Chapter 1

Concepts Definitions

This is the thorough definition the concepts used by $\mathrm{G_STL}$ algorithms.

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.

1.1 Basic concepts

1.1.1 Assignable

This is concept is defined in the STL (see M. Austern, Generic Programming and the STL, Addison Wesley Longman, 1999) **??**.

Notations

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

Valid Expressions

- **Copy Constructor**

  `A(a)`

Return type:	`A`
Semantics:	`A(a)` is a copy of `a`

- **Copy Constructor**

  `A a(b)`

Return type:	`A`
Semantics:	`a` is a copy of `b`

- **Assignment**

  `a=b`

Return type:	`A&`
Semantics:	`b` is assigned to `a`

1.1.2 Default Constructible

This is concept is defined in the STL (see M. Austern, Generic Programming and the STL, Addison Wesley Longman, 1999) **??**.

Notations

A	A type that is a model of Default Constructible
a	objects of type A

Valid Expressions

- **Default Constructor**

  `A()`

Return type:	`A`

- **Default Constructor**

  `A a()`

Return type:	`A`

1.1.3 Equality Comparable

This is concept is defined in the STL (see M. Austern, Generic Programming and the STL, Addison Wesley Longman, 1999) **??**.

Notations

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

Valid Expressions

- **Equality**

  `a == b`

Return type:	a type that is convertible to bool
Semantics:	is true if `a` is equal to `b`

- **Inequality**

  `a != b`

Return type:	a type that is convertible to bool
Semantics:	same as `!(a==b)`

1.1.4 Forward Iterator

Refinement of

Assignable (see Section1.1.1), Equality Comparable (see Section1.1.3)

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:	`I&`
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

- **Difference**

  `i-j`

Return type:	`int`
Semantics:	returns the size of range delimited by `i` and `j`
Postcondition:	`i` is dereferenceable or past the end

- **dereference**

  `*i`

Return type:	`value_type`
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.

1.1.5 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.

- **const Iterator type**

  `A::const_iterator`

  An iterator that points to the Container's elements. This iterator can not be used to modify the value it points to.

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` if the container is mutable (i.e. can be written to), or `const_iterator` if the container is read-only.
Semantics:	returns an iterator pointing to the first element of the Container

- **End of range**

  `a.end()`

Return type:	`iterator` if the container is mutable, otherwise `const_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

1.1.6 Euclidean Vector

An euclidean vector is represented by its Cartesian coordinates: $(p_0, p_1, \ldots, p_{d-1})$ (in a d-dimensional vectorial space). It can be defined by the difference between two points (Locations), A and B: if A has coordinates $(a_0, a_1, \ldots, a_{d-1})$ and B $(b_0, b_1, \ldots, b_{d-1})$, then vector $\mathbf{AB} = B{-}A$ has coordinates $(b_0 - a_0, b_1 - a_1, \ldots, b_{d-1} - a_{d-1})$.

The requirements of Euclidean Vector are very similar to those of Location, and could actually have been represented by a Location. However, an euclidean vector is very different from a point, or location, (from a mathematical point of view) and it would have been confusing to represent these two entities by the same *concept*.

Refinement of

Default Constructible (see Section1.1.2), Assignable (see Section1.1.1), Equality Comparable (see Section1.1.3)

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 type `double`

Notations

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

Valid Expressions

- **Dimension**

  `A::dimension`

Return type:	`int`
Semantics:	returns the dimension of vectors of type `A`

- **Copy Constructor**

  `A(a)`

Return type:	A
Postcondition:	`A(a)` is a copy of `a`

- **Copy Constructor**

  `A b(a)`

Postcondition:	`b` is a copy of `a`

- **Assignment**

  `b = a`

Return type:	`A&`
Postcondition:	`b` is a copy of `a`

- **Coordinate Access Function**

  `a[i]`

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

- **Equality Comparison**

  `a == b`

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

Models

- `euclidean_vector_2d`
- `euclidean_vector_3d`

1.1.7 Location

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

Refinement of

Default Constructible (see Section1.1.2), Assignable (see Section1.1.1), Equality Comparable (see Section1.1.3)

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 type `double`.

