# GsTL: the Geostatistical Template Library in C++

Nicolas Remy[1], Arben Shtuka[2], Bruno Levy[3], Jef Caers[1]

[1]Department of Petroleum Engineering
Stanford University,
Stanford, CA 94305-2115

[2]Ecole Nationale Supérieure de Géologie - INPL
Nancy, France

[3]ISA-INRIA Lorraine
Nancy, France

June 5, 2001

**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.

# 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. *Generic Programming*, a more recent and probably less known programming paradigm, was indeed prefered to object-oriented programming because it offers many interesting possibilities.

# 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 $\mathbf{u}$ 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 specific to 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 knows how to compare them

The algorithm would then be implemented as follows:

[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.

```
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`:

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

```
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:

```
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 any 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 will be the last part of the library design.

# 3 Overview of the Main Algorithms of Geostatistics

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 $\{z(\mathbf{u}_\alpha)\}$:

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

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:

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

Substituting $Z^*(\mathbf{u})$ in (2) by its expression (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 (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 $(\lambda_\alpha)$ 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 $(\lambda_\alpha)$ such that:

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

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 $\mathbf{u}$. The kriging system at location $\mathbf{u}$ is then given by:

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

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

$$\begin{aligned} Z^*(\mathbf{u}) - \mathrm{m}(\mathbf{u}) \;\; = \;\; & \sum_{\alpha=1}^{n} \lambda_\alpha [Z(\mathbf{u}_\alpha) - \mathrm{m}(\mathbf{u}_\alpha)] \\ & + \sum_{i=1}^{n_v} \sum_{\alpha_1=1}^{n_i} \lambda_{\alpha_i} [S_i(\mathbf{u}_{\alpha_i}) - \mathrm{m_i}(\mathbf{u}_{\alpha_i})] \end{aligned} \tag{3}$$

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 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 $\mathbf{u}$ being estimated are accounted for. These data will be referred to as the *neighborhood* of $\mathbf{u}$. 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:

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

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 $\lambda_m$ 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 of [Remy, 2001] details how kriging can be constrained to a block average value by using the GsTLcokriging algorithm.

## 3.2 Simulation

The aim of simulation is to find a function

$$\left\{\begin{array}{ccc} \mathbb{U} & \longrightarrow & \mathbb{E}^N \\ (\mathbf{u}_i)_{1\leq i\leq N} & \longmapsto & \left(z(\mathbf{u_i})\right)_{1\leq i\leq N} \end{array}\right.$$

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 location-dependent 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 simulated 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 location-dependent 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 $V(\mathbf{u}_i)$:

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

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

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

