpha_1=1}^{n_i}\lambda_{\alpha_i}[S_i(\mathbf{u}_{\alpha_i})-\mathrm{m_i}(\mathbf{u}_{\alpha_i})] \end{aligned} \tag{3.3}$$

where $m_i(\mathbf{u}_\mathrm{j})$ is the expected value of $S_i(\mathbf{u}_\mathrm{j})$. This version of kriging is called cokriging.

The kriging weights are obtained by minimizing the error variance as defined in 3.2. As in the single variable case, different models can be assumed for the means $m(\mathbf{u}_\mathrm{j})$ and $m_i(\mathbf{u}_\mathrm{j})$, hence leading to three types of cokriging.

All these methods require solving possibly large systems of linear equations, depending on the number of conditioning data $z(\mathbf{u}_\alpha)$ and secondary data. Hence, in order to reduce computation costs, only the data closest to the location **u** being estimated are accounted for. These data will be refered to as the *neighborhood* of **u**. This approximation is acceptable because the closest data tend to screen the influence of further away data: the weights associated with the distant data are usually negligible.

From an algorithmic point of view, kriging and its variants can be decomposed into two parts:

- a *weighting system* which to location $\mathbf{u}$, neighborhood $V(\mathbf{u})$ and set of covariance and cross-covariance functions $C_{ij} = Cov(Z_i'(\mathbf{u}), \mathrm{Z_j'}(\mathbf{u}+\mathbf{h}))$ (Z_i' can either be Z or one of the secondary variables S_k, $k = 1, \ldots, n_v$) associates a set of kriging weights and a kriging variance (the kriging variance is independent of the values $z(\mathbf{u}_\alpha)$):

$$\Big(\mathbf{u}, \mathrm{V}(\mathbf{u}), \{\mathrm{C_{ij}}\}\Big) \longmapsto \Big(\{\lambda_\alpha\}_{1\leq\alpha\leq\mathrm{n}(\mathbf{u})}\ ,\ \sigma^2(\mathbf{u})\Big)$$

  The cross-covariance functions between variables i and j are only needed in the case of cokriging. For kriging with a single variable, the set $\{C_{ij}\}$ is a single covariance function.

  The system of equations leading to the kriging weights is composed of a set of equations common to all kriging variants to which different equations are added to account for additional constraints, e.g an unknown locally constant mean, or an unknown smoothly varying mean. Hence the weighting system consists, in the most general case, of two parts: a first part accounts for the correlation and the redundancy between the data through the covariance functions, while a second part, implements the additional constraint equations.

- a *combiner*, which from the previous weights and an a-priori mean, computes the kriging estimate:

$$\Big(\{\lambda_\alpha\}\ ;\ \Big\{z(\mathbf{u}_\alpha)\Big\}\ ;\ \mathrm{m}\Big) \longmapsto \mathrm{z^*(u)} \qquad 1 \leq \alpha \leq \mathrm{n(u)}$$

  where m is the a-priori mean.

  The *combiner* is a mere linear combination:

$$\sum_{\alpha=1}^{n(\mathbf{u})} \lambda_\alpha z(\mathbf{u}_\alpha) + \lambda_\mathrm{m}\mathrm{m}$$

with

$$\lambda_m = 1 - \sum_{\alpha=1}^{n(\mathbf{u})} \lambda_\alpha$$

Notice that in ordinary kriging and kriging with a trend, the weight λ_m associated with the mean m is 0. Hence the actual value of m which is input to the *combiner* has no influence on the estimate.

Note that other types of kriging have been developed, like block-kriging, which are not covered in the previous overview of kriging. However, using the kriging techniques described previously, data at different scales can still be accounted for. Chapter 6 details how kriging can be constrained to a block average value by using cokriging.

3.2 Simulation

The aim of simulation is to find a function

$$\begin{cases} \mathbb{U} & \longrightarrow & \mathbb{E}^N \\ (\mathbf{u}_i)_{1\leq i\leq N} & \longmapsto & \Big(z(\mathbf{u_i})\Big)_{1\leq i\leq N} \end{cases}$$

such that the sequence of values $z(\mathbf{u}_i)$ $i = 1, \ldots, N$, honors a set of constraints ($\mathbb{E}$ is the space in which z is valued). The constraints can be of various type:

- local equality constraints, or data conditioning: the value of the variable is known at a subset of locations $(\mathbf{u}_\mathrm{j})$ $\mathrm{j} = 1, \ldots, \mathrm{K} < \mathrm{N}$. This constraint is of great importance in many applications of geostatistics.

- inequality constraints: the values of the variable must be lesser or greater than a given threshold $t(\mathbf{u})$ at a subset of locations $(\mathbf{u}_\mathrm{j})$ $\mathrm{j} = 1, \ldots, \mathrm{K} \leq \mathrm{N}$.

- correlation constraint: the values of the variable must honor a given model of correlation. Most often a variogram is imposed, but more complicated models, which involve the correlation between more than two locations at a time, could be chosen.

- histogram constraints: the values must match a given histogram which could for example reflect some prior knowledge of variable z.

- other variables correlated to z are known, possibly at all locations, and thus impose a constraint on the values of z. For example, in petroleum applications, z could be rock permeability, and the constraining variable the pressure drop observed during a well test.

Because the set of constraints does usually not suffice to fully characterize the sequence $\Big(z(\mathbf{u_i})\Big)$, many solutions exist. Different solutions, termed realizations, provide a model of the uncertainty about the unknown $Z(\mathbf{u})$.

Four types of simulation algorithms can be distinguished:

Sequential simulation. A path visiting all locations is defined and each location is simulated sequentially. The variable to be simulated is interpreted as a regionalized random variable $Z(\mathbf{u})$. At each location $\mathbf{u}$ the cumulative function distribution (cdf) $F\Big(\mathbf{u}, \mathrm{Z} \mid (\mathrm{n})\Big)$ conditional to some information (n), is estimated and sampled. As in kriging, the conditioning information is sought only in the vicinity of the location to be simulated, in order to reduce computation costs. Contrary to kriging, the conditioning information includes both the original data (if any) and the previously simulated values. Sequential simulation is the most versatile class of simulation algorithms due to its low CPU demand and its large potential to integrate various data types.

P-field. The p-field simulation is divided into two parts: first a cdf $F(\mathbf{u}, \mathrm{Z} \mid (\mathrm{n}'_{\mathbf{u}}))$ conditional to only the original data $(n'_{\mathbf{u}})$ is estimated at each location $\mathbf{u}$ to be simulated ((n') depends on $\mathbf{u}$ if only the closest original data are retained at each location $\mathbf{u}$). The family of conditional cdfs (ccdf) $\Big(F(\mathbf{u}, \mathrm{Z} \mid (\mathrm{n}'_{\mathbf{u}})\ )\Big)_{\mathbf{u}\in\mathbb{U}}$ is then sampled using a field of correlated probability values (p-field). The generation of the p-field can be made very fast by using methods based on the fast Fourier transform (FFT), hence yielding a computationally efficient class of simulation algorithms. P-field however has a major drawback: a map simultated by p-field can present un-desired artifacts, especially discontinuities at data locations.

Boolean simulation. The aim of boolean techniques is to reproduce shapes described by specific parameterizations, which honor the original data (n'). For example, it can be used to simulate channels of given sinuousities and extent, or ellipses parametrized by their dimensions and orientations. This simulation technique fits well into the generic programming approach since, at least for unconditional simulation (i.e. without any sample data), the only difference between two boolean algorithms is the object description. However, boolean algorithms are not provided in the current release of GsTL.

Optimization techniques. Instead of approaching the simulation problem from a statistical point of view, i.e. interpreting the variable to be simulated as a regionalized random variable, simulation can be envisioned as a mere optimization problem: the satisfaction of the constraints is measured through an objective function which must be minimized. Deutsch (1992) proposed to use simulated annealing [Geman and Geman, 1984] to minimize the objective function. This class of simulation techniques is not implemented in the current release of GsTL.

This first release of GsTL focuses on sequential simulation and p-field simulation. These two simulation paradigms interpret the sequence of values $z(\mathbf{u_i})$, $i = 1, \ldots, N$, to be simulated as an outcome of the sequence of random variables $Z(\mathbf{u_i})$, $i = 1, \ldots, N$. The two simulation algorithms proceed as follows:

1. Define a partition $I = (P_j)_{1 \leq j \leq J}$ of {1; ... ;N}:

$$\begin{cases} \bigcup_{1 \leq j \leq J} P_j = \{1; \ldots; N\} \\ \forall j \neq j' \quad P_j \bigcap P_{j'} = \emptyset \end{cases}$$

2. For each P_j, visited in a pre-defined order,

   (a) for every $i \in P_j$, estimate the cumulative distribution of $Z(\mathbf{u}_i)$ conditional to some neighboring data $V(\mathbf{u}_i)$:

$$\Big(\mathbf{u}_i \ , \ \mathrm{V}(\mathbf{u}_i)\Big) \quad \longmapsto \quad F\Big(\mathbf{u}_i, \mathrm{Z} \mid (\mathrm{n}(\mathbf{u}_i))\Big)$$

   (b) for every $i \in P_j$, draw a realization from $F\Big(\mathbf{u}_i, \mathrm{Z} \mid (\mathrm{n}(\mathbf{u}_i))\Big)$

$$F\Big(\mathbf{u}_i, \mathrm{Z} \mid (\mathrm{n}(\mathbf{u}_i))\Big) \quad \longmapsto \quad z(\mathbf{u_i})$$

If the P_j are singletons, the algorithm described is sequential simulation. If $I = \{1; \ldots; N\}$, the algorithm described belongs to the p-field family.

Varying the order of visit of the P_j, the way the cumulative distributions are estimated, and the way new values are deduced from the cdf's, provide a broad family of algorithms.

Order of visit of the P_j

In p-field simulation, there is only one set of indices P_1 ($J = 1$), hence there is no order to decide.

In sequential simulation, each cdf is conditional to only the neighboring data $V(\mathbf{u})$, and visiting each location along a "structured" path (e.g. column by column, if the locations are arranged in a Cartesian grid) could create artificial continuity. Hence a random path is usually chosen in practice. However, other types of path could be used, for example a path that would preferentially visit locations close to the original data, so as to increase the weight of the original data and possibly improve the data conditioning.

Some techniques like MCMC simulation also use a completely random "path", allowing locations to be visited many times. In MCMC simulation, the set of locations to be simulated is initialized with some arbitrary values (random for example). This set of values is then sequentially modified, until it honors the constraints: at a randomly selected location, a sample of a cdf model is generated. This new sample value can either be accepted and replace the former value at that location, or be rejected, in which case the location's value is unchanged. The key lies in defining the correct acceptance probability in order to reproduce a given variogram or histogram and constraint to other data types. The process is then iterated until convergence. MCMC algorithms are not sequential algorithms from a theoretical point of view, but they follow the same scheme, and hence could share the same implementation: the cdf at a given location is estimated, conditional to the neighboring information, and is sampled. The sampled value is either retained or rejected, and the algorithm proceeds to a new random location.

Estimation of the conditional cdf's

Two approaches can be distinguished:

- First: the cdf is built from estimated values. If the variable $Z(\mathbf{u})$ is multi-Gaussian, all cdf $F\left(\mathbf{u}_i, \mathrm{Z} \mid (\mathrm{n}_{\mathbf{u}_i})\right)$ are also Gaussian, and it suffices to estimate two values: a mean and a variance. When no Gaussian assumption is made, the cdf is estimated

for given z-values $z_1, \ldots, z_k$ and an interpolation of these estimates $F_Z^*(\mathbf{u}, \mathrm{z_i} \mid (\mathrm{n})\ )$ yield a model of the function $z \longmapsto F_Z(\mathbf{u}, \mathrm{z} \mid (\mathrm{n})\ )$.

Most simulation algorithms estimate these values by kriging. In the case of a Gaussian cdf, the mean is the kriging estimate, and the variance the kriging variance. In the non-parametric case, the probabilities $F_Z(\mathbf{u}, \mathrm{z_i} \mid (\mathrm{n})\ ) = \mathrm{Prob}(\mathrm{Z}(\mathbf{u}) \leq \mathrm{z_i} | (\mathrm{n}))$ are estimated by kriging the indicator random variable $I(\mathbf{u}, \mathrm{z_i})$ defined as follows:

$$i(\mathbf{u}, \mathrm{z_i}) = \begin{cases} 1 & \text{if } z(\mathbf{u}) \leq \mathrm{z_i} \\ 0 & \text{otherwise} \end{cases}$$

The conditional probability $F(\mathbf{u}, \mathrm{z_i} \mid (\mathrm{n})\ )$ is indeed equal to the conditional expectation of $I(\mathbf{u}, \mathrm{z_i})$:

$$F_Z(\mathbf{u}, \mathrm{z_i} \mid (\mathrm{n})\ ) = \mathrm{E}\Big[\mathrm{I}(\mathbf{u}, \mathrm{z_i}) \mid (\mathrm{n})\Big]$$

and the least squares estimate of the indicator $i(\mathbf{u}, \mathrm{z_i})$ is also the kriging (least-squares) estimate of its conditional expectation [Luenberger, 1969].

- A second possibility is to infer the ccdf directly from the neighboring information, i.e. no estimation of parameters of a ccdf is required. The cdf can for example be read from a table which entries are the conditioning data values and geometry. It is the method used in the sequential normal equation simulation (SNESIM) algorithm [Strebelle, 2000]. The ccdf can also be inferred by a classification algorithm like a neural network [Caers and Journel, 1998].

Drawing new values

The new simulated value is usually obtained by drawing a value from the ccdf, using uncorrelated random probabilities. This is the technique used in sequential Gaussian simulation,

sequential indicator simulation or sequential normal equation simulation. However, it is not the sole option.

The p-field technique uses a field of correlated "random" probabilities to draw from the cdf's.

The MCMC approach also uses a different sampling scheme, called the Metropolis-Hastings sampling scheme: a new value is drawn from a cdf using uncorrelated random probabilities, but it does not automatically become the simulated value. It is indeed retained or discarded, with a given probability.

From this brief description of the different families of geostatistics algorithms, certain concepts common to most, if not all algorithms emerge :

- A location: coordinates in space or time.
- A geo-value: a location plus a single property value.
- A geovalue-iterator: the device that allows to go through the set of geo-values to be simulated or estimated.
- A neighborhood: most generally, only the data closest to the location of interest are taken into account in order to decrease the computation cost. However, if speed is not an issue, the neighborhood can be made large enough to always include all the available data.
- A cdf (cumulative distribution function): it can represent a conditional, marginal or likelihood distribution. It is either parametric (Gaussian, . . .) or non-parametric, i.e. defined by a finite set of values $F_Z(Z_i)$ at thresholds Z_i: $\Big(z_i\ ,\ F_Z(z_i)\Big)$.

- A cdf-estimator: to provide an estimate of the cdf, be it marginal or conditional. An estimator can either directly estimate a cdf given a node and its neighborhood as in SNESIM or built the cdf from estimated values, as in sequential Gaussian or indicator simulation.

- A sampler: determines the new simulated value given a cdf.

These concepts, along with others more specific to certain algorithms, are thoroughly described in the next chapter.

Section 4

Concepts Definitions

In chapter 3 various geostatistical algorithms are analyzed, and some recurring concepts are identified (like the notion of neighborhood or the notion of conditional cdf estimation). Building on these concepts, a library of generic algorithms can then be implemented (see Chapter 5 for a comprehensive description of these algorithms). However, as explained in Chapter 2, these algorithms are of little value if the concepts they rely on are not thoroughly defined. The aim of this chapter is thus to thoroughly define the concepts used by GsTL algorithms.