- **Difference type**

  `A::difference_type`

  The type of the euclidean vector obtained by computing the difference between two Locations. `difference_type` is a model of Euclidean Vector (see Section 1.1.6).

Notations

L	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

- **Dimension**

  `L::dimension`

Return type:	`int`
Semantics:	returns the dimension of locations of type `L`

- **Copy Constructor**

  `L(a)`

Return type:	L
Postcondition:	`L(a)` is a copy of `a`

- **Copy Constructor**

  `L b(a)`

Postcondition:	`b` is a copy of `a`

- **Assignment**

  `b = a`

Return type:	`L&`
Postcondition:	`b` is a copy of `a`

- **Coordinate Access Function**

  `a[i]`

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

- **Equality Comparison**

  `a == b`

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

- **Difference Operator**

  `a - b`

Return type:	a model of EuclideanVector
Semantics:	Computes the euclidean vector between two locations `a` and `b`

Models

- `location2d`
- `location3d`

1.1.8 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. The property value of a Geo-Value can be changed, however its location is fixed.

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 (see Section1.1.7).

Notations

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

Valid Expressions

- **Read property value**

  `a.property_value()`

Return type:	P
Semantics:	return the property value hold by the Geo-Value (read-only function).

- **Write property value**

  `a.set_property_value(p)`

Return type:	void
Semantics:	Sets the property value hold by the Geo-Value to p.

- **Get Location**

  `a.location()`

Return type:	L
Semantics:	returns the location of the Geo-Value

- **Is informed**

  `a.is_informed()`

Return type:	`bool`
Semantics:	returns true if the Geo-Value contains an actual property value.

Models

- `node_2d`
- `node_3d`

1.1.9 Neighborhood

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

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

$\mathbf{u}$ 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:

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

Refinement of

Container (see Section1.1.5)

Associated Types

- **Geovalue type**

  `A::value_type`

  `value_type` is the type of geovalue stored into the neighborhood.

- **Neighborhood Iterator**

  `A::iterator`

  A neighborhood iterator is a model of Forward Iterator (see previous description). 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 A
I	A type the is a model of Forward Iterator (see Section1.1.4)
u	An object of a type that models Geovalue (see Section1.1.8)

Valid Expressions

- **find neighbors**

  `a.find_neighbors(u)`

Return type:	`void`
Semantics:	finds the neighbors of geovalue u and stores them

- **center**

  `a.center()`

Return type:	GeoValue
Semantics:	Returns the center of the neighborhood

- **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.is_empty()`

Return type:	a type that is convertible to bool
Semantics:	returns "true" if the size of the neighborhood is 0

Models

- `elliptical_neighborhood`

1.1.10 Window Neighborhood

Call $\mathbf{h}_1, \ldots, \mathbf{h}_\mathrm{k}$ a set of vectors in space. A Window Neighborhood of $\mathbf{u}$ is the set of k geovalues $z(\mathbf{u}+\mathbf{h}_1), \ldots, \mathrm{z}(\mathbf{u}+\mathbf{h}_\mathrm{k})$ ($z(\mathbf{u}+\mathbf{h}_\mathrm{i})$ is the geovalue whose location is $\mathbf{u}+\mathbf{h}_\mathrm{i}$ and property value $z(\mathbf{u}+\mathbf{h}_\mathrm{i})$).

The order of the elements of a Window Neighborhood is important: the i^{th} element must be $z(\mathbf{u}+\mathbf{h}_\mathrm{i})$. If $z(\mathbf{u}+\mathbf{h}_\mathrm{i})$ is not defined (either because this location has not been estimated or simulated yet, or because it does not belong to the simulation grid), it must be included into the Window Neighborhood nevertheless. However, if a non-defined geovalue is the last element of a Window Neighborhood, it is not included. This is a major difference with Neighborhood (see Section 1.1.9): a Neighborhood (see Section 1.1.9) only contains informed geovalues, i.e. geovalues with a defined property value.