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 $F\Big(\mathbf{u}_{\mathrm{i}}, \mathrm{Z} \mid (\mathrm{n}_{\mathbf{u}_{\mathrm{i}}})\Big)$ 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.

# 4 Concepts and Algorithms

From the previous overview 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. It is the interface between the algorithm and the grid of geo-values.
- 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. 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:

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

- 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 (in SNESIM, the cdf is read from a table whose entries are the neighborhood geometry and the neighboring data values) or built the cdf from estimated values, using kriging for example, 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 [Remy, 2001].

Building on these key concepts, the following algorithms are implemented:

- Cdf Transform: transforms a set of values so that their final cumulative distribution function is a given target cdf.

- Kriging Weights: computes the kriging weights at a given location. The same algorithm allows to perform simple kriging, ordinary kriging or kriging with trend.

- cokriging: computes the cokriging weights at a given location. The same algorithm allows to perform simple kriging or ordinary kriging, in each case using the full cokriging system, hence requiring all the covariances and cross-covariances between all the variables, or using the MM1 or MM2 hypotheses to reduce the number of cross-covariances to be inferred.

- sequential simulation: this algorithm allows to sequentially simulate a variable on a set of locations. Depending on the *Cdf Estimator* used, the algorithm can perform sequential Gaussian simulation, sequential indicator simulation or single normal equation simulation (multiple-point statistics-based algorithm).

- p-field simulation

These algorithms are fully documented in [Remy, 2001].

# 5 Application: kriging complex geometries in *gOcad*

In order to illustrate the genericness of the library, GsTL algorithms are applied on grids implemented outside the GsTL framework.

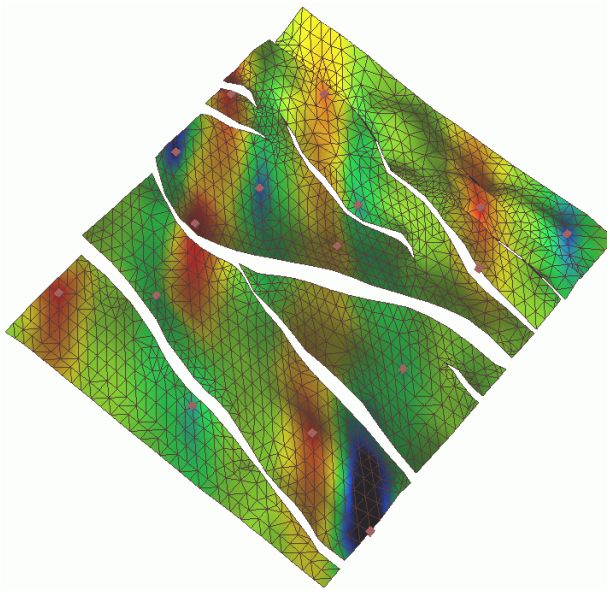
In GsTL, The geostatistical algorithms that work with grids of geo-values do not rely on a specific type of grid. Applying such algorithms to different types of grids, which were possibly implemented outside the GsTL framework, is therefore straightforward. The main step is to check that the already 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.

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 1

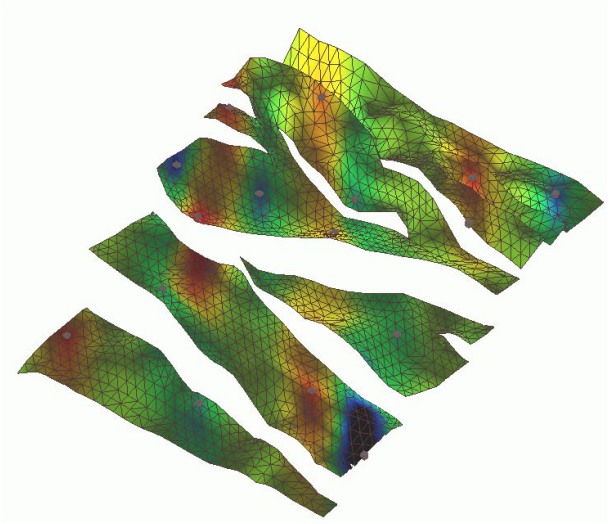
This same GsTL algorithm could also be used to estimate a *gOcad* T-solid, i.e. an unstructured grid with polyhedra cells. Two snapshots of the resulting grid are shown on Figure 2. Recall that to obtain both results in Figure 1 and Figure 2 no change is made to the GsTL kriging algorithm.

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 geometrical constraints like faults.

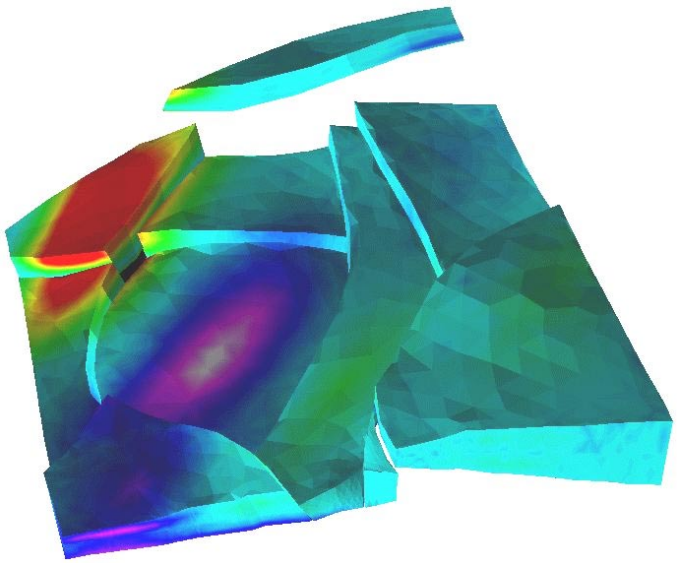


(a) View 1

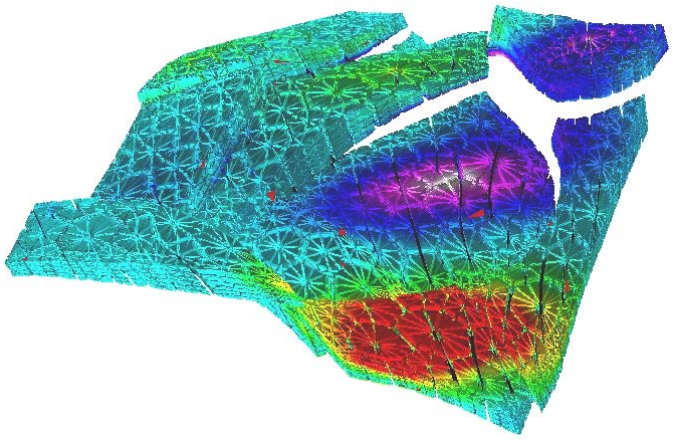


(b) View 2

Figure 1: Kriging on a triangulated faulted surface



(a) View 1



(b) structure of the “T-solid”

Figure 2: Kriging on a T-solid: an unstructured grid with polyhedra cells

# 6 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 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.

# References


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.

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.

Remy, N.: 2001, *$G_s$TL: The geostatistical template library in C++*, Master's thesis, Stanford University.

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.