The definition of each concept and algorithm follows the layout used by Austern (1999). The different fields of the description are the following:

Refinement of: concept B is said to be a refinement of concept A if the requirements of A are a subset of the requirements of B. Consider the example of a Forward Iterator. A forward iterator is an object that points to other objects. It is used to iterate over a range of objects, in a single direction: it does not allow to go backward. The concept of Forward Iterator hence requires an operator `++` to advance to the next object of the range, an operator `*` which allows access to the value the iterator is pointing

to, an operator `!=` to compare two iterators, and an operator `=` which assigns an iterator to another. A Bidirectional Iterator is an iterator which allows to iterate over a range of objects in both forward and backward directions. It thus requires the same four operators as Forward Iterator, plus operator `--`, which moves the iterator to the previous object in the range. Since the requirements of Forward Iterator are a subset of the requirements of Bidirectional Iterator, Bidirectional Iterator is a refinement of a Forward Iterator. Following this definition, if concept B is a refinement of concept A, any *model* of B is also a *model* of A.

Associated types: a list of C++ types associated to the concept

Notations: some notations used in the remaining of the description of the concept

Valid expressions: a list of expressions that can be applied to a model of the concept. For instance, if `iter` is a model of the afore-mentioned Forward Iterator concept, such valid expression could be `iter++` or `*iter`.

Models: some examples of models of the concept.

4.1 Basic concepts

4.1.1 Forward Iterator

Associated Types

- **Value Type**

  `I::value_type`

  The type obtained by dereferencing (applying operator `*`) to a model of Forward Iterator.

Notations

I	A type that is a model of Forward Iterator
i,j	objects of type I

Valid Expressions

- **Assignment**

  `i=j`

Return type:	a type that is convertible to bool
Semantics:	`j` is assigned to `i`

- **Preincrement**

  `++i`

Return type:	I
Precondition:	i is dereferenceable
Semantics:	`i` is modified to point to the next value
Postcondition:	`i` is dereferenceable or one past the end

- **Postincrement**

  `i++`

Return type:	I
Precondition:	`i` is dereferenceable
Semantics:	`i` is modified to point to the next value
Postcondition:	`i` is dereferenceable or past the end

- **dereference**

  `*i`

Return type:	G
Precondition:	`i` is incrementable (operator `++` can be applied to `i`)
Semantics:	Returns the element `i` is pointing to.

- **comparison**

  `i!=j`

Return type:	a type convertible to `bool`
Semantics:	Returns true if `i` is different from `j`, i.e `i` and `j` are pointing to different elements.

4.1.2 Container

A Container is an object that stores other objects (the container elements) and has facilities to access its elements.

Associated Types

- **Value Type**

  `A::value_type`

  The type of the elements of the container.

- **Iterator type**

  `A::iterator`

  An iterator that points to the Container's elements.

Notations

A	A type that is a model of Container
a,b	objects of type A

Valid Expressions

- **Copy constructor**

  `X(a)`

  Return type: `X`

  Semantics: `X()` contains a copy of each element of a

- **Copy constructor**

  `X b(a)`

  Semantics: b contains a copy of each element of a

- **Assignment operator**

  `a=b`

  Return type: `X&`

  Semantics: b contains a copy of each element of a

- **Beginning of range**

  `a.begin()`

  Return type: iterator

  Semantics: returns an iterator pointing to the first element of the Container

- **End of range**

  `a.end()`

Return type:	iterator
Semantics:	returns an iterator pointing to one past the last element of the Container

- **Size**

  `a.size()`

Return type:	a type that represents a positive integer
Semantics:	returns the number of elements in the Container

- **Empty**

  `a.empty()`

Return type:	a type that is convertible to bool
Semantics:	returns “true” if the size of the Container is 0

4.1.3 Location

A location is an element of a d-dimensional Euclidean space $\mathbb{E}$. It is represented by its Cartesian coordinates: $(p_0, p_1, \ldots, p_{d-1})$.

Associated Types

- **Coordinate type**

  `A::coordinate_type`

  The type of the coordinates p_i, $0 \leq i \leq d-1$. Operations +, -, * must be defined on this type. The coordinate type should also be convertible to a type that supports operation /.

Notations

A	A type that is a model of Location
a, b	Objects of type L
C	coordinate type: `L::coordinate_type`
i	object of type `unsigned int`

Valid Expressions

- **Coordinate Access Function**

  `a[i]`

Return type:	C
Precondition:	$0 \leq i \leq d-1$
Semantics:	returns the i^{th} coordinate of the Location

- **Coordinate Access Function**

  `a == b`

Return type:	`bool`
Semantics:	checks if two locations are identical

Models

- `location2d`

- `location3d`

4.1.4 Geo-Value

A Geo-Value is the base element a geostatistical algorithm operates on. It is defined by a Location **u** and a property value, which can be either categorical ('sand', 'mud', ...) or continuous.

Associated Types

- **Property type**

  `A::property_type`

  The type of the property value hold by the Geo-Value. It could be a floating point type (`double`, `float`) or a "discrete type" (`int`, `bool`, ...).

- **location type**

  `A::location_type`

  A type that is a model of Location.

Notations

A	A type that is a model of Geo-Value
a	Object of type G
P	the type of the property associated to A
p	Object of type P
L	the type of the location associated to A

Valid Expressions

- **Access property value**

`a.property_value()`

Return type: P

Semantics: returns a reference to the property value hold by the Geo-Value. Expression `a.property_value() = x` must be valid.

- **Get Location**

  `a.location()`

  Return type: L

  Semantics: returns the location of the Geo-Value

Models

- `node_2d`
- `node_3d`

4.1.5 Neighborhood

Call f a binary predicate taking two geo-values as arguments. If **u** is a geo-value, a Neighborhood of **u** in $\mathbb{U}$ is the set of geo-values $V(\mathbf{u})$ such that:

$$V(\mathbf{u}) = \{\mathbf{v} \in \mathbb{U} \quad / \quad \mathrm{f}(\mathbf{u}, \mathbf{v}) = \mathrm{true}\}$$

u is called the center of the Neighborhood. A neighbor is, of course, an element of the Neighborhood. The neighborhood can be made to contain no more than a fixed number of neighbors, even if more geo-values actually satisfy criterion f. The retained geo-values could be selected according to their variogram distance from the center, or according to

their "nature" (i.e. the geo-value's property is a "hard datum" or is a previously simulated value), etc.

In geostatistics two types of neighborhoods are often used: elliptical neighborhoods and window (or template) neighborhoods. An elliptical neighborhood is a neighborhood for which $f(\mathbf{u}, \mathbf{v}) = \text{true}$ if $\mathbf{v}$ is inside a given ellipsoid centered on $\mathbf{u}$. A window neighborhood, is defined by a set of vectors $\mathbf{h}_1, \ldots, \mathbf{h}_n$ and:

$$f(\mathbf{u}, \mathbf{v}) = \text{true} \quad \text{if} \quad \exists\, j \in [1, n] \quad \mathbf{v} = \mathbf{u} + \mathbf{h}_j$$

Refinement of

Container

Associated Types

- **Neighborhood Iterator**

  `A::iterator`

  A neighborhood iterator is a model of Forward Iterator (see previous description and [Austern, 1999]). It is an iterator that returns a geo-value when dereferenced, and supports operator `=` (assignment), operator `++` (increment), operator `*` (dereference) and operator `!=` (comparison).

Notations

A	A type that is a model of Neighborhood
a	Object of type N
I	A type the is a model of Forward Iterator
u	An object of a type that models Location

Valid Expressions

- **find neighbors**

  `a.find_neighbors(u)`

  Return type: `void`

  Semantics: finds the neighbors of location u and stores them

- **begin**

  `a.begin()`

  Return type: I

  Semantics: returns an iterator to the first geo-value in the neighborhood

- **end**

  `a.end()`

  Return type: I

  Semantics: returns an iterator to one past the last geo-value in the neighborhood

- **size**

  `a.size()`

  Return type: `unsigned int`

  Semantics: returns the number of geo-values in the neighborhood

- **Empty**

  `a.empty()`

  Return type: a type that is convertible to bool

  Semantics: returns “true” if the size of the neighborhood is 0

Models

- `elliptical_neighborhood`
- `window_neighborhood`

4.1.6 Cdf

It is a cumulative distribution function (cdf):

$$F_Z : \quad z \longmapsto Prob\{Z \leq z\}$$

It can be either defined analytically (e.g. a Gaussian cdf, or an exponential cdf) or by a family of couples $\Big(z_i, F_Z(z_n)\Big)$, $i = 1, \ldots, n$ (non-parametric cdf).

Associated Types

- **Value Type**

  `A::value_type`

  The type of z (see definition above).

Notations

A	A type that is a model of Cdf
a	Object of type X
z	Object of type X::value_type
P	A type that represents a real number (e.g. float, double)
p	an object of type P

Valid Expressions

- **Cdf Evaluation**

  `a.prob(z)`

Return type:	P
Semantics:	returns $p = Prob\{Z \leq z\}$
Postcondition:	$0 \leq p \leq 1$ (p is a probability)

- **Cdf Inverse**

  `a.inverse(p)`

Precondition:	$0 \leq p \leq 1$ (p is a probability)
Return type:	`X::value_type`
Semantics:	returns the value z such that $p = Prob\{Z \leq z\}$

Definition

A Cdf is valid if

- $\forall z \quad F_Z(z) \in [0, 1]$.
- F_Z is a monotonous, increasing function.

This means for example that order relation problems (resulting from some indicator-based algorithms) must be corrected before the object is actually a valid cdf.

Models

- `gaussian_cdf`
- `discrete_variable_non_parametric_cdf`

4.1.7 Non-Parametric Cdf

A non-parametric cdf of variable Z is a cdf F defined by a discrete set of points $\Big(z_i, F(z_i)\Big)$, $i = 1, \ldots, n$. If variable Z is categorical, the z_i are all the possible values of Z, and the points $\Big(z_i, F(z_i)\Big)$ fully describe the cdf of Z.

If on the other hand Z is a continuous variable, the points $\Big(z_i, F(z_i)\Big)$ must be interpolated in order to associate a probability to any z-value.

Associated Types

- **Value Type**

  `A::value_type`

  The type of z (see definition above).

- **Z-iterator**

  `A::z_iterator`

  The type of the iterator to the values $z_1, \ldots, z_n$ (see definition above).

- **Z-iterator**

  `A::p_iterator`

  The type of the iterator to the values $p_1, \ldots, z_n$ (see definition above).

Refinement of

Cdf

Notations

A	A type that is a model of Cdf
a	Object of type X
m	object of type `unsigned int`

Valid Expressions

- **Resize**

  `a.resize(m)`

  Return type: void

  Semantics: Redefines the size of the discretizations $z_1, \ldots, z_n$ and $p_1, \ldots, z_n$. Spzce for m values is allocated.

- **Access to the beginning of the z-set**

  `a.z_begin()`

  Return type: `z_iterator`

  Semantics: provides an iterator to the z_i's

- **Access to the end of the z-set**

  `a.z_end()`

  Return type: `z_iterator`

  Semantics: provides an iterator to the z_i's

- **Access to the beginning of the p-set**

  `a.p_begin()`

  Return type: `p_iterator`

  Semantics: provides an iterator to the p_i's

- **Access to the end of the p-set**

  `a.p_end()`

Return type:	`p_iterator`
Semantics:	provides an iterator to the p_i's

4.2 Function Objects

A *function object*, or *functor*, is an object that can be called using an ordinary function call. It can be a pointer to a function or an object with a member function `operator()`. For example, object `sine` is a function object:

```
struct sine{
  double operator(double x) {return sin(x);}
};
```

because if `my_sine` is of type `sine`, `my_sine` can be called by: `my_sine(x)` and return $sin(x)$, just as if `my_sine` was a function.

4.2.1 Sampler

A sampler determines a value z according to a cdf F. Different methods are possible. The most used is Monte-Carlo simulation: a probability p is determined randomly and z is such that $p = F(z)$. Another solution would be to use constant probabilities instead of random ones. Hence z would be a given quantile of the cdf.

Associated Types

None

Notations

A	A type that is a model of Sampler
a	an object of type A
C	A type that is a model of Cdf
c	Object of type C

Valid Expressions

- **Number generation**

  `a(c)`

Return type:	`C::value_type`
Semantics:	returns a realization of random variable Z described by its cdf c.

Models

- `random_sampler`

  Draws a value from a cdf using Monte-Carlo simulation.

- `quantile_sampler`

  Returns a given quantile of the cdf.

4.2.2 Covariance

A covariance is a positive-definite function that characterizes the correlation between two random variables. In geostatistics, the covariance is computed between two stationary

random variables $Z_1(\mathbf{u})$ and $Z_2(\mathbf{u}+\mathbf{h})$ (if $Z_1 \neq Z_2$ the covariance is actually a called a cross-covariance), hence is a function of two locations:

$$C: \ (\mathbf{u}_1, \mathbf{u}_2) \longmapsto \mathrm{C}\Big(\mathrm{Z}_1(\mathbf{u}_1), \mathrm{Z}_2(\mathbf{u}_2)\Big)$$

Associated Types

None

Notations

A	A type that is a model of Covariance
a	an object of type A
L	A type that is a model of Location
u1, u2	two objects of type L
T	A type that represents a real number (e.g. float, double)

Valid Expressions

- **Compute covariance**

  `a(u1,u2)`

Return type:	`T`
Semantics:	returns the covariance $C($`u1`,`u2`$)$.

Models

- `spherical_covariance`
- `exponential_covariance`

4.2.3 Kriging Constraint

A kriging system can be divided into two parts: one accounts for the correlation and the redundancy between the data modeled through the covariance, while the other expresses various constraints as: “the kriging weights have to sum-up to 1”. A functor that models `Kriging Constraint` sets up this second part of the system. It takes in charge of defining the total size of the kriging system and filling in the kriging matrix entries belonging to non-covariance terms.

Notations

A	a type that is a model of Kriging Constraint
a	an object of type A
M	an object of type Matrix (see the description of kriging in 5.4.1 for a detailed description of this type)
V	an object of type Vector (see the description of kriging in 5.4.1 for a detailed description of this type)
first	a pointer to an array of pointers to “neighborhoods”
Nv	the number of pointers to neighborhoods in the array “first” points to
u	an object of type that models Location

Valid Expressions

- **Set-up the system**

```
a(M,V,u,first,Nv)
```

Return type:	a type convertible to `unsigned int`
Semantics:	The total size N of the kriging system is computed from the total number of neighboring data. The kriging matrix M is then resized to $\mathrm{N} * \mathrm{N}$, and vector V (second member of the kriging system) is resized to N. Finally, the part of the system that does not depend on the covariances is computed, possibly using location `u` and the neighboring geo-values of `u`. The returned value is the total number of conditioning data, i.e. the sum of the sizes of the all neighborhoods.

Models

- `OK_constraints`

  Imposes the constraints of ordinary kriging

- `KT_constraint`

  Imposes the constraints of kriging with trend

4.2.4 Covariance set

Cokriging of variable Z_1 accounting for variables $Z_2, \ldots, Z_M$ requires the covariances between all the variables: $C_{i,j}(\mathbf{h})$, $i, j = 1, \ldots, M$. These covariances are specified through a Covariance Set. Different models can actually be used to determine these covariances. The full cokriging approach (LMC approach) is very demanding because it requires the complete knowledge of all covariance functions $C_{i,j}(\mathbf{h})$. Modeling cross-covariances from data is a difficult task, because all the covariances can not be modeled independently from

one another. The Markov models MM1 and MM2 alleviate the difficulties of full cokriging by using only the collocated secondary variables: the cokriging of location $\mathbf{u}$ depends on the neighboring values of $Z_1(\mathbf{u}+\mathbf{h_j})$ and only the collocated variables $Z_i(\mathbf{u})$, $i = 2, \ldots, M$ (instead of many neighborhoods of variables: $Z_i(\mathbf{u} + \mathbf{h_j^i})$). Hence the covariances $C_{i,j}(\mathbf{h})$, $i, j = 2, \ldots, M$ need only be modeled for distance $\mathbf{h} = 0$. Moreover, the covariances $C_{1,j}$, $j = 2, \ldots, M$ are (approximately) proportional to $C_{1,1}$ or $C_{j,j}$, depending on the Markov approximation used (MM1 or MM2).