For example, consider a Window Neighborhood W defined by six vectors $\mathbf{h}_1, \ldots, \mathbf{h}_6$, and a location $\mathbf{u}$ such that $z(\mathbf{u}+\mathbf{h}_2)$, $z(\mathbf{u}+\mathbf{h}_5)$ and $z(\mathbf{u}+\mathbf{h}_6)$ are not defined. Then W must only contain: $z(\mathbf{u}+\mathbf{h}_1)$, $z(\mathbf{u}+\mathbf{h}_2)$, $z(\mathbf{u}+\mathbf{h}_3)$, $z(\mathbf{u}+\mathbf{h}_4)$. A Neighborhood (see Section 1.1.9) would have contained only $z(\mathbf{u}+\mathbf{h}_1)$, $z(\mathbf{u}+\mathbf{h}_3)$ and $z(\mathbf{u}+\mathbf{h}_4)$

Refinement of

Neighborhood (see Section 1.1.9)

Associated Types

Same as Neighborhood (see Section 1.1.9)

Notations

a	Object of a type that is a model of Window Neighborhood
begin, end	Iterators on a container of models of Euclidean Vector (see Section 1.1.6)

Valid Expressions

- **set geometry**

  ```
  a.set_geometry(begin, end)
  ```

Return type:	`void`
Semantics:	defines the geometry $\mathbf{h}_1, \ldots, \mathbf{h}_k$ of the window. The range of Euclidean Vectors need not be copied, i.e. `a` does not be the owner of the geometry.

1.1.11 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 CDF1.1.11
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`

1.1.12 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). A z-iterator is a model of Random-Access Iterator (as defined by STL).

- **Z-iterator**

  `A::p_iterator`

  The type of the iterator to the values $p_1, \ldots, z_n$ (see definition above). A p-iterator is a model of Random-Access Iterator (as defined by STL).

Refinement of

CDF1.1.11, Default Constructitble1.1.2

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 $p_1, \ldots, z_n$. Space for m values is allocated.

- **Size**

  `a.size()`

Return type:	int
Semantics:	Returns the size of the discretization: n

- **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

1.2 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.

1.2.1 Sampler

A sampler determines a value z according to a cdf F and assigns it to a Geovalue. Different methods are possible. The most used is Monte-Carlo simulation: a new value is drawn randomly from cdf F and assigned to the Geovalue. Other sampler can be defined:

- Metropolis Sampling: a value is drawn randomly from F but has a probability of being rejected. If it is rejected, the property value of the Geovalue remains unchanged.

- Quantile Sampling: always draw the same quantile from the cdf.

The sampler could also modify the cdf before drawing from it.

Associated Types

- **Return Type**

  `A::return_type`

  The type convertible to int.

Refinement of

Assignable

Notations

a	an object of a type that models Sampler
c	An object of a type that is a model of CDF1.1.11
g	An object of a type that is a model of Geovalue1.1.8

Valid Expressions

- **Draw Realization**

  `a(g,c)`

Return type:	`int`
Semantics:	Assigns a value to `g` given cdf `c`. Returns 0 if successful

Models

- `Monte_carlo_sampler`

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

- `Quantile_sampler`

  Returns a given quantile of the cdf.

1.2.2 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). Because the variables are stationnary, the covariance actually only depends on vector $\mathbf{h}$.

$$C: \ \mathbf{h} \longmapsto \mathrm{C}\Big(\mathrm{Z}_1(\mathbf{u}), \mathrm{Z}_2(\mathbf{u}+\mathbf{h})\Big) \quad \text{independent of } \ \mathbf{u}$$

Associated Types

- **Return Type**

  `A::return_type`

  The type of $C(\mathbf{u}_1, \mathbf{u}_2)$ (see definition above).

- **Argument Type**

  `A::argument_type`

  The type of location $\mathbf{u}$ (see definition above).

Refinement of

Assignable

Notations

A	A type that is a model of Covariance
a	an object of type A
U	A type that is a model of Location (see Section1.1.7)
u1,u2	two objects of type U
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`

1.2.3 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.