Notations

A	a type that is a model of Covariance set
a	an object of type A
L	a type that models Location
u1, u2	objects of type L
i, j	objects of type convertible to `unsigned int`

Valid Expressions

- **Set-up the system**

  `a(i,j, u1,u2)`

Return type:	a type convertible to `double`
Semantics:	returns the covariance value $C_{i,j}(\mathbf{u}_1, \mathbf{u}_2)$

Models

- `LMC_covariance`
- `MM1_covariance`
- `MM2_covariance`

4.2.5 Single Variable Cdf Estimator

A Single Variable Cdf Estimator (more precisely a conditional-cdf estimator) estimates a conditional distribution function of variable Z at a location **u** given some neighboring Z-values. It can estimate parametric cdf's (usually Gaussian) or non-parametric cdf's, using different approaches: kriging, search trees (SNESIM [Strebelle, 2000]) or even neural networks ([Caers and Journel, 1998]).

Notations

A	a type that is a model of cdf estimator
a	an object of type A
f	an object of a type that models Cdf
u	an object of a type that models a location
V	an object of a type that models a neighborhood

Valid Expressions

- **Estimate cdf**

  `a(u,V,f)`

  Return type: int

  Semantics: estimates the "parameters" of cdf `f` (mean and variance in the case of a Gaussian cdf, or the probabilities $Prob(Z \leq z_i) = F(z_i)$, $i = 1, \ldots, n$, if F is non-parametric). The returned value is 0 if no problem was encountered during the execution of the function.

Models

- `gaussian_cdf_Kestimator`: estimator of a gaussian cdf using kriging (K)
- `indicator_cdf_Kestimator`: estimator of a non-parametric cdf using indicator kriging
- `search_tree_estimator`: estimator of a non-parametric cdf using a search tree (SNESIM)

4.2.6 Multiple Variables Cdf Estimator

A Multiple Variables Cdf Estimator estimates a conditional distribution function of variable Z_1 at a location **u** given some neighboring information. This information can consist of outcomes of several different variables $Z_2, \ldots, Z_{N_v}$. A Multiple Variables Cdf Estimator can estimate parametric cdf's (usually Gaussian) or non-parametric cdf's, using different approaches: kriging, search trees (SNESIM [Strebelle, 2000]).

Notations

A	a type that is a model of cdf estimator
a	an object of type A
f	an object of a type that models Cdf
u	an object of a type that models a location
V	a pointer to an array of pointers to objects of a type that models Neighborhood
Nv	the number of variables (primary and secondary)

Valid Expressions

- **Estimate cdf**

  `a(u,V,Nv,f)`

Return type:	int
Semantics:	estimates the "parameters" of cdf `f` (mean and variance in the case of a Gaussian cdf, or the probabilities $Prob(Z \leq z_i) = F(z_i)$, $i = 1, \ldots, n$, if F is non-parametric). The returned value is 0 if no problem was encountered during the execution of the function.

Models

- `gaussian_cdf_coKestimator`: estimator of a gaussian cdf using cokriging (coK)

4.3 Iterators

4.3.1 Geo-Value Iterator

A geo-value iterator is an iterator on a set of geo-values. This set could be for example a simulation grid, a region of a simulation grid or a Neighborhood. Geo-value iterators can be random paths through the set of geo-values or deterministic paths. These paths can be constrained to visit every geo-value only once, or they can allow multiple visits to the same geo-value. Some iterative simulation methods start the simulation with a seed image which

is iteratively modified. Each location to be modified is chosen randomly, and the same geo-value can be changed twice in a row, while other geo-values would never be visited.

Refinement of

Forward Iterator

Associated Types

- **Value Type**

  `I::value_type`

  The type obtained by dereferencing (applying operator `*`) to a model of Geo-Value Iterator.

Notations

I	A type that is a model of geo-value iterator
i,j	objects of type I
G	A type that is a model of Geo-Value

Valid Expressions

- **Assignment**

  `i=j`

Return type:	a type that is convertible to bool
Semantics:	`j` is assigned to `i`

- **Preincrement**

  `++i`

Return type:	I
Precondition:	i is dereferenceable
Semantics:	`i` is modified to point to the next value
Postcondition:	`i` is dereferenceable or one past the end

- **Postincrement**

  `i++`

Return type:	I
Precondition:	`i` is dereferenceable
Semantics:	`i` is modified to point to the next value
Postcondition:	`i` is dereferenceable or past the end

- **dereference**

  `*i`

Return type:	G
Precondition:	`i` is incrementable (operator `++` can be applied to `i`)
Semantics:	Returns the element `i` is pointing to.

- **comparison**

  `i!=j`

Return type:	a type convertible to `bool`
Semantics:	Returns true if `i` is different from `j`, i.e `i` and `j` are pointing to different elements.

Models

- `random_path`
- `deterministic_path`

Section 5

Algorithms and Classes

GSTL has three main constituents: a library of generic algorithms, a thorough description of the concepts used by these algorithms (Chapter 4), and finally a library of models of the concepts, i.e. objects that meet the requirements of the concepts defined in Chapter 4. This Chapter is a reference manual to GSTL. It details all the algorithms and models of concepts available in GSTL.

5.1 Algorithms

The descriptions of the algorithms follow the same layout used by Austern (1999):

- The algorithm prototype is stated. Some algorithms can have multiple prototypes. The first version of the algorithm usually assumes some default behavior, which can be overriden by using the second version.
- **Preconditions** lists all the conditions to be met before the algorithm can be used.
- **Requirement on types** details what the types of the algorithms' arguments must be.

- Finally, a very simple example is given.

All the examples illustrating each algorithm's description refer to the two following simple implementations of a model of Location and Geo-Value:

```
class location2d{
  public:
    typedef int coordinate_type;
    location2d(int X, int Y) {coord[0]=X; coord[1]=Y;}
    int& operator[](unsigned int i) {
      assert(i<2); return coord[i];
    }

  private:
    int coord[2];
};
class geo_value2d{
  public:
    typedef double property_type;
    typedef location2d location_type;
    geo_value2d();
    geo_value2d(location2d u, double prop) : loc(u), pval(prop) {};
    const location_type& location() const {return loc;}
    property_type& property_value() {return pval;}
    const property_type& property_value() const {return pval;}

  private:
    double pval;
    location2d loc;
};
```

5.1.1 Construct Non-Parametric Cdf

1. ```
   template<class non_param_cdf, class random_iterator>
   void
   build_cdf(random_iterator first, random_iterator last,
             non_param_cdf& new_cdf, unsigned int nb_of_thresholds)
   ```

2. ```
   template<class non_param_cdf, class random_iterator>
   void
   build_cdf(random_iterator first, random_iterator last,
             non_param_cdf& new_cdf)
   ```

This function builds the cdf F of a random variable Z from a set of outcomes of Z, contained in range `[first,last)`. Cdf F is a function defined by a set of thresholds z_i and the associated probabilities $F(z_i)$, $i = 1, \ldots, n$.

In version 1, function argument `nb_of_thresholds` indicates the number n of thresholds to be used to define the cdf. The n values retained are n equally-spaced quantiles of the distribution. For example, if $n = 5$, z_1 is the smallest value in range `[first,last)`, z_2 is the first quartile, z_3 is the median, z_4 is the upper quartile, and z_5 is the highest value in range `[first,last)`. With this version of the algorithm, $F(z_1) = Prob(Z < z_1) = 0$ and $F(z_n) = Prob(Z < z_n) = 1$.

Version 2 of the algorithm assumes that the cdf `new_cdf` is already initialized: it already contains the values z_i, $i = 1, \ldots, n$, and function `build_cdf` computes the corresponding probabilities $F(z_i)$. This version gives a complete flexibility in the choice of the thresholds values z_i.

Note that range `[first,last)` will be modified by `build_cdf` (it will be sorted). If the range must not be modified, a copy should be made first.

Where defined

In header file `<univariate_stats.h>`

Preconditions

- The range `[first,last)` is a valid range.
- The values in range `[first,last)` are of type `cdf::value_type`, or are convertible to this type.

Requirements on types

- `random_iterator` is a model of Random Access Iterator. Random Access Iterator is a refinement of Forward Iterator. If `i` is a model of Random Access Iterator pointing to the i-th element of a range, `i-n` is an iterator to the (i-n)-th element of the range, `i[n]` is equivalent to `*(i+n)`, and `i<j` compares to iterators. For a thorough description of Random Access Iterator, see [Austern, 1999].

- `non_param_cdf` is a model of Non-Parametric Cdf.

Example

```
int main()
{
  gaussian_cdf normal_cdf(0,1);

  vector<double> gaussian_values;

  for(int i=1; i<=100; i++)
    gaussian_values.push_back( normal_cdf.inverse(drand48()) );

  non_parametric_cdf new_cdf;

  build_cdf(gaussian_values.begin(), gaussian_values.end(),
            new_cdf);
}
```

`new_cdf` now contains a discrete representation of a standard normal cdf, and the elements of `gaussian_values` are sorted.

5.1.2 CDF Transform

```
template<class target_cdf, class data_cdf, class forward_iterator>
void
cdf_transform(forward_iterator first, forward_iterator last,
              data_cdf& from, target_cdf& to)
```

`cdf_transform` transforms the range of values `[first,last)` so that their final cumulative distribution function is `to`. Cdf `from` is the cumulative distribution function before the data are transformed. It could be computed with `build_cdf`.

Where defined

In header file `<univariate_stats.h>`

Preconditions

- The range `[first,last)` is a valid range.
- The values in range `[first,last)` are of type `taret_cdf::value_type`, or are convertible to this type.

Requirements on types

- `forward_iterator` is a model of Forward Iterator.
- `target_cdf` is a model of CDF.
- `data_cdf` is a model of CDF.

Remark

The values in range `[first,last)` are overwritten by the transformed values.

Example

Transform 100 uniformly distributed values so that the transformed values follow a standard normal distribution:

```
int main()
{
  vector<double> uniform_values;

  for(int i=1; i<=100; i++)
    uniform_values.push_back( rand() );

  non_param_cdf from;
  vector<double> tmp(uniform_values);
  build_cdf(tmp.begin(), tmp.end(), from);

  gaussian_cdf normal_cdf(0,1);

  cdf_transform(uniform_values.begin(), uniform_values.end(),
                from, normal_cdf);
}
```

5.1.3 Kriging Weights

1. ```
   template<class location, class neighborhood,
            class covariance, class kriging_constraints,
            class Vector>
     double
     kriging_weights(Vector& weights,
                     const location& center, const neigborhood* neighbors,
                     covariance& covar, kriging_constraints& Kconstraint);
   ```
   ```

2.
```
template<class matrix_lib,
          class location, class neighborhood,
          class covariance, class kriging_constraints,
          class Vector>
  double
  kriging_weights(Vector& weights,
                  const location& center, const neigborhood* neighbors,
                  covariance& covar, kriging_constraints& Kconstraint);
```

The `kriging_weights` algorithm solves a kriging system :

$$\begin{cases} Var\Big(\sum_{\alpha=1}^{n} \lambda_\alpha [Z(\mathbf{u}_\alpha) - \mathrm{m}(\mathbf{u}_\alpha)] - [\mathrm{Z}(\mathbf{u}) - \mathrm{m}(\mathbf{u})]\Big) \quad \text{is minimum} \\ \\ f_1\Big(\{\lambda_\alpha\}\Big) = 0 \\ \vdots \\ f_p\Big(\{\lambda_\alpha\}\Big) = 0 \end{cases}$$

where $f_i\Big(\{\lambda_\alpha\}\Big) = 0$, $i = 1, \ldots, p$ are linear constraints expressed by the Kriging Constraints `Kconstraint`. The solution to this system is a set of weights:

$$(\lambda_1, \ldots, \lambda_n, \mu_1, \ldots, \mu_p)$$

where the weights μ_j are the Lagrange weights used to account for constraints $f_1, \ldots, f_p$.

The `kriging_weights` function returns the kriging variance and stores the kriging weights into `Vector weights` in the following order: $(\lambda_1, \ldots, \lambda_n, \mu_1, \ldots, \mu_p)$. Only the weights $\lambda_1, \ldots, \lambda_n$ are useful to compute the kriging estimate. The weights $\mu_1, \ldots, \mu_p$ are used to compute the kriging variance which is returned by the function. `Vector weights` does not need to be of the correct size $n + p$ when passed to the function: the function automatically resizes the vector if necessary.

Version 2 of the algorithm allows to change the linear algebra library (which defines matrices, matrix inversion routines, ...) used by `kriging_weights` (see 5.4).

Where defined

In header file `<kriging.h>`

Requirements on types

- `Vector` is a container on which an iterator is defined. It must have three member functions whose prototypes are:

  1. `iterator Vector::begin()` which returns an iterator to the first element of the `Vector`
  2. `size_type Vector::size()` which returns the size of the `Vector`.
  3. `void Vector::resize(size_type n)` which changes the size of the `Vector` to n

- `location` is a model of Location.

- `neighborhood` is a model of Neighborhood. The location type of the geo-values contained in the neighborhood must be `location`.

- `covariance` is a model of Covariance that takes two objects of type `location` as arguments.

- `kriging_constraints` is a model of Kriging Constraints.

- `matrix_lib` specifies the linear algebra library to use. The requirements on `matrix_lib` are fully defined in 5.4. By default, it is the *TNT* library (slightly modified), a public domain library by Roldan Pozo, Mathematical and Computational Sciences Division, National Institute of Standards and Technology. The library is freely available from `http://math.nist.gov/tnt/`.

Remarks

- `Vector weights` is resized after it is passed to `kriging_weights`, unless it is of the correct size. The cost of this resize can be decreased by smartly managing the `Vector`'s memory (in the same style as an STL `vector`): the vector allocates more memory than it needs, hence does not need to re-allocate memory each time its size increases.

  However, each call to `kriging_weights` implies a matrix inversion, hence the cost of the resize would usually be negligible.

- If the same data $z(\mathbf{u}_1), \ldots, \mathrm{z}(\mathbf{u}_\mathrm{d})$ are used for estimating different locations, algorithm `global_neigh_kriging_weights` can be more suitable than `kriging_weights`. The kriging matrix does not depend on the location to be estimated, but only on the data locations $\mathbf{u}_1, \ldots, \mathbf{u}_\mathrm{d}$. Hence if the exact same data are used to estimate multiple locations, the kriging matrix remains unchanged: it can be inverted once for all and solving further kriging systems reduces to a mere matrix-vector multiplication. `global_neigh_kriging_weights` implements such "global kriging".