Refinement of

Assignable

Notations

a	an object of a type that models Kriging Constraint
M	an object of type Symmetric Matrix (see the description of kriging in 2.4.1 for a detailed description of this type)
V	an object of type Vector (see the description of kriging in 2.4.1 for a detailed description of this type)
neighb	an object of a type that models Neighborhood (see Section1.1.9)
u	an object of a type that models Location (see Section1.1.7)

Valid Expressions

- **Set-up the system**

  `a(M,V,u,neighb)`

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 $N * N$, 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. `u` is the location at which th kriging system is computed, `neighb` is a neighborhood of `u`. The returned value is the final size of the kriging system.

Models

- `OK_constraints`

  Imposes the constraints of ordinary kriging

- `KT_constraint`

  Imposes the constraints of kriging with trend

1.2.4 Cokriging Constraint

A (co)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 linear constraints as: "the kriging weights have to sum-up to 1". A functor that models `Cokriging Constraint` sets up this second part of the system. It takes in charge of defining the total size of the cokriging system and filling in the cokriging matrix entries belonging to non-covariance terms.

Refinement of

Assignable

Notations

a	an object of a type that models Cokriging Constraint
M	an object of type Symmetric Matrix (see the description of kriging in 2.4.1 for a detailed description of this type)
V	an object of type Vector (see the description of kriging in 2.4.1 for a detailed description of this type)
first,last	Two Input Iterators to a range of types that model Neighborhood (see Section1.1.9)
u	an object of type that models Location (see Section1.1.7)

Valid Expressions

- **Set-up the system**

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

Return type:	a type convertible to `unsigned int`
Semantics:	The total size N of the cokriging system is computed from the total number of neighboring data. The cokriging matrix M is then resized to N $*$ N, and vector V (second member of the cokriging system) is resized to N. Finally, the part of the system that does not depend on the covariances is computed. `u` is the location at which the cokriging system is computed, and `[first,last)` is a range of neighborhood of location `u` (one neighborhood for each variable involved in the cokriging system). The returned value is the final size of the cokriging system.

Models

- `OK_constraints`

  Imposes the constraints of ordinary kriging

1.2.5 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 $Z_i(\mathbf{u})$, $i = 2, \dots, M$ (instead of many neighborhoods of variables: $Z_i(\mathbf{u}+\mathbf{h^i_j})$). Hence the covariances $C_{i,j}(\mathbf{h})$, $i, j = 2, \dots, M$ need only be modeled for distance $\mathbf{h} = 0$. Moreover, the covariances $C_{1,j}$, $j = 2, \dots, M$ are (approximately) proportional to $C_{1,1}$ or $C_{j,j}$, depending on the Markov approximation used (MM1 or MM2).

Associated Types

- **Return Type**

  `A::return_type`

  The type of $C_{i,j}(\mathbf{u}_1, \mathbf{u}_2)$ (see definition above).

- **Argument Type**

  `A::argument_type`

  The type of location $\mathbf{u}$ (see definition above).

Refinement of

Assignable

Notations

a	an object of a type that models Covariance set
U	a type that models Location (see Section1.1.7)
u1,u2	objects of type U
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`

1.2.6 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 $\mathbf{u}$ 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) or even neural networks.

Refinement of

Assignable

Notations

a	an object of a type that models Cdf Estimator
f	an object of a type that models CDF1.1.11
u	an object of a type that models a Location (see Section1.1.7)
V	an object of a type that models a Neighborhood (see Section1.1.9)

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. If the returned value is different from 0 (i.e. the estimation of `f` failed), ccdf `f` has not been modified by the function call

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)

1.2.7 Multiple Variables Cdf Estimator

A Multiple Variables Cdf Estimator estimates a conditional distribution function of variable Z_1 at a location $\mathbf{u}$ 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).