Example

Estimate $z(0,0)$ from $z(2,3) = 0.21$ and $z(4,-7) = 0.09$ by ordinary kriging. After the call to `kriging_weight`, vector `weights` has size 3; its two first elements are the weights corresponding to $z(2,3)$ and $z(4,-7)$ respectively, and its last element is the Lagrange parameter.

```
typedef std::vector<geo_value2d> neighborhood;

// gaussian covariance of range 4
inline double gauss_covariance(Location2d u1, Location2d u2){
  double square_dist = pow(u1[0]-u2[0], 2) +
                       pow(u1[1]-u2.[1], 2);
  return exp(-3*square_dist/16);
}


int main()
{
  location2d u1(2,3);
  location2d u2(4,-7);

  geo_value2d Z1(u1,0.21);
  geo_value2d Z2(u2,0.09);

  Neighborhood neighbors;
  neighbors.push_back(Z1);
  neighbors.push_back(Z2);

  location2d u(0,0);

  std::vector<double> weights;

  double kvariance = kriging_weights(weights,
                                     u, &neighbors,
                                     gauss_covariance, OK_constraint);
}
```

5.1.4 Cokriging Weights

1. ```
   template<class location, class neighborhood,
            class covariance_set, class kriging_constraints,
            class Vector>
     double
     cokriging_weights(Vector& weights,
                       const location& center,
                       const neigborhood** first_neigh,
                       unsigned int nb_of_neighborhoods,
                       covariance_set& covar,
                       kriging_constraints& Kconstraint);
   ```

2. ```
   template<class matrix_lib,
            class location, class neighborhood,
            class covariance_set, class kriging_constraints,
            class Vector>
     double
     cokriging_weights(Vector& weights,
                       const location& center,
                       const neigborhood** first_neigh,
                       unsigned int nb_of_neighborhoods,
                       covariance_set& covar,
                       kriging_constraints& Kconstraint);
   ```

The `cokriging_weights` algorithm solves a cokriging system :

$$\begin{cases} Var\Big(\sum_{\alpha=1}^{n} \lambda_\alpha [Z(\mathbf{u}_\alpha) - \mathrm{m}(\mathbf{u}_\alpha)] + \sum_{\mathrm{i}=1}^{\mathrm{N_v}} \sum_{\beta=1}^{\mathrm{n_i(u)}} \lambda_\beta^{\mathrm{i}} [\mathrm{Z}(\mathbf{u}_\beta^{\mathrm{i}}) - \mathrm{m}(\mathbf{u}_\beta^{\mathrm{i}})] \\ \qquad -[Z(\mathbf{u}) - \mathrm{m}] \Big) \quad \text{is minimum} \\ \\ f_1\Big(\{\lambda_\alpha\}, \{\lambda_{\alpha_1}\}, \ldots, \{\lambda_{\alpha_{N_v}}\} \Big) = 0 \\ \vdots \\ f_p\Big(\{\lambda_\alpha\}, \{\lambda_{\alpha_1}\}, \ldots, \{\lambda_{\alpha_{N_v}}\} \Big) = 0 \end{cases}$$

where $f_i\Big(\{\lambda_\alpha\}, \{\lambda_{\alpha_1}\}, \ldots, \{\lambda_{\alpha_{N_v}}\}\Big)$, $i = 1, \ldots, p$ are p constraints expressed by the `Kconstraints`. The solution to this system is a set of weights:
$(\lambda_1, \ldots, \lambda_n, \lambda_1^1, \ldots, \lambda_{n_{N_v}}^{N_v}, \mu_1, \ldots, \mu_p)$, where the weights μ_j are the Lagrange weights used to account for the constraints $f_1, \ldots, f_p$.

The `cokriging_weights` function returns the kriging variance and stores the kriging weights into `Vector weights` in the following order:
$(\lambda_1, \ldots, \lambda_n, \lambda_1^1, \ldots, \lambda_{n_{N_v}}^{N_v}, \mu_1, \ldots, \mu_p)$.

Only the weights $\lambda_1, \ldots, \lambda_{n_{N_v}}^{N_v}$ are useful to compute the kriging estimate. The weights $\mu_1, \ldots, \mu_p$ are used to compute the kriging variance which is returned by the function. `Vector weights` does not need to be of the correct size $n + p$ when passed to the function.

Version 2 of the algorithm allows to change the linear algebra library (which defines matrices, matrix inversion routines, ...) used by `cokriging_weights` (see 5.4).

Where defined

In header file `<kriging.h>`

Requirements on types

- `Vector` is a container on which an iterator is defined. It must have three member functions whose prototypes are:

  1. `iterator Vector::begin()` which returns an iterator to the first element of the `Vector`
  2. `size_type Vector::size()` which returns the size of the `Vector`.

3. `void Vector::resize(size_type n)` which changes the size of the `Vector` to n

- `location` is a model of Location.

- `neighborhood` is a model of Neighborhood. The location type of the geo-values contained in the neighborhood must be `location`. Cokriging requires multiple neighborhoods, `nb_of_neighborhoods` in total. Pointers to these neighborhoods must be stored in an array `first_neigh`, so that `first_neigh[0]` is a pointer to the first neighborhood.

- `covariance_set` is a model of Covariance Set. In expression `covar(i,j,u1,u2)`, required by Covariance Set, `u1` and `u2` must be of type `location`.

- `kriging_constraints` is a model of Kriging Constraints.

- `matrix_lib` specifies the linear algebra library to use. The requirements on `matrix_lib` are fully defined in 5.4. By default, it is the *TNT* library (slightly modified), a public domain library by Roldan Pozo, Mathematical and Computational Sciences Division, National Institute of Standards and Technology. The library is freely available from `http://math.nist.gov/tnt/`.

Remarks

`Vector weights` is resized after it is passed to `cokriging_weights`, unless it is of the correct size. The cost of this resize can be decreased by smartly managing the `Vector`'s memory (in the style of an STL `vector`). However, each call to `cokriging_weights` implies a matrix inversion, hence the cost of the resize would usually be negligible.

Example

Estimate $z(0,0)$ from $z(2,3) = 0.21$, $z(4,-7) = 0.09$ and one secondary data $s(1,3) = 42.1$ by ordinary (full-)cokriging. After the call to `kriging_weight`, vector `weights` has size 5; its three first elements are the weights corresponding to $z(2,3)$, $z(4,-7)$ and $s(1,3)$ respectively, and its two last elements are the Lagrange parameters.

```
typedef std::vector<geo_value2d> neighborhood;

int main()
{
  location2d u1(2,3);
  location2d u2(4,-7);

  geo_value2d Z1(u1,0.21);
  geo_value2d Z2(u2,0.09);

  Neighborhood neighbors1, neighbors2;
  neighbors1.push_back(Z1);
  neighbors1.push_back(Z2);

  location2d u3(1,2);
  geo_value2d S1(u3,42.1);
  neighbors2.push_back(S1);

  neighborhood* neigh_array[2]={&neighbors1, &neighbors2};

  location2d u(0,0);

  std::vector<double> weights;

  double kvariance = kriging_weights(weights,
                                     u, neigh_array, 2,
                                     LMC_covariance, OK_constraint);
}
```

5.1.5 Linear Combination

```
template<class iterator, class neighborhood>
double
linear_combination(iterator begin_weights, iterator end_weights,
                   const neigborhood* neighbors)
```

Computes the linear combination of the weights in range `[begin_weights, end_weights)`, and the property values of the geo-values contained in the neighborhood that `neighbors` points to. Denote $\lambda_1, \ldots, \lambda_P$ the weights, and $z(\mathbf{u}_1), \ldots, \mathrm{z}(\mathbf{u}_\mathrm{N})$ the geo-values property values. `linear_combination` returns: $\sum_{i=1}^{N} \lambda_i \; z(\mathbf{u}_\mathrm{i})$

Where defined

In header file `<kriging.h>`

Preconditions

P, the number of weights, must be equal or greater than the number of geo-values in the neighborhood. The algorithm loops on the geo-values, hence if $P > N$ only the N first weights are used, the others are ignored.

The type of the geo-values property must be convertible to `double`.

Requirements on types

- `iterator` is a model of Forward Iterator: it is assignable, default constructible, supports operator `++`, operator `*`, and two iterators can be compared with operator `!=`. The dereference type of the iterator must be convertible to `double`.

- `neighborhood` is a model of Neighborhood.

Example

Compute the kriging estimate from the kriging weights. `gauss_covariance` is the Gaussian covariance defined in the example following the description of `kriging_weights`.

```
typedef std::vector<geo_value2d> neighborhood;

int main()
{
  location2d u1(2,3);
  location2d u2(4,-7);

  geo_value2d Z1(u1,0.21);
  geo_value2d Z2(u2,0.09);

  Neighborhood neighbors;
  neighbors.push_back(Z1);
  neighbors.push_back(Z2);

  location2d u(0,0);

  std::vector<double> weights;

  double kvariance = kriging_weights(weights,
                                     u, &neighbors,
                                     gauss_covariance, OK_constraint);

  double kmean = linear_combine(weights.begin(),weights.end(),
                                &neighbors);
}
```

Replacing the last line by

```
  std::vector<double>::iterator end_weights = weights.begin()+2;

  double kmean = linear_combine(weights.begin(), end_weights,
                                &neighbors);
```

would have led to the same value of `kmean`.

5.1.6 Multi Linear Combination

```
template<class iterator, class neighborhood>
double
multi_linear_combination(iterator begin_weights, iterator end_weights,
                         const neigborhood** first_neighbors,
                         unsigned int nb_of_neighborhoods)
```

Computes the linear combination of the weights in range `[begin_weights, end_weights)` and the property values of the geo-values contained in the neighborhoods in array `first_neighbors`. `first_neighbors` is an array of pointers to neighborhoods, so that `first_neighbors[0]` is a pointer to the first neighborhood. `nb_of_neighborhoods` is the total number of pointers to neighborhoods in the array. Denote $\lambda_1, \ldots, \lambda_P$ the weights, and $z_j(\mathbf{u}_1^j), \ldots, z_j(\mathbf{u}_{n_j}^j)$, $j = 1, \ldots, N_v$ property values of the geo-values contained in the N_v neighborhoods. `linear_combination` returns:

$$\sum_{j=1}^{N_v} \sum_{i=1}^{n_j} \lambda_{\alpha(i,j)}\ z_j(\mathbf{u}_i^j)$$

where $\alpha(i,j) = \sum_{k=1}^{j-1} n_k + i$

Where defined

In header file `<kriging.h>`

Preconditions

P, the number of weights, must be equal or greater than the total number of geo-values in all the neighborhoods combined. The algorithm loops on the geo-values, hence if $P > N$ only the N first weights are used, the others are ignored.

The type of the geo-values properties must be convertible to `double`.

Requirements on types

- `iterator` is a model of Forward Iterator: it is assignable, default constructible, supports operator `++`, operator `*`, and two iterators can be compared with operator `!=`. The dereference type of the iterator must be convertible to `double`.

- `neighborhood` is a model of Neighborhood.

Example

Compute the kriging estimate from the kriging weights. `gauss_covariance` is the Gaussian covariance defined in the example following the description of `kriging_weights`.

```
int main()
{
  location2d u1(2,3);
  location2d u2(4,-7);

  geo_value2d Z1(u1,0.21);
  geo_value2d Z2(u2,0.09);

  Neighborhood neighbors1, neighbors2;
  neighbors1.push_back(Z1);
  neighbors1.push_back(Z2);

  location2d u3(1,2);
  geo_value2d S1(u3,42.1);
  neighbors2.push_back(S1);

  neighborhood* neigh_array[2]={&neighbors1, &neighbors2};

  location2d u(0,0);

  std::vector<double> weights;

  double kvariance = kriging_weights(weights,
                                     u, neigh_array, 2,
                                     LMC_covariance, OK_constraint);

  double kmean = multi_linear_combine(weights.begin(),
                                      weights.end(),
                                      neigh_array, 2);
}
```

5.1.7 Sequential Simulation, Single-Variable Case

1. ```
   template<class gval_iterator, class neighborhood,
            class cdf, class cdf_estimator, class marginal_cdf>
   void
   sequential_simulation(gval_iterator begin, gval_iterator end,
                         neigborhood* neighbors, cdf& ccdf,
                         cdf_estimator& estim, marginal_cdf& marginal)
   ```

2. ```
   template<class gval_iterator, class neighborhood,
            class cdf, class cdf_estimator, class marginal_cdf,
            class sampler>
   void
   sequential_simulation(gval_iterator begin, gval_iterator end,
                         neigborhood* neighbors, cdf& ccdf,
                         cdf_estimator& estim, marginal_cdf& marginal,
                         sampler& samp)
   ```

This function performs a sequential simulation of the range of geo-values delimited by iterators `begin` and `end`. At each location **u** being simulated, the neighborhood of **u** is retrieved and stored into the neighborhood that `neighbors` points to. If no neighbor is found, a new value is simulated from the marginal cumulative distribution `marginal`. Otherwise `estim` estimates a conditional cdf which is stored into `ccdf` and a new value is drawn from `ccdf`.

In version 1, a new value is simulated using Monte-Carlo simulation: a probability is determined randomly and used to draw a realization from conditional cdf `ccdf`.

Version 2 allows to modify the way a simulated value is drawn from the `ccdf`.

Where defined

In header file `<simulation.h>`

Preconditions

- The range `[begin,end)` is a valid range.
- `estim` and `ccdf` do not conflict: if `estim` is designed to estimate Gaussian cdf's, `ccdf` should be a Gaussian cdf.

Requirements on types

- `gval_iterator` is a model of Geo-Value Iterator.
- `neighborhood` is a model of Neighborhood.
- `cdf` is a model of Cdf.
- `marginal_cdf` is a model of Cdf. It can be different from `cdf`.
- `cdf_estimator` is a model of Cdf Estimator.
- `sampler` (in version 2) is a model of Sampler.

Example

A call to

```
// ...

location2d u(0,0);
geo_value2d Z(u,-99);

sequential_simulate(&Z, &Z+1,
                    neighbors_ptr, gauss_cdf,
                    gauss_cdf_estim);

//...
```

would simulate the single geo-value `Z`.

5.1.8 Sequential Simulation, Multiple-Variable Case

1. ```
   template<class gval_iterator, class neighborhood,
            class cdf, class cdf_estimator, class marginal_cdf>
   void
   sequential_cosimulation(gval_iterator begin, gval_iterator end,
                           neigborhood* first_neigh,
                           unsigned int nb_of_neighborhoods,
                           cdf& ccdf, cdf_estimator& estim,
                           marginal_cdf& marginal)
   ```

2. ```
   template<class gval_iterator, class neighborhood,
            class cdf, class cdf_estimator, class marginal_cdf,
            class sampler>
   void
   sequential_cosimulation(gval_iterator begin, gval_iterator end,
                           neigborhood** first_neigh,
                           unsigned int nb_of_neighborhoods,
                           cdf& ccdf, cdf_estimator& estim,
                           marginal_cdf& marginal, sampler& samp)
   ```

This function performs a sequential simulation of the range of geo-values delimited by iterators `begin` and `end`, accounting for multiple variables. At each location **u** being simulated, the conditional cdf is estimated based on the primary information stored in the first neighborhood `*first_neigh[0]`, and the secondary information contained in the other `nb_of_neighborhoods-1` neighborhoods (`first_neigh` is an array of pointers to `neighborhood`s). If at a given location no neighboring data is found, a new value is drawn from the marginal cumulative distribution: `marginal`.

In version 1, a new value is simulated using Monte-Carlo simulation: a probability is determined randomly and used to draw a realization from conditional cdf `ccdf`.

Version 2 allows to modify the way a simulated value is drawn from the `ccdf`.

Where defined

In header file `<simulation.h>`

Preconditions

- The range `[begin,end)` is a valid range.
- `first_neigh[n]` is a pointer to the (n+1)-th neighborhood to be accounted for. (n $<$ `nb_of_neighborhoods`).
- `estim` and `ccdf` do not conflict: if `estim` is designed to estimate Gaussian cdf's, `ccdf` should be a Gaussian cdf.

Requirements on types

- `gval_iterator` is a model of Geo-Value Iterator.
- `neighborhood` is a model of Neighborhood.
- `cdf` is a model of Cdf.
- `marginal_cdf` is a model of Cdf. It can be different from `cdf`.
- `cdf_estimator` is a model of Cdf Estimator.
- `sampler` (in version 2) is a model of Sampler.

5.1.9 P-Field Simulation

1. ```
   template<class gval_iterator, class forward_iterator,
            class neighborhood, class cdf, class cdf_estimator>
   void
   pfield_simulation(gval_iterator begin, gval_iterator end,
                     neigborhood* neighbors,
                     cdf& ccdf, cdf_estimator& estim,
                     forward_iterator pf_begin, forward_iterator pf_end)
   ```

2. ```
   template<class gval_iterator, class forward_iterator,
            class neighborhood, class cdf, class cdf_estimator>
   void
   pfield_simulation(gval_iterator begin, gval_iterator end,
                     neigborhood** first_neighbors,
                     unsigned int nb_of_neighborhoods,
                     cdf& ccdf, cdf_estimator& estim,
                     forward_iterator pf_begin, forward_iterator pf_end)
   ```

This function performs a p-field simulation on geo-values in range `[begin,end)`. For each geo-value, a cdf conditional to only the original data (contrary to sequential simulation where the cdf at each geo-value is conditional to both the original data and the previously simulated values) is estimated by Cdf Estimator `estim`. All the cdf's are then sampled using the correlated probabilities stored in range `[pf_begin, pf_end)` ("pf" stands for p-field). The cdf corresponding to the geo-value that `begin+i` points to, is sampled using the probability that `pf_begin+i` points to. Hence the order in which the geo-values and the p-field values are stored is important.

Version 2 of the algorithm allows to use multiple properties to estimate each cdf.

Where defined

In header file `<simulation.h>`

Preconditions

- Ranges `[begin,end)` and `[pf_begin, pf_end)` are of the same size (i.e. there are as many geo-values as p-field values).
- The values of the p-field (in range `[pf_begin, pf_end)`) are probabilities, i.e. real numbers between 0 and 1.
- `estim` and `ccdf` do not conflict: if `estim` is designed to estimate Gaussian cdf's, `ccdf` should be a Gaussian cdf.
- In version 2 of the function, `first_neigh[n]` is a pointer to the (n+1)-th neighborhood to be accounted for. (n $<$ `nb_of_neighborhoods`).

Requirements on types

- `gval_iterator` is a model of Forward Iterator, which iterates on Geo-Values. It is not a model of GeoValue Iterator.
- `forward_iterator` is a model of Forward Iterator, iterating on floating points values.
- `neighborhood` is a model of Neighborhood. The `find` method of the neighborhood must consider only original data as potential neighbors, and ignore any other geo-value: in p-field simulation, each cdf is only conditional to the original data.
- `cdf` is a model of CDF.
- `cdf_estimator` is a model of CDF Estimator.

Remarks

The cdf corresponding to the geo-value that `begin+i` points to, is sampled using the probability that `pf_begin+i` points to. This means that the two sequences of geo-values and p-field values are tightly linked. Element i of the geo-value range and element i of the p-field form a pair. Swapping elements of either range would completely change the correlation of the simulated field. In particular, `gval_iterator` can not be a random path.

5.2 Basic classes

The descriptions of the models follow the same layout used by Austern (1999):

- The object prototype is stated.
- **Template Parameters** lists what are the template parameters and what kind of concept they model (these could be non-GSTL concepts).
- **Model of** indicates what concept is modeled by the object.
- **Requirement on types** describes possible additional requirements on the template argument types.
- **Members** is a list of all the member function of the object.

5.2.1 Gaussian Cdf

`gaussian_cdf`

`gaussian_cdf` defines a Gaussian cumulative distribution function of a continuous variable Z. The function and its inverse are computed using numerical approximations as proposed by W.J. Kennedy and Gentle (1980) and Abramowitz and Stegun (1965). In both cases the approximation error is lesser than 0.0001.

Where Defined

In header file `<cdf.h>`

Model of

Cdf

Members

- **`gaussian_cdf::value_type`**

  The type of the variable Z. It is set to be `double`.

- **`gaussian_cdf::gaussian_cdf()`**

  Creates a standard Gaussian cdf (mean 0 and variance 1).

- **`gaussian_cdf::gaussian_cdf(double m, double var)`**

  Creates a Gaussian cdf of mean `m` and variance `var`.

- **`double& gaussian_cdf::mean()`**

  Returns the mean of the cdf.

- **`const double& gaussian_cdf::mean() const`**

  Returns the mean of the cdf.

- **`double& gaussian_cdf::variance()`**

  Returns the variance of the cdf.

- **`const double& gaussian_cdf::variance() const`**

  Returns the variance of the cdf.

- **`double gaussian_cdf::prob(value_type z)`**

  Returns the probability $Prob(Z \leq z)$.

- **`value_type gaussian_cdf::inverse(double p)`**

  Returns the value z such that $p = Prob(Z \leq z)$.

5.2.2 Non-Parametric Cdf, continuous variable

`non_param_cdf<lower_tail_interpol,middle_interpol,upper_tail,T>`

A non-parametric cdf of variable Z is a cdf F defined by a discrete set of points $\Big(z_i, F(z_i)\Big)$, $i = 1, \ldots, n$, $z_1 \leq \ldots \leq z_n$. As Z is a continuous variable, the points $\Big(z_i, F(z_i)\Big)$ must be interpolated in order to associate a probability to any z-value different from the z_i's. Call z an outcome of Z different from z_i (for all i). If $z < z_1$ or $z > z_n$, a function of type `lower_tail_interpol` or `upper_tail_interpol` is used to interpolate the cdf and compute $F(z)$. If $z_1 \leq z \leq z_n$ a function of type `middle_interpol` is used. Similarly, when computing the inverse of the cdf for a probability p, if $p < p_1$ or $p > p_n$, a function of type `lower_tail_interpol` or

`upper_tail_interpol` is used to interpolate the cdf and compute $F^{-1}(p)$. If $p_1 \leq p \leq p_n$ a function of type `middle_interpol` is used.

Where Defined

In header file `<cdf.h>`

Template Parameters

`lower_tail_interpol`	the type of the function used to interpolate the lower tail of the distribution
`upper_tail_interpol`	the type of the function used to interpolate the upper tail of the distribution
`middle_interpol`	the type of the function used to interpolate between two known values z_j and z_{j+1}
`T`	the cdf's type value. It is `double` by default.

Model of

Non-Parametric Cdf

Type Requirements

- `lower_tail_interpol` and `upper_tail_interpol`: an object that models these concepts must have two member functions:

  1. `double p(value_type z1, double p1, value_type z)`

     returns the interpolated value of p given the point (z1,p1) and new value z.

  2. `value_type z(value_type z1, double p1, double p)`

     returns the interpolated value of z given the point (z1,p1) and new value p.

- `middle_interpol` : an object that models these concepts must have two member functions:

  1. `double p(value_type z1, double p1, value_type z2, double p2, value_type z)`

     returns the interpolated value of p given the point (z1,p1) and new value z.

  2. `double z(value_type z1, double p1, value_type z2, double p2, double p)`

     returns the interpolated value of z given the point (z1,p1) and new value p.

Members

- **`non_param_cdf::value_type`**

  The type of variable Z. It is set to be `double` by default.

- **`non_param_cdf::z_iterator`**

  An iterator to the sequence of values $z_1 \leq \ldots \leq z_n$. It is a model of Forward Iterator.

- **`non_param_cdf::p_iterator`**

  An iterator to the sequence of values $p_1 \leq \ldots \leq p_n$. It is a model of Forward Iterator.

- `non_param_cdf::non_param_cdf()`

  Default constructor.

- `non_param_cdf::non_param_cdf(z_iterator z_begin, z_iterator z_end)`

  Creates a non-parametric cdf. The z-values $z_1, \ldots, z_n$ are read from range `[z_begin,z_end)`. The corresponding probabilities are not initialized. The range `[z_begin,z_end)` must be sorted, in increasing order.

- `non_param_cdf::non_param_cdf(z_iterator z_begin, z_iterator z_end, p_iterator p_begin)`

  Creates a non-parametric cdf.

  The z-values $z_1, \ldots, z_n$ are read from range `[z_begin,z_end)`, and the corresponding probabilities are read starting from `p_begin`. The range `[z_begin,z_end)` must be sorted, in increasing order.

- `void non_param_cdf::resize(unsigned int m)`

  Allocates space for a discretization of size m.

- `void non_param_cdf::z_set(z_iterator z_begin, z_iterator z_end)`

  Redefines the discretization $z_1, \ldots, z_n$ to the values in range `[z_begin,z_end)`. The range `[z_begin,z_end)` must be sorted, in increasing order.

  The new discretization can contain more values than the previous one. The probability values corresponding to the former discretization are invalidated.

- `z_iterator non_param_cdf::z_begin()`

  Returns a model of Forward Iterator to the first element of the discretization $z_1, \ldots, z_n$.

- `z_iterator non_param_cdf::z_end()`

  Returns a model of Forward Iterator to the end of the discretization $z_1, \ldots, z_n$.

- `p_iterator non_param_cdf::p_begin()`

  Returns a model of Forward Iterator to the first element of the set of probabilities $p_1, \ldots, p_n$.

- `p_iterator non_param_cdf::p_end()`

  Returns a model of Forward Iterator to the end of the set of probabilities $p_1, \ldots, p_n$.

- **`double non_param_cdf::prob(value_type z)`**

  Returns the probability $Prob(Z \leq z)$.

- **`value_type non_param_cdf::inverse(double p)`**

  Returns the value z such that $p = Prob(Z \leq z)$.

5.2.3 Non-Parametric Cdf, categorical variable

`categ_non_param_cdf<T>`

`categ_non_param_cdf<T>` is a non-parametric cdf of a categorical (discrete) variable Z. It is defined by n category labels $z_1, \ldots, z_n$ and the corresponding probabilities. We define the cumulative probability of being in class z_j by:

$$Prob(Z \leq z_j) = \sum_{i=1}^{j-1} Prob(Z = z_i)$$

Where Defined

In header file `<cdf.h>`

Template Parameters

`T`	the cdf's type value. It can be any "discrete" type (e.g. `int` or `bool`). It is `unsigned int` by default.

Model of

Non-Parametric Cdf

Members

The leading `categ_non_param_cdf::` is omitted in the following list of member fonctions.

- **`categ_non_param_cdf::value_type`**

  The type of variable Z. It is set to be `unsigned int` by default.

- **`non_param_cdf::z_iterator`**

  An iterator to the sequence of values $z_1 \leq \ldots \leq z_n$. It is a model of Forward Iterator.

- **`non_param_cdf::p_iterator`**

  An iterator to the sequence of values $p_1 \leq \ldots \leq p_n$. It is a model of Forward Iterator.

- `categ_non_param_cdf()`

  Default constructor.

- `categ_non_param_cdf(z_iterator z_begin, z_iterator z_end)`

  Creates a non-parametric cdf.

  `z_iterator` is a model of Forward Iterator. The z-values $z_1, \ldots, z_n$ are read from range `[z_begin,z_end)`. The corresponding probabilities are not initialized.

- `categ_non_param_cdf(z_iterator z_begin, z_iterator z_end,`
  `p_iterator p_begin)`

Creates a non-parametric cdf.

`z_iterator` and `p_iterator` are models of Forward Iterator. The z-values $z_1, \ldots, z_n$ are read from range `[z_begin,z_end)`, and the corresponding probabilities are read starting from `p_begin`. Since the probabilities are cumulative probabilities, the (cumulative) probability associated to the last class is necessarily equal to 1.

- `void non_param_cdf::resize(unsigned int m)`

  Allocates space for a discretization of size m.

- `void z_set(z_iterator z_begin, z_iterator z_end)`

  Redefines the labels $z_1, \ldots, z_n$ to the labels in range `[z_begin,z_end)`.

  `z_iterator` is a model of Forward Iterator. The new set of labels can contain more values than the previous one. The probability values corresponding to the former labels are invalidated.

- `z_iterator categ_non_param_cdf::z_begin()`

  Returns a model of Forward Iterator to the first element of set $z_1, \ldots, z_n$.

- `z_iterator categ_non_param_cdf::z_end()`

  Returns a model of Forward Iterator to the end of set $z_1, \ldots, z_n$.

- `p_iterator non_param_cdf::p_begin()`

  Returns a model of Forward Iterator to the first element of the set of probabilities $p_1, \ldots, p_n$.

- `p_iterator non_param_cdf::p_end()`

  Returns a model of Forward Iterator to the end of the set of probabilities $p_1, \ldots, p_n$.

- **`double categ_non_param_cdf::prob(value_type z)`**

  Returns the probability $Prob(Z \leq z)$.

- **`value_type categ_non_param_cdf::inverse(double p)`**

  Returns the value z such that $p = Prob(Z \leq z)$.

5.3 Function Object Classes

5.3.1 Random Sampler

`random_sampler<random_number_generator>`

`random_sampler` randomly draws a value z from a cdf F: a random probabilty p is generated, and $z = F^{-1}(p)$. By default, probability p is generated by `drand48`, a random number generator of *stdlib.h* that uses the linear congruential algorithm and 48-bit integer arithmetic.

Where Defined

In header file `<sampler.h>`

Model of

Sampler

Type Requirements

`random_number_generator` is a function object that takes no argument and returns a `double` between 0 and 1. Its constructor must take a `long int` as argument to seed the pseudo-random number sequence.

Members

- `random_sampler::random_sampler()`

  Default constructor.

- `random_sampler::random_sampler(long int seed)`

  Constructor. Initializes the random number generator with `seed`.

- ```
  template<class cdf>
  typename cdf::value_type
  operator()(cdf f)
  ```

  Function call operator. Type `cdf` is a model of Cdf. Draws a value from `f`.

5.3.2 Simple Kriging Constraints

`SK_constraints`

In simple kriging, the error variance is minimized without any additional constraint. The kriging system is then:

$$\left\{ \; Var\Big(\sum_{\alpha=1}^{n} \lambda_\alpha [Z(\mathbf{u}_\alpha) - \mathrm{m}(\mathbf{u}_\alpha)] - [\mathrm{Z}(\mathbf{u}) - \mathrm{m}(\mathbf{u})] \Big) \quad \text{is minimum} \right.$$
  ```

or, if secondary variables are accounted for:

$$\begin{cases} Var\Big(\sum_{\alpha=1}^{n} \lambda_\alpha [Z(\mathbf{u}_\alpha) - \mathrm{m}(\mathbf{u}_\alpha)] + \sum_{\mathrm{i}=1}^{\mathrm{N_v}} \sum_{\beta=1}^{\mathrm{n_i(u)}} \lambda_\beta^{\mathrm{i}} [\mathrm{Z}(\mathbf{u}_\beta^{\mathrm{i}}) - \mathrm{m}(\mathbf{u}_\beta^{\mathrm{i}})] \\ \qquad -[Z(\mathbf{u}) - \mathrm{m}] \Big) \quad \text{is minimum} \end{cases}$$

`SK_constraints` computes the kriging system size, resizes the kriging matrix and the second member, and returns the system size.

Where Defined

In header file `<kriging.h>`

Model of

Kriging Constraints

Type Requirements

See 5.4 for a thorough description of the requirements on the matrix library.

Members

- `SK_constraints::SK_constraints()`

  Default constructor.

- ```
  template<class neighborhood, class location, class Matrix, class Vector>
  unsigned int
  SK_constraints::operator()(Matrix& A, Vector& b,
                             const location& u
                             Neighborhood** first_neigh,
                             unsigned int nb_of_neighborhoods)
  ```
  ```

Function call operator. `neighborhood` is a model of Neighborhood, and `location` is a model of Location. `A` is the kriging matrix and `b` the second member of the kriging system. `u` is the location being estimated. The function returns the total number of neighbors, i.e. the sum of the number of neighbors in each neighborhoods. The requirements on concepts Matrix and Vector are fully described in 5.4. `first_neigh` is an array of pointers to `neighborhood`s.