Refinement of

Assignable

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 CDF1.1.11
u	an object of a type that models a Location (see Section1.1.7)
first,last	two models of Input Iterator (see STL) iterating on a range of models of Neighborhood (see Section1.1.9)

Valid Expressions

- **Estimate cdf**

  `a(u,first,last,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. If the returned value is different from 0 (i.e. the estimation of `f` failed), ccdf `f` has not been modified by the function call

Models

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

1.3 Iterators

1.3.1 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 (see Section1.1.4)

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 (see Section1.1.8)

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`

Chapter 2

Algorithms and Classes

2.1 Algorithms

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

- 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 (see Section1.1.7) and Geo-Value (see Section1.1.8):

```
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;
};
```

2.1.1 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 $F(z_i)$, $i = 1, \dots, n$.

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, $i = 1, \dots, n$, 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 Iterator1.1.4. 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.
- `non_param_cdf` is a model of Non-Parametric Cdf1.1.12.

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.

2.1.2 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 Iterator1.1.4.

- `target_cdf` is a model of CDF1.1.11.

- `data_cdf` is a model of CDF1.1.11.

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);
}
```

2.1.3 Kriging Weights

1.
```
template<
          class MatrixLibrary,
          class Location,
          class Neighborhood,
          class Covariance,
          class KrigingConstraints,
          class Vector
         >
int
kriging_weights(Vector& weights, double& kriging_variance,
                const Location& center, const Neigborhood& neighbors,
                Covariance& covar, KrigingConstraints& Kconstraint);
```

2.
```
template<
          class MatrixLibrary,
          class Location,
          class Neighborhood,
          class Covariance,
          class KrigingConstraints,
          class Vector
         >
int
kriging_weights(Vector& weights,
                const Location& center, const Neigborhood& neighbors,
                Covariance& covar, KrigingConstraints& 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 $f_i\Big(\{\lambda_\alpha\}\Big) = 0,\ i = 1,\dots,p$ are linear constraints expressed by the Kriging Constraint1.2.3 `Kconstraint`. The solution to this system is a set of weights:

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

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

The `kriging_weights` function computes the kriging weights and stores them into `Vector weights` in the following order: $(\lambda_1,\dots,\lambda_n,\mu_1,\dots,\mu_p)$. Only the weights $\lambda_1,\dots,\lambda_n$ are useful to compute the kriging estimate. The weights $\mu_1,\dots,\mu_p$ are used to compute the kriging variance. `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. Two versions of the algorithm exist. Version 1 computes the kriging variance and stores it into `kriging_variance`, while Version 2 does not compute the kriging variance (there are cases where the kriging variance is useless, e.g. in indicator kriging). Both version of the algorithm allow to change the linear algebra library (which defines matrices, matrix inversion routines, ...) used by `kriging_weights` (see 2.4). The default linear algebra library 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 value returned by `kriging_weights` indicates if any problem were encountered:

- returns 0 if the function executed successfully
- returns 1 if the kriging system could not be solved
- returns 2 if the neighborhood of conditioning data was empty

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

- `Location` is a model of Location (see Section1.1.7).
- `Neighborhood` is a model of Neighborhood (see Section1.1.9). The location type of the geo-values contained in the neighborhood must be `location`.
- `Covariance` is a model of Covariance1.2.2 that takes two objects of type `location` as arguments.
- `KrigingConstraints` is a model of Kriging Constraints1.2.3.
- `MatrixLibrary` specifies the linear algebra library2.4 to use. The requirements on `matrix_lib` are fully defined in 2.4. By default, it is the *TNT*library (slightly modified).