5.3.3 Ordinary Kriging Constraints

`OK_constraints`

In ordinary kriging, the error variance is minimized with the constraint that the kriging weights sum-up to 1. The kriging system is then:

$$\begin{cases} Var\Big(\sum_{\alpha=1}^{n} \lambda_\alpha [Z(\mathbf{u}_\alpha) - \mathrm{m}(\mathbf{u}_\alpha)] - [\mathrm{Z}(\mathbf{u}) - \mathrm{m}(\mathbf{u})]\Big) \quad \text{is minimum} \\ \\ \sum_{\alpha=1}^{n} \lambda_\alpha = 1 \end{cases}$$

or, if secondary variables are accounted for:

$$\begin{cases} Var\Big(\sum_{\alpha=1}^{n} \lambda_\alpha [Z(\mathbf{u}_\alpha) - \mathrm{m}(\mathbf{u}_\alpha)] + \sum_{\mathrm{i}=1}^{\mathrm{N_v}} \sum_{\beta=1}^{\mathrm{n_i(u)}} \lambda_\beta^{\mathrm{i}} [\mathrm{Z}(\mathbf{u}_\beta^{\mathrm{i}}) - \mathrm{m}(\mathbf{u}_\beta^{\mathrm{i}})] \\ \qquad -[Z(\mathbf{u}) - \mathrm{m}]\Big) \quad \text{is minimum} \\ \\ \sum_{\alpha=1}^{n} \lambda_\alpha = 1 \\ \sum_{\beta=1}^{n_i(\mathbf{u})} \lambda_\beta^i = 0 \quad i = 1, \ldots, N_v \end{cases}$$

`OK_constraints` computes the kriging system size, resizes the kriging matrix and

the second member, computes the terms of the system that are associated with the constraint on the kriging weights and finally returns the system size.

Where Defined

In header file `<kriging.h>`

Model of

Kriging Constraints

Type Requirements

See 5.4 for a thorough description of the requirements on the matrix library.

Members

- `OK_constraints::OK_constraints()`

  Default constructor.

- ```
  template<class neighborhood, class location, class Matrix, class Vector>
  unsigned int
  OK_constraints::operator()(Matrix& A, Vector& b,
                             const location& u,
                             Neighborhood** first_neigh,
                             unsigned int nb_of_neighborhoods)
  ```

  Function call operator. `neighborhood` is a model of Neighborhood, and `location` is a model of Location. `A` is the kriging matrix and `b` the second member of the
  ```

kriging system. `u` is the location being estimated. The function returns the total number of neighbors, i.e. the sum of the number of neighbors in each neighborhoods. The requirements on concepts Matrix and Vector are fully described in 5.4. `first_neigh` is an array of pointers to `neighborhood`s.

5.3.4 Kriging with Trend Constraints

`KT_constraints<forward_iterator>`

`KT_constraints` adds constraints to account for the variations of the mean of the krigged variable Z. The mean is assumed to be of the form:

$$m(\mathbf{u}) = \sum_{\mathrm{k}=0}^{\mathrm{K}} \mathrm{a_k}(\mathbf{u})\mathrm{f_k}(\mathbf{u})$$

where a_k are unknown but locally constant and f_k are known functions of **u**.

The kriging system at location **u** is then given by:

$$\begin{cases} Var\Big(\sum_{\alpha=1}^{n} \lambda_\alpha [Z(\mathbf{u}_\alpha) - \mathrm{m}(\mathbf{u}_\alpha)] - [\mathrm{Z}(\mathbf{u}) - \mathrm{m}(\mathbf{u})]\Big) \quad \text{is minimum} \\ \sum_{\alpha=1}^{n} \lambda_\alpha = 1 \\ \sum_{\alpha=1}^{n} \lambda_\alpha(\mathbf{u})\mathrm{f_k}(\mathbf{u}_\alpha) = \mathrm{f_k}(\mathbf{u}) \quad \forall \mathrm{k} \in [1, \mathrm{K}] \end{cases}$$

Kriging with trend is rarely used with secondary variables. Hence `KT_constraints` assumes no secondary variable is to be accounted for. `KT_constraints` computes the kriging system size, resizes the kriging matrix and the second member, computes the terms of the system that are associated with the constraints on the kriging weights and finally returns the system size.

Where Defined

In header file `<kriging.h>`

Template Parameters

`forward_iterator`	is a model of Forward Iterator.

Model of

Kriging Constraints

Type Requirements

See 5.4 for a thorough description of the requirements on the matrix library.

Members

- `KT_constraints(forward_iterator begin, forward_iterator end)`

  Constructs a `KT_constraint`. The range `[begin,end)` contains the functions $f_i, i = 1, \ldots, K$ that define the mean of Z. These functions must be unary functions and associate to a location a value of type convertible to `double`. The prototype of each function f_i must be:

  ```
  double f(location u)
  ```

  where `location` is a model of Location.

- `template<class neighborhood, class location, class Matrix, class Vector>`
  `unsigned int`
  `KT_constraints::operator()(Matrix& A, Vector& b,`
  `const location& u,`
  `Neighborhood** first_neigh,`
  `unsigned int nb_of_neighborhoods=1)`

  Function call operator. `neighborhood` is a model of Neighborhood, and `location` is a model of Location. `A` is the kriging matrix and `b` the second member of the kriging system. `u` is the location being kriged. The function returns the total number of neighbors, i.e. the sum of the number of neighbors in each neighborhoods. The requirements on concepts Matrix and Vector are fully described in 5.4. `first_neigh` is an array of pointers to `neighborhood`s.

5.3.5 LMC Covariance

`LMC_covariance<covariance_matrix>`

Cokriging of variable Z_1, accounting for secondary variables $Z_2, \ldots, Z_{N_v}$, requires the covariances $(\mathbf{u}_1, \mathbf{u}_2) \longmapsto \mathrm{C_{i,j}}(\mathbf{u}_1, \mathbf{u}_2)$.

LMC cokriging requires the knowledge of the covariances between any two locations $\mathbf{u}_1$, $\mathbf{u}_2$.

Where Defined

In header file `<kriging.h>`

Template Parameters

`covariance_matrix`	is an object that has member function `covariance_matrix[int i][int j]`, which returns element (i,j) of the matrix. The elements of the matrix must be models of Covariance

Model of

Covariance Set

Type Requirements

The elements of the matrix must be models of Covariance

Members

- `LMC_covariance::LMC_covariance(const covariance_matrix& A, unsigned int size)`

  Constructs a `LMC_covariance`. Matrix `A` contains pointers to the covariance functions $(\mathbf{u}_1, \mathbf{u}_2) \longmapsto \mathrm{C_{i,j}}(\mathbf{u}_1, \mathbf{u}_2)$. `size` is the size of the covariance matrix. It is equal to the number of variables N_v.

- `double operator()(unsigned int i, unsigned int j, const location& u1, const location& u2)`

  Function call operator. Returns the covariance $C_{i,j}(\mathbf{u}_1, \mathbf{u}_2)$.

5.3.6 MM1 Covariance

`MM1_covariance<covariance, matrix>`

Full cokriging of Z_1 accounting for secondary variables $Z_2, \ldots, Z_{N_v}$ requires the inference of all covariance functions $(\mathbf{u}_1, \mathbf{u}_2) \longmapsto \mathrm{C_{i,j}}(\mathbf{u}_1, \mathbf{u}_2)$ between variables i and j. This very difficult task can be eased by considering only the colocated secondary variables. The underlying hypotheses is that the colocated value screens out the influence of further away data. In this situation, only the covariances $C_{1,j}$ need be inferred. The MM1 approximation alleviate the modeling effort further with the following approximation:

$$C_{1,j}(\mathbf{u}_1, \mathbf{u}_2) = \frac{\mathrm{C_{1,j}}(0)}{\mathrm{C_{1,1}}(0)} \mathrm{C_{1,1}}(\mathbf{u}_1, \mathbf{u}_2)$$

where $C_{i,j}(0) = C_{i,j}(\mathbf{u}_1, \mathbf{u}_1) = \mathrm{C_{i,j}}(\mathbf{u}_2, \mathbf{u}_2)$

This approximation is acceptable if the support of the secondary variables is not larger than the support of the primary variable Z_1. For example, if Z_1 is rock porosity and Z_2 rock permeability, MM1 approximation is acceptable. It would not be if Z_2 were seismic amplitude, because seismic amplitude is generally defined on a much larger scale than porosity.

Where Defined

In header file `<kriging.h>`

Template Parameters

`covariance`	is a model of Covariance
`matrix`	is an type such that if `A` is of type `matrix`, `A[i][j]` is a valid expression (`i` and `j` are two integers), which returns element (i,j) of A.

Model of

Covariance Set

Type Requirements

The elements of the matrix are of type convertible to `double`.

Members

- `MM1_covariance::MM1_covariance(const covariance& cov, const matrix& A, unsigned int size)`

  Constructs a `MM1_covariance`. Covariance function `cov` is $C_{1,1}$, and matrix `A` contains the covariance values $C_{i,j}(0)$. Notice that matrix `A` contains numbers, not pointers to functions as in LMC. `size` is the size of the covariance matrix. It is equal to the number of variables N_v.

- `double operator()(unsigned int i, unsigned int j, const location& u1, const location& u2)`

  Function call operator. Returns the covariance $C_{i,j}(\mathbf{u}_1, \mathbf{u}_2)$ using the MM1 approximation.

5.3.7 MM2 Covariance

`MM2_covariance<covariance_vector, matrix>`

The MM1 approximation is not valid if the support of the secondary variables is larger than the support of the primary variable. In this case, the covariances $C_{1,j}$ can be approximated as follows (MM2 hypothesis):

$$C_{1,j}(\mathbf{u}_1, \mathbf{u}_2) = \frac{\mathrm{C_{1,j}}(0)}{\mathrm{C_{1,1}}(0)} \mathrm{C_{j,j}}(\mathbf{u}_1, \mathbf{u}_2)$$

This approximation is less convenient than the MM1 approximation, because it requires the inference of all covariances $C_{j,j}$.

Where Defined

In header file `<kriging.h>`

Template Parameters

`covariance_vector`	is an object that has member function `covariance_vector[int i]`, that returns element `i` of the vector. The elements of the vector must be pointers to models of Covariance
`matrix`	is an object that has member function `matrix[int i][int j]`, that returns element (i,j) of the matrix.

Model of

Covariance Set

Type Requirements

- The elements of the matrix are of type convertible to `double`.
- `covariance_vector` contains pointers to models of Covariance.

Members

- `MM2_covariance::MM2_covariance(covariance_vector& cov_vect, const matrix& A, unsigned int size)`

  Constructs a `MM2_covariance`. `cov_vect` contains pointers to the covariance functions $C_{i,i}$, $i = 1, \ldots, N_v$.

  Matrix `A` contains the covariance values $C_{i,j}(0)$. Notice that matrix `A` contains numbers, not pointers to functions as in LMC.

  `size` is the size of the covariance matrix. It is equal to the number of variables N_v.

- `double operator()(unsigned int i, unsigned int j, const location& u1, const location& u2)`

  Function call operator. Returns the covariance $C_{i,j}(\mathbf{u}_1, \mathbf{u}_2)$ using the MM1 approximation.

5.3.8 Kriging-Based, Gaussian Cdf Estimator

`gaussian_cdf_Kestimator<covariance,kriging_constraints,matrix_lib>`

This cdf estimator assumes the cdf of variable Z is Gaussian, and does not account for any secondary variable. The mean and variance of the Gaussian cdf are estimated by kriging.

Where Defined

In header file `<cdf_estimators.h>`

Template Parameters

`covariance`	is a model of Covariance
`kriging_constraints`	is model of Kriging Constraints. It is set by default to `OK_constraints<tnt_lib>`
`matrix_lib`	defines the library of linear algebra to be used. The default value is `tnt_lib`, the *TNT* library.

Model of

Single Variable Cdf Estimator

Members

The leading `gaussian_cdf_Kestimator::` is omitted in the list of member functions.

- `gaussian_cdf_Kestimator(const covariance& cov,`
  `const kriging_constraints& Kconstraints)`

  Constructs a `gaussian_cdf_Kestimator`. It requires the covariance function: $Cov\Big(Z(\mathbf{u}), \mathrm{Z}(\mathbf{u}+\mathbf{h})\Big)$, and a set of kriging constraints (e.g. simple kriging constraints).

- `template<class location, class neighborhood, class gaussian_cdf>`
  `void operator()(const location& u, const neighborhood& neighbors,`
  `gaussian_cdf& ccdf)`

  Function call operator. It estimates the gaussian cdf parameters and modifies `ccdf` accordingly. `location` is a model of Location, and `neighborhood` is a model of Neighborhood.

  `u` is the location at which the Gaussian conditional cdf is estimated.

  `neighbors` is the neighborhood of location `u`.

5.3.9 Cokriging-Based, Gaussian Cdf Estimator

```
gaussian_cdf_coKestimator<covariance_set,
                          kriging_constraints,matrix_lib>
```

This cdf estimator assumes the cdf of variable Z is Gaussian, and does account for secondary variables. The mean and variance of the Gaussian cdf are estimated by cokriging.

Where Defined

In header file `<cdf_estimators.h>`

Template Parameters

`covariance_set`	is a model of Covariance Set
`kriging_constraints`	is model of Kriging Constraints. It is set by default to `OK_constraints<tnt_lib>`
`matrix_lib`	defines the library of linear algebra to be used. The default value is `tnt_lib`, the *TNT* library.

Model of

Multiple Variable Cdf Estimator

Members

The leading `gaussian_cdf_coKestimator::` is omitted in the list of member functions.

- ```
  gaussian_cdf_coKestimator(const covariance_set& cov_set,
                            const kriging_constraints& Kconstraints)
  ```

  Constructs a `gaussian_cdf_coKestimator`. Covariance set `cov_set` gives the covariances $C_{i,j}(\mathbf{u}_\mathrm{i}, \mathbf{u}_\mathrm{j})$ between variables $Z(\mathbf{u}_\mathrm{i})$ and $Z(\mathbf{u}_\mathrm{j})$.

- ```
  template<class location, class neighborhood, class gaussian_cdf>
  void operator()(const location& u, neighborhood** neighbors,
                  unsigned int nb_of_neighborhoods,
                  gaussian_cdf& ccdf)
  ```

  Function call operator. It estimates the gaussian cdf parameters and modifies `ccdf` accordingly. `location` is a model of Location, and `neighborhood` is a model of Neighborhood.

`u` is the location at which the Gaussian conditional cdf is estimated.

`neighbors` is an array of pointers to neighborhoods of location `u` (`*(neighbors+i)` points to the i^{th} neighborhood of `u`, informing variable Z_i).

There are `nb_of_neighborhoods` neighborhoods in the array.

5.3.10 Indicator Cdf Estimator

```
indicator_cdf_estimator<covar_iterator,
                        constraints_iterator,matrix_lib>
```

Indicator kriging estimates a non-parametric cdf $\Big(z_i, F(z_i)\Big)$, $i = 1, \ldots, n$ of variable Z by kriging n indicator variables $I(\mathbf{u}, \mathrm{z_i})$:

$$i(\mathbf{u}, \mathrm{z_i}) = \begin{cases} 1 & \text{if } z(\mathbf{u}) \leq \mathrm{z_i} \\ 0 & \text{otherwise} \end{cases}$$

The kriging estimate of $I(\mathbf{u}, \mathrm{z_i})$ is indeed the least-squares estimate of $Prob(Z(\mathbf{u}) \leq \mathrm{z_i})$ (see section 3.2).

Each indicator variable can be estimated using a different kriging method, e.g. with different kriging constraints. This cdf estimator only allows to change the kriging constraints, and none of the kriging systems can account for multiple variables (no cokriging).

It must be stressed that an Indicator Cdf Estimator expects n indicator variables $I(\mathbf{u}, \mathrm{z_i})$, not the single variable $Z(\mathbf{u})$ itself.

Where Defined

In header file `<cdf_estimators.h>`

Template Parameters

`covar_iterator`	is a model of Forward Iterator
`constraints_iterator`	is model of Forward Iterator
`matrix_lib`	defines the library of linear algebra to be used. The default value is `tnt_lib`, the *TNT* library.

Model of

Multiple Variable Cdf Estimator

Type Requirements

- `covar_iterator` iterates on a set of pointers to covariance functions, i.e. pointers to objects that are models Covariance.
- `constraints_iterator` iterates on a set of pointers to kriging constraints, i.e. pointers to objects that are models of Kriging Constraints.

Members

The leading `indicator_cdf_estimator::` is omitted in the list of member functions.

- ```
  indicator_cdf_estimator(covar_iterator cov_begin,
                          covar_iterator cov_end,
                          constraints_iterator begin,
                          constraints_iterator end)
  ```

  Constructs an `indicator_cdf_estimator`. Ranges of iterators `[cov_begin,cov_end)` and `[begin,end)` need not be of the same size, nor do
  ```

they need to be of size n, the number of discretizations of the non-parametric cdf. If they are of size lesser than n, the last element of the range is used for kriging the remaining indicators. For example, if $n = 5$ and both `[cov_begin,cov_end)` and `[begin,end)` contain only two elements, the first indicator variable $I(\mathbf{u}, \mathrm{z}_1)$ will be kriged using covariance `*cov_begin` and kriging constraints `*begin`, while all four remaining indicator variables $I(\mathbf{u}, \mathrm{z}_\mathrm{i})$, $i = 1, \ldots, 4$ will be kriged using the same covariance and the kriging constraints `*(cov_begin+1)` and `*(begin+1)`.

- ```
  template<class location, class neighborhood, class non_parametric_cdf>
  void operator()(const location& u, neighborhood** neighbors,
                  unsigned int nb_of_neighborhoods,
                  non_parametric_cdf& ccdf)
  ```

  Function call operator. It estimates the non-parametric cdf parameters $F(z_i)$ and modifies `ccdf` accordingly. `location` is a model of Location, and `neighborhood` is a model of Neighborhood.

  `u` is the location at which the non-parametric conditional cdf is estimated.

  `neighbors` is an array of pointers to neighborhoods of location `u` (`*(neighbors+i)` points to the i^{th} neighborhood of `u`). There are `nb_of_neighborhoods` neighborhoods in the array. Neighborhood i informs variable $I(\mathbf{u}, \mathrm{z}_\mathrm{i})$.

  `ccdf` must contain the values z_i, $i = 1, \ldots, n$.

5.3.11 Search Tree

`search_tree<neighborhood>`
  ```

Consider a random variable $Z(\mathbf{u})$ which can take K different values $Z_1, \ldots, Z_K$. The aim of a Search Tree is to infer the ccdf of $Z(\mathbf{u})$ from a known realization of $Z(\mathbf{u})$: $z(\mathbf{u}_{\alpha_1}), \ldots, \mathrm{z}(\mathbf{u}_{\alpha_\mathrm{N}})$, called "training image".

Call $T = \{\mathbf{h_1}, \ldots, \mathbf{h_t}\}$ the family of vectors defining a geometric template (or "window") of t locations. These vectors are also called *nodes* of the template. A data event implied by T, at location **u**, is the sequence of values:

$$D_T(\mathbf{u}) = \{\mathrm{Z}(\mathbf{u}+\mathbf{h_1}), \ldots, \mathrm{Z}(\mathbf{u}+\mathbf{h_t})\}$$

u is called the central node of the template. Provided the training image is stationary, the same data event (i.e. the same sequence of values) can be observed at different locations.

Guardiano and Srivastava (1993), and Strebelle (2000) proposed to model the probability

$$P\Big(Z(\mathbf{u}) = \mathrm{z_k} \mid \{\mathrm{z}(\mathbf{u}+\mathbf{h_1}), \ldots, \mathrm{z}(\mathbf{u}+\mathbf{h_t})\}\Big)$$

by the frequency of occurrence in the training image of event

$$z(\mathbf{u}_\alpha) = \mathrm{z_k} \mid \{\mathrm{z}(\mathbf{u}_\alpha+\mathbf{h_1}), \ldots, \mathrm{z}(\mathbf{u}_\alpha+\mathbf{h_t})\}$$

($\mathbf{u}_\alpha$ is a location of the training image): if for a given data event $\mathrm{d_T}$, there are n locations $\mathbf{u_i}$ in the training image such that:

$$D_T(\mathbf{u_i}) = \mathrm{d_T} \quad \mathrm{i} = 1, \ldots, \mathrm{n}$$

and among these n locations, n_k are such that the central pixel value $z(\mathbf{u_j}) = \mathrm{z_k}$ ($j = 1, \ldots, n_k$), then the probability $P\Big(Z(\mathbf{u}) = \mathrm{z_k} \mid \mathrm{d_T}\Big)$ is modeled by

$$P\Big(Z(\mathbf{u}) = \mathrm{z_k} \mid \mathrm{d_T}\Big) = \frac{\mathrm{n_k}}{\mathrm{n}}$$

In some cases, data event d_T can not be found in the training image. Call T_{-1} the subset of T obtained by dropping one of the vectors (nodes) of T, and similarly, T_{-j} the subset of T after dropping j vectors of T, for any $j \in \{0, \ldots, t-1\}$, with $T_{-0} = T$. If data event d_T can not be found in the training image, template T is recursively simplified into T_{-1}, $\ldots$,T_{-j}, until $d_{T_{-j}}$ can be found. Typically, the nodes are dropped according to the amount of information they bring in estimating the probability distribution of $Z(\mathbf{u}) = \mathrm{z_k}$. The probability $P\Big(Z(\mathbf{u}) = \mathrm{z_k} \mid \mathrm{d_T}\Big)$ is then approximated by

$$P\Big(Z(\mathbf{u}) = \mathrm{z_k} \mid \mathrm{d_T}\Big) \simeq \mathrm{P}\Big(\mathrm{Z}(\mathbf{u}) = \mathrm{z_k} \mid \mathrm{d_{T_{-j}}}\Big)$$

A search tree is a data structure that enables to store all the data events $d_{T_{-j}}$ $(j = 0, \ldots, t-1)$ present in the training image, along with the corresponding frequencies of occurrence of z_k $(k = 1, \ldots, K)$ at the central node.

Where Defined

In header file `<cdf_estimators.h>`

Model of

Single Variable Cdf Estimator

Members

- `template<class forward_iterator>`
  `search_tree::search_tree(forward_iterator begin, forward_iterator end`
  `neighborhood& neighbors)`

  Constructs a `search_tree`.

`forward_iterator` is a model of Forward Iterator. It iterates on a set of geo-values, the training image.

`neighborhood` is a model of Neighborhood. It is used to define the data event associated to each geo-value in range `[begin,end)`. It is usually a window-neighborhood.

- ```
  template<class location, class non_param_cdf>
  void search_tree::operator()(const location& u,
                               const neighborhood& neighbors,
                               non_param_cdf& ccdf)
  ```

  Function call operator. It estimates the non-parametric cdf parameters and modifies `ccdf` accordingly. `location` is a model of Location, and `neighborhood` is a model of Neighborhood.

  `u` is the location at which the Gaussian conditional cdf is estimated.

  `neighbors` is neighborhood of location `u`. It must be the same object (or different object with the same characteristics) as the one used in the search tree constructor. The order in which the neighbors are stored inside the neighborhood is important. Denote $\mathbf{u} + \mathbf{h_i}$, $i = 1, \ldots, N$ the locations of the N neighbors of $\mathbf{u}$. The algorithm assumes that the i^{th} geo-value in the neighborhood is $Z(\mathbf{u} + \mathbf{h_i})$.

5.4 Changing the Linear Algebra Library

Kriging, which is at the root of many geostatistical algorithms requires basic linear algebra facilities, essentially matrix inversion. Most of the computing time in kriging is actually
  ```

spent building and solving the kriging system, hence the importance of using an efficient linear algebra library.

The default library used by GSTL is a slightly modified version of *TNT*, the *Template Numerical Toolkit*. It is a public domain library written by Roldan Pozo, Mathematical and Computational Sciences Division, National Institute of Standards and Technology (it can be freely downloaded from `http://math.nist.gov/tnt/`). This library was chosen because it is relatively efficient, it is easy to use and modify, and it is public domain.

However, it is easy to change the linear algebra library used by the GSTL algorithms without having to modify existing code. The procedure is twofold. The first step is to wrap all the needed functionalities of the library into a single structure, call it `new_matrix_lib`. This structure will contain nested types (a matrix type, a vector type, ...) and static functions (for example `static void inverse(matrix& A)`), which must honor the requirements detailed in section 5.4.1.

The following is an extract from the *TNT* wrapper:

```
template<class T>
struct tnt_lib{

  typedef TNT::Subscript Subscript;
  typedef TNT::Matrix<T> tnt_Matrix;
  typedef TNT::Vector<T> tnt_Vector;

  // Cholesky factorization.
  static inline int cholesky(TNT::Matrix<T>& A, TNT::Matrix<T>& B){
    return Cholesky_upper_factorization(A,B);
  }

  // LU factorization.
  static inline int LU_factor(TNT::Matrix<T>& A, TNT::Vector<int>& index){
    return TNT::LU_factor(A,index);
  }

  static inline TNT::Matrix<T> transpose(TNT::Matrix<T>& A){
    return TNT::transpose(A);
  }
```

It has three nested types: `Subscript`, `Matrix`, and `Vector` which are defined by *TNT*, and three functions: a Cholesky factorization, a LU factorization, and a transpose function, which simply call existing *TNT* functions.

The second step is to specialize the matrix library trait class for the new library wrapper `new_matrix_lib`. The trait class is defined in `<matrix_lib_traits.h>` (for a

detailed introduction to the idea of trait classes, see [Myers, 1995]). What has been done in the case of *TNT* can serve as a model.

5.4.1 Linear Algebra Library Requirements

Matrix

The matrix type represents a matrix of `double` and must have the following interface:

- `Matrix::Matrix()`

  Default constructor.

- `Matrix::Matrix(int m, int n)`

  Creates a $m * n$ matrix.

- `double Matrix::operator()(subscript i, subscript j)`

  returns element (i,j) of the matrix. The indices have offset 1: the first element is (1,1), not (0,0). `subscript` is a type convertible to `int`.

- `int Matrix::num_rows()`

  returns the number of rows of the matrix

- `int Matrix::num_cols()`

  returns the number of columns of the matrix

- `void Matrix::resize(int n, int m)`

  Resizes the matrix to size $n * m$.

Vector

Vector type is a vector of `double` which has the following interface:

- `Vector::Vector()`

  Default constructor.

- `Vector::Vector(int m)`

  Creates a vector of size m.

- `double Vector::operator()(subscript i)`

  returns element i of the vector. The index has offset 1: the first element is number 1, not 0. `subscript` is a type convertible to `int`.

- `void Vector::resize(int n)`

  Resizes the vector to size n.

LU solve

The prototype of the function must be:

```
template< class random_iterator>
int LU_solve(Matrix& A, Vector& b, iterator solution_begin)
```

This function solves linear system $Ax = b$ using a LU decomposition of A, and stores the resulting vector x in the container `solution_begin` points to. This container must be of size equal to the size of vector b. The iterator must be a Random Access Iterator. A Random Access Iterator is a refinement of Forward Iterator. If `it` is a Random Access Iterator, `it[j]` is a valid expression which makes iterator `it` point to the j-th element of the container.

Cholesky solve

The prototype of the function must be:

```
template< class random_iterator>
int Cholesky_solve(Matrix& A, Vector& b, iterator solution_begin)
```

This function solves linear system $Ax = b$ using a Cholesky decomposition of A (A must be positive-definite), and stores the resulting vector x in the container `solution_begin` points to. This container must be of size equal to the size of vector b.

Section 6

Two Example Applications

The algorithms described in chapter 5 are not numerous compared to the great number of geostatistical algorithms one can find in the literature. This does not mean the algorithms implemented in GsTL were limited to a selected few. The aim of the library design was indeed to obtain generic algorithms that capture the commonalities of the various existing geostatistical algorithms and capitalize on those to provide a generic and extendable implementation.

Two examples of the genericness of the algorithms are presented. The first one details how to use GsTL to implement a kriging algorithm that accounts for block average constraints. The notion of scale was never mentioned in GsTL algorithms, yet it is possible to use GsTL to constraint kriging to block data, i.e. data that inform a larger support than a single point.

The second example shows how the kriging algorithm of GsTL could be integrated into an existing software, *gOcad*, to estimate grids of complex geometry.

6.1 Kriging constrained to a block average value

The traditional kriging algorithm estimates variable Z at location **u** from data of the same support: the data are either related to points in space, or supports of identical volumes. However, it is sometimes necessary to account for information at other scales.

Consider the case where a location **u** in space region V(**u**) has to be estimated. The average property value $z_{V(\mathbf{u})}$

$$z_{V(\mathbf{u})} = \frac{1}{|V(\mathbf{u})|} \int_{V(\mathbf{u})} z(\mathbf{u}') \mathrm{d}\mathbf{u}'$$

is known, along with some values $z(\mathbf{u}_\alpha)$, $\alpha = 1, \ldots, n$ in V(**u**). The aim is to estimate $z(\mathbf{u})$ from both the data $z(\mathbf{u}_\alpha)$ and the average value $z_{V(\mathbf{u})}$. If all the locations in V(**u**) are estimated, their average must then be equal to $z_{V(\mathbf{u})}$. This situation occurs in down-scaling applications.

The kriging estimate is of the form:

$$z(\mathbf{u}) = \sum_{\alpha=1}^{\mathrm{n}} \lambda_\alpha \mathrm{z}(\mathbf{u}_\alpha) + \lambda_{\mathrm{V}(\mathbf{u})} \mathrm{z}_{\mathrm{V}(\mathbf{u})}$$

The kriging system is then:

$$\begin{pmatrix} C(\mathbf{u}_1, \mathbf{u}_1) & \ldots & C(\mathbf{u}_1, \mathbf{u}_\mathrm{n}) & \bar{C}(\mathbf{u}_1, \mathrm{V}) \\ \vdots & \ddots & \vdots & \vdots \\ C(\mathbf{u}_\mathrm{n}, \mathbf{u}_1) & \ldots & C(\mathbf{u}_\mathrm{n}, \mathbf{u}_\mathrm{n}) & \bar{C}(\mathbf{u}_\mathrm{n}, \mathrm{V}) \\ \bar{C}(\mathbf{u}_1, \mathrm{V}) & \ldots & \bar{C}(\mathbf{u}_\mathrm{n}, \mathrm{V}) & \bar{C}(V, V) \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_n \\ \lambda_{V(\mathbf{u})} \end{pmatrix} = \begin{pmatrix} C(\mathbf{u}, \mathbf{u}_1) \\ \vdots \\ C(\mathbf{u}, \mathbf{u}_\mathrm{n}) \\ C(\mathbf{u}, \mathrm{V}) \end{pmatrix}$$

where $C(\mathbf{u}_\mathrm{i}, \mathbf{u}_\mathrm{j})$ is the average covariance between $Z(\mathbf{u}_\mathrm{i})$ and $Z(\mathbf{u}_\mathrm{j})$, $\bar{C}(\mathbf{u}_\mathrm{i}, \mathrm{V})$ is the covariance between $Z(\mathbf{u}_\mathrm{i})$ and $Z_{V(\mathbf{u})}$, and $\bar{C}(V, V)$ is the covariance between $Z_{V(\mathbf{u})}$ and $Z_{V(\mathbf{u})}$.

These last two covariances are defined by:

$$\bar{C}(Z(\mathbf{u}_\alpha), \mathrm{Z_V}) = \frac{1}{|\mathrm{V}|} \int_{\mathrm{V}(\mathbf{u})} \mathrm{C}\Big(\mathrm{Z}(\mathbf{u}_\alpha), \mathrm{Z}(\mathbf{u}')\Big) \mathrm{d}\mathbf{u}'$$

and

$$\bar{C}(Z_V, Z_V) = \frac{1}{|V|^2} \int_{V(\mathbf{u})} \int_{V(\mathbf{u})} C\Big(Z(\mathbf{u}), \mathrm{Z}(\mathbf{u}')\Big) \mathrm{d}\mathbf{u} \mathrm{d}\mathbf{u}'$$

This kriging system is in fact a cokriging system, with primary variable Z(**u**) and secondary variable $Z_{V(\mathbf{u})}$.