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 solving a linear system of equations, 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, kvariance,
                  u, &neighbors,
                  gauss_covariance, OK_constraint);
}
```

2.1.4 Cokriging Weights

1.
   ```
   template <
             class MatrixLibrary,
             class Location,
             class InputIterator,
             class CovarianceSet,
             class KrigingConstraints,
             class Vector
            >
   int
   cokriging_weights(
                     Vector& weights, double kriging_variance,
                     const Location& center,
                     InputIterator first_neigh, InputIterator last_neigh,
                     CovarianceSet& covar,
                     KrigingConstraints& Kconstraints
                     )
   ```

2.
   ```
   template <
             class MatrixLibrary,
             class Location,
             class InputIterator,
             class CovarianceSet,
             class KrigingConstraints,
             class Vector
            >
   int
   cokriging_weights(
                     Vector& weights,
                     const Location& center,
                     InputIterator first_neigh, InputIterator last_neigh,
                     CovarianceSet& covar,
                     KrigingConstraints& Kconstraints
                     )
   ```

The `cokriging_weights` algorithm solves a cokriging system :

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

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 `Kconstraint`. 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 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. `Vector weights` does not need to be of the correct size $n + p$ when passed to the function.

Two versions of the algorithm exist. Version 1 computes the kriging variance and stores it into `kriging_variance`, while Version 2 does not compute the kriging variance (there are cases where the kriging variance is useless, e.g. in indicator kriging). Both version of the algorithm allow to change the linear algebra library (which defines matrices, matrix inversion routines, ...) used by `cokriging_weights` (see 2.4). The default linear algebra library 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 value returned by `kriging_weights` indicates if any problem were encountered:

- returns 0 if the function executed successfully
- returns 1 if the kriging system could not be solved
- returns 2 if the neighborhood of conditioning data was empty

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 (see Section1.1.7).

- `InputIterator` is a model of Input Iterator (see STL requirements). It iterates on a range of Neighborhood (see Section1.1.9).

- `CovarianceSet` is a model of Covariance Set1.2.5. In expression `covar(i,j,u1,u2)`, required by Covariance Set1.2.5, `u1` and `u2` must be of type `Location`.

- `KrigingConstraints` is a model of Kriging Constraint1.2.3.

- `MatrixLibrary` specifies the linear algebra library2.4 to use. The requirements on `matrix_lib` are fully defined in 2.4. 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.

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 solving a linear system of equations, 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, kvariance,
                  u, neigh_array, 2,
                  LMC_covariance, OK_constraint);
}
```

2.1.5 Linear Combination

```
template<class InputIterator, class Neighborhood>
double
linear_combination(InputIterator begin_weights, InputIterator 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

- `InputIterator` is a model of Input Iterator (see STL).
- `Neighborhood` is a model of Neighborhood (see Section1.1.9).

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`.

2.1.6 Multi Linear Combination

```
template<class InputIterator, class InputIterator2>
double
multi_linear_combination(InputIterator begin_weights, InputIterator end_weights,
                         InputIterator2 first_neigh, InputIterator2 last_neigh)
```

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 range `[first_neigh, last_neigh)`. Denote $\lambda_1, \dots, \lambda_P$ the weights, and $z_j(\mathbf{u}_1^{\mathrm{j}}), \dots, \mathrm{z_j}(\mathbf{u}_{\mathrm{n_j}}^{\mathrm{j}})$, $j = 1, \dots, N_v$ property values of the geo-values contained in the N_v neighborhoods. `linear_combination` returns:

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

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

- `InputIterator` is a model of Input Iterator (see STL). The dereference type of the iterator must be convertible to `double`.

- `InputIterator2` is a model of Input Iterator (see STL). It iterates on a range of Neighborhood (see Section1.1.9)

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, kvariance,
                  u, neigh_array, 2,
                  LMC_covariance, OK_constraint);

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

2.1.7 Sequential Simulation, Single-Variable Case

1.
   ```
   template
   <
     class GeovalueIterator,
     class Neighborhood,
     class Cdf,
     class CdfEstimator,
     class MarginalCdf
   >
   inline int
   sequential_simulation(
     GeovalueIterator begin, GeovalueIterator end,
     Neighborhood& neighbors,
     Cdf& ccdf,
     CdfEstimator& estim,
     MarginalCdf& marginal
   )
   ```

2.
   ```
   template
   <
     class GeovalueIterator,
     class Neighborhood,
     class Cdf,
     class CdfEstimator,
     class MarginalCdf,
     class Sampler
   >
   inline int
   sequential_simulation(
     GeovalueIterator begin, GeovalueIterator end,
     Neighborhood& neighbors,
     Cdf& ccdf,
     CdfEstimator& estim,
     MarginalCdf& 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 neighborhood `neighbors`. 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 simulated by `sampler`.