Hence, this kriging technique could be implemented easily using algorithms `cokriging_weights` and `multi_linear_combination` (described page 75 and 81).

Implementation

The implementations proposed hereafter are kept very simple. The aim is not to provide a generic and efficient implementation, but rather to illustrate the use of GsTL algorithms with some basic objects.

First, the covariance functions $C(\mathbf{u}_\mathrm{i}, \mathbf{u}_\mathrm{j})$, $\bar{C}(\mathbf{u}_\mathrm{i}, \mathrm{V})$ and $\bar{C}(V, V)$ must be defined.

The implementation of $C(\mathbf{u}_\mathrm{i}, \mathbf{u}_\mathrm{j})$ could be, for a gaussian covariance of fixed range:

```
class Czz{
  public:
    inline double gauss_covariance(Location2d u1, Location2d u2){
      double square_dist = pow(u1[0]-u2[0], 2) +
                           pow(u1[1]-u2[1], 2);
      return exp(-3*square_dist/16);
    }
```

Type `location2d` is defined at the beginning of part 5.1.

Assume type `neighborhood` is a model of Neighborhood, covariance function $\bar{C}(\mathbf{u}_\mathrm{i}, \mathrm{V})$ can be implemented as:

```
typedef neighborhood block;

class Czblock{
  private:
    Czz& point_cov_;
    block& B_;

  public:
    Czblock(block& B) : B_(B) {};
    double operator()(location2d u1, location2d u2){
      B_.find_neighbors(u2);
      double covar = 0;
      for(block::iterator it =B_.begin(); it!=B_.end(); it++)
        covar += point_cov_(u1, *it);

      covar = covar / double(B_.size());

      return covar;
    }
};
```

Notice that block V(**u**) is implemented as a neighborhood. It is indeed a set of geo-values, "centered" on a location **u**, which corresponds to the definition of Neighborhood.

Finally $\bar{C}(\mathbf{u}_\mathrm{i}, \mathrm{V})$ could be implemented as:

```
class Cblockblock{
  private:
    Czz& point_cov_;
    block& B1_;

  public:
    Czblock(block& B1) : B1_(B1) {};
    double operator()(location2d u1, location2d u2){
      B1_.find_neighbors(u2);
      block B2_ = B1_;
      double covar = 0;
      for(block::iterator it_B1 =B1_.begin(); it_B1 != B1_.end(); it_B1++)
        for(block::iterator it_B2 = B2_.begin(); it_B2 != B2_.end(); it_B2++)
        covar += point_cov_(*it_B1, *it_B2);

      covar = covar / double(B_.size()*B_.size());

      return covar;
    }
};
```

A model of Covariance Set can then be defined using these three covariance functions:

```
class Block_covariance_set{
  private:
    Czz&         point_cov;
    Czblock&     point_block_cov;
    Cblockblock& block_block_cov;

  public:
    Block_covariance_set(Czblock& C2, Cblockblock& C3) :
       point_cov(C1), point_block_cov(C2), block_block_cov(C3) {};

    double operator()(int i, int j, location2d u1, location2d u2){
      switch(i){
        case 0:
          if(j==0) return point_cov(u1,u2);
          else return point_block_cov(u1,u2);
          break;
        case 1:
          if(j==0) return point_block_cov(u1,u2);
          else return block_block_cov(u1,u2);
          break;
      }
    }
};
```

A function call to `cokriging_weights` would then compute the kriging weights $\lambda_1, \ldots, \lambda_n, \lambda_V$:

```
cokriging_weights(weights, center, neighborhood_array, 2,
                  covariance_set, ordinary_krig)
```

where `covariance_set` is of type `Block_covariance_set`, `ordinary_krig` is of type `OK_constraints`, and `neighborhood_array` contains a neighborhood of z-values $z(\mathbf{u}_\alpha)$, and another neighborhood of block-values Z_V (only one block value in this example).

These weights can then be combined with the variable values $z(\mathbf{u}_1), \ldots, \mathrm{z}(\mathbf{u}_\mathrm{n}), \mathrm{z}_{\mathrm{V}(\mathbf{u})}$ to obtain the kriging estimate.

6.2 Integration into an existing software: kriging complex geometries in *gOcad*[1]

In GSTL, The geostatistical algorithms that work with grids of geo-values do not rely on a specific type of grid. Applying such algorithms to different types of grids, which were possibly implemented outside the GSTL framework, is therefore straightforward. The main step is to check that the already existing objects meet the requirements of the generic algorithms. If they do not, "wrapper" classes have to be implemented, which modify the former behavior of the object to make it compliant with the GSTL requirements.

This is an example of implementation of a model of Neighborhood:

[1] **The figures and computer code in this section were realized with the help of Arben Stuka, *gOcad*, France.**

```
class neighborhood{
  public:
    typedef std::vector<GTLNode>::iterator iterator;
  public:
    neighborhood() {}
    neighborhood(const neighborhood& ng) : neigh_(ng.neigh_) {}
    neighborhood(TSurf* ts, int u_idx, int v_idx, int p_idx );
    ~neighborhood(){}

    void add_node(GTLNode n){
      neigh_.push_back(n);
    }
    iterator begin(){
      return neigh_.begin();
    }
    iterator end(){
      return neigh_.end();
    }
    size_t size(){
      return neigh_.size();
    }
    void find_neighbors(point3d u);

  private:
    std::vector<GTLNode> neigh_;
};
```

The neighborhood constructor is defined as follows:

```
neighborhood::neighborhood(
    TSurf* ts, int u_idx, int v_idx, int p_idx
) {
    for ( AtomicGroupAtomsItr it(*ts); it.more(); it.next()) {
        Atom* a = it.cur();
        if (a->CN()) {
            Atom& ra = *a;
            float u = ra[u_idx];
            float v = ra[v_idx];
            double prop = ra[p_idx];
            GTLNode gnode(u, v, prop);
            add_node(gnode);
        }
    }
}
```

`AtomicGroupAtomsItr` is the *gOcad* class that represents the working grid.

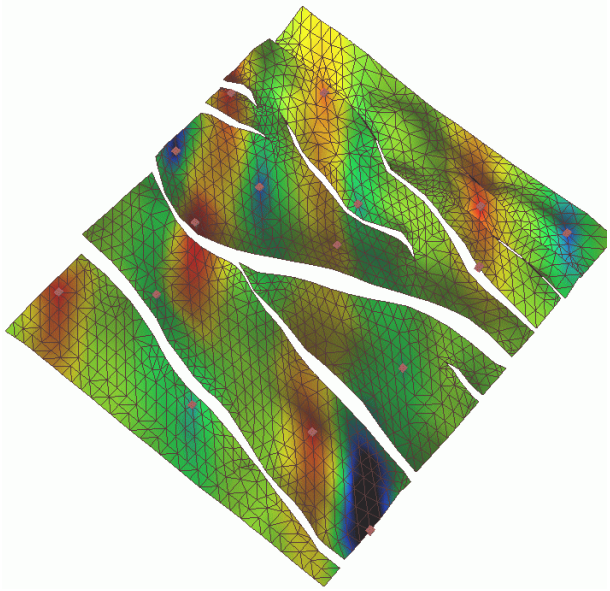
Ordinary kriging is performed on a *gOcad* triangulated faulted surface using the GSTL. The kriging uses a global neighborhood (all the data are accounted for at every kriged location), and the variogram had a strong anisotropy. Two snapshots of the result are shown in Figure 6.1

This same GSTL algorithm could also be used to estimate a *gOcad* T-solid, i.e. an unstructured grid with polyhedra cells. Two snapshots of the resulting grid are shown on Figure 6.2. Recall that to obtain both results in Figure 6.1 and Figure 6.2 no change i smade to the GSTL kriging algorithm.

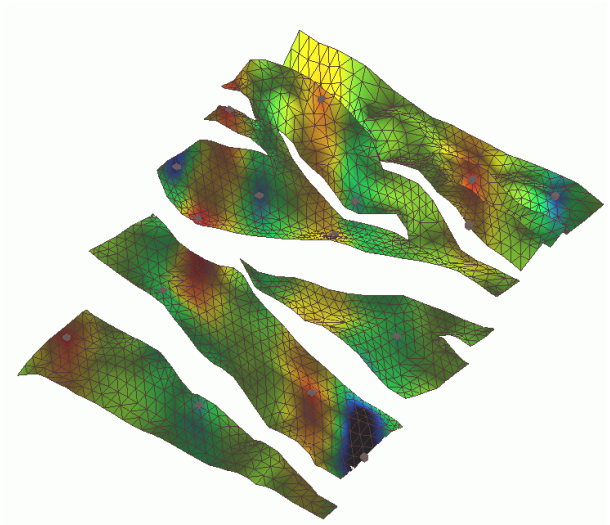
In both cases, the property is continuous across the faults. This assumes that the fault

appeared after the genesis of the rock. However, it could have been possible to make the property discontinuous across the faults by modifying the way the neighbors of each location are retrieved: If no neighbors are sought across a fault, the property would have been continuous between two faults, but discontinuous across the faults.

Working directly on these complex grids hence allows to incorporate some important geometrical features into the model, which was not feasible with the tradition approach. In the traditional approach, the properties are simulated or estimated on a Cartesian grid and then transported to a complex grid. Such a methodology does not allow to account for gometrical constraints like faults.

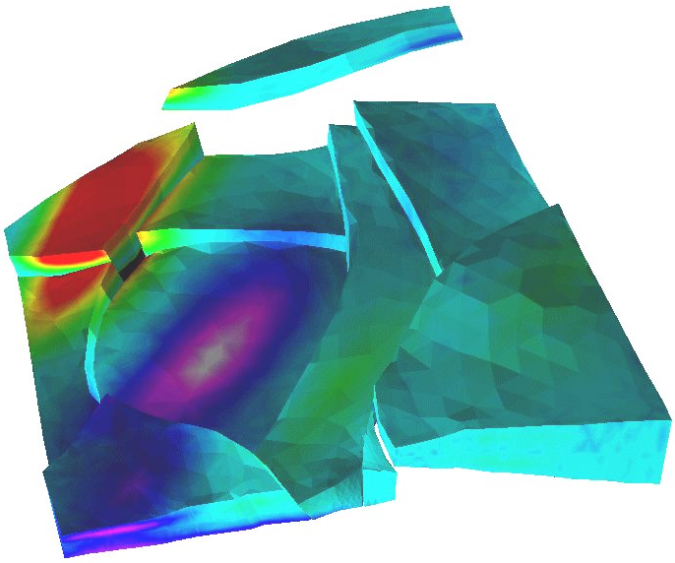


(a) View 1

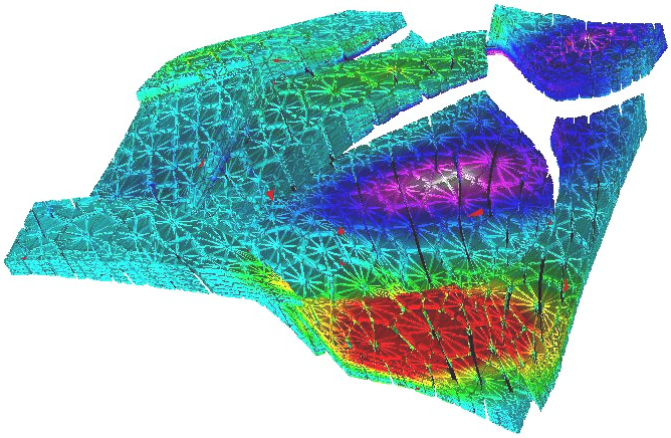


(b) View 2

Figure 6.1: Kriging on a triangulated faulted surface



(a) View 1



(b) structure of the “T-solid”

Figure 6.2: Kriging on a T-solid: an unstructured grid with polyhedra cells

Section 7

Conclusion

GsTL is a C++ library of geostatistical algorithms. It has three major components: the source code of the geostatistical algorithms, the detailed description of the requirements on the concepts used by the algorithms, and a collection of ready-to-use models of the concepts, i.e. actual C++ objects.

Contrary to the two other components, the description of the concepts is not C++ code. It is a mere textual description of the assumptions made by the GsTL algorithms, yet it is an essential part of the library. These descriptions are the analogue of the hypotheses of a mathematical theorem: the statement of a theorem has little value if the hypotheses are omitted.

This similarity with mathematical theorems makes the use of the generic algorithms intuitive. The procedure is indeed the same as when one wants to call a theorem: first check that the hypotheses are verified, and then apply the theorem.

This is much more intuitive than the object-oriented approach, which requires the library user to have a detailed understanding of the class hierarchies before being able to efficiently use the library

The GsTL code is compliant with the ISO/ANSI C++ standard. It is uniquely composed of header files and does not require to be pre-compiled.

It must be stressed that GsTL is a library of programming components, not a collection of softwares. Its aim is to provide tools for quickly building new geostatistics algorithms, sparing from the need to re-invent the wheel each time a kriging routine is needed.

An extension of this work would then be to implement a set of geostatistical softwares, in the style of *GSLIB* [Deutsch and Journel, 1992], based on GsTL. Programming this "library" of softwares would be the opportunity to cash in on the *GSLIB* experience and propose a more convenient interface. This includes better file formats for input and output and possibly a graphical user interface.

GSLIB parameter files are indeed assumed to have a static structure: parameter X is expected at line j. A more convenient approach would be to use keywords to specify what parameter is passed. The data file format could also be modified to at least include essential information as, for example, grid dimensions.

Bibliography

Abramowitz and Stegun: 1965, *Handbook of Mathematical Functions*, Dover Publications.

Austern, M. H.: 1999, *Generic Programming and the STL*, Addison-Wesley Professional Computing Series.

Barton, J. J. and Nackman, L. R.: 1994, *Scientific and engineering C++*, Addison Wesley.

Caers, J. and Journel, A.: 1998, Stochastic reservoir simulation using neural networks trained on outcrop data. SPE paper # 49026.

Deutsch, C.: 1992, *Annealing techniques applied to reservoir modeling and the integration of geological and engineering (well test) data*, PhD thesis, Stanford University, Stanford, CA.

Deutsch, C. and Journel, A.: 1992, *GSLIB: Geostatistical Software Library and User's Guide*, Oxford University Press, New York.

Geman, S. and Geman, D.: 1984, Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **PAMI-6**(6), 721–741.

Goovaerts, P.: 1997, *Geostatistics for natural resources evaluation*, Oxford University Press, New York.

Guardiano, F. and Srivastava, R. M.: 1993, Multivariate geostatistics: beyond bivariate moments, *in* A. Soares (ed.), *Geostatistics-Troia*, Vol. 1, Kluwer Academic Publ., Dordrecht, pp. 133–144.

Isaaks, E.: 1990, *The Application of Monte Carlo Methods to the Analysis of Spatially Correlated Data*, PhD thesis, Stanford University, Stanford, CA.

Journel, A.: 1989, *Fundamentals of Geostatistics in Five Lessons*, Volume 8 Short Course in Geology, American Geophysical Union, Washington, D.C.

Krige, D. G.: 1951, *A statistical approach to some mine valuations and allied problems at the witwatersrand*, Master's thesis, University of Witwatersrand, South Africa.

Luenberger, D.: 1969, *Optimization by Vector Space Methods*, John Wiley & Sons, New York.

Myers, N.: 1995, Traits: A new and useful template technique, *C++ Report* **PAMI-6**.

Ripley, B.: 1987, *Stochastic Simulation*, John Wiley & Sons, New York.

Strebelle, S.: 2000, *Sequential simulation drawing structures from training images*, PhD thesis, Stanford University, Stanford, CA.

W.J. Kennedy, J. and Gentle, J. E.: 1980, *Statistical Computing*.