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`. The random number generator is initialized by a default (constant) value. If control over the random number generator is needed, version 2 of the algorithm should be used.

Version 2 allows to specify a way to sampler from `ccdf`.

The value returned is the number of problems that occured during the simulation.

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

- `GeovalueIterator` is a model of Geo-Value Iterator (see Section1.3.1).
- `Neighborhood` is a model of Neighborhood (see Section1.1.9).
- `Cdf` is a model of CDF1.1.11.
- `MarginalCdf` is a model of CDF1.1.11. It can be different from `cdf`.
- `CdfEstimator` is a model of Single Variable Cdf Estimator (see Section1.2.6).
- `Sampler` (in version 2) is a model of Sampler (see Section1.2.1).

Example

A call to

```
// ...

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

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

//...
```

would simulate the single geo-value `Z`.

2.1.8 Sequential Simulation, Multiple-Variable Case

1.
   ```
   template
   <
     class GeovalueIterator,
     class ForwardIterator,
     class Cdf,
     class CdfEstimator,
     class MarginalCdf,
   >
   inline int
   sequential_cosimulation(
     GeovalueIterator begin, GeovalueIterator end,
     ForwardIterator first_neigh, ForwardIterator last_neigh,
     Cdf& ccdf,
     CdfEstimator& estim,
     MarginalCdf& marginal
   )
   ```

2.
```
template
<
  class GeovalueIterator,
  class ForwardIterator,
  class Cdf,
  class CdfEstimator,
  class MarginalCdf,
  class Sampler
>
inline int
sequential_cosimulation(
  GeovalueIterator begin, GeovalueIterator end,
  ForwardIterator first_neigh, ForwardIterator last_neigh,
  Cdf& ccdf,
  CdfEstimator& estim,
  MarginalCdf& 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 `neighborhoods`). 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`.

The value returned is the number of problems that occured during the simulation.

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

- `GeovalueIterator` is a model of Geo-Value Iterator (see Section1.3.1).
- `Neighborhood` is a model of Neighborhood (see Section1.1.9).
- `Cdf` is a model of CDF1.1.11.
- `MarginalCdf` is a model of CDF1.1.11. It can be different from `cdf`.
- `CdfEstimator` is a model of Multiple Variables Cdf Estimator (see Section1.2.7).
- `Sampler` (in version 2) is a model of Sampler (see Section1.2.1).

2.1.9 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 (see Section1.1.4), which iterates on Geo-Values (see Section1.1.8). It is not a model of GeoValue Iterator.
- `forward_iterator` is a model of Forward Iterator (see Section1.1.4), iterating on floating points values.
- `neighborhood` is a model of Neighborhood (see Section1.1.9). 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 Cdf1.1.11.
- `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.

2.2 Basic classes

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

- The object prototype is stated.
- **Template Parameters** lists what are the template parameters and what kind of concept they model (these could be non-G$_S$TL 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.

2.2.1 Gaussian Cdf

`Gaussian_cdf`

`Gaussian_cdf` defines a Gaussian cumulative distribution function of a continuous variable Z.

Where Defined

In header file `<cdf.h>`

Model of

CDF1.1.11

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 $p = Prob(Z \leq z)$.

2.2.2 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 $\Big(z_i, F(z_i)\Big)$, $i = 1, \dots, n$, $z_1 \leq \dots \leq z_n$. 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 (see Section1.1.12)

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 (see Section1.1.4).

- `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 (see Section1.1.4).

- `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 (see Section1.1.4) 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 (see Section1.1.4) to the end of the discretization $z_1, \dots, z_n$.

- `p_iterator Non_param_cdf::p_begin()`

  Returns a model of Forward Iterator (see Section1.1.4) to the first element of the set of probabilities $p_1, \dots, p_n$.

- `p_iterator Non_param_cdf::p_end()`

  Returns a model of Forward Iterator (see Section1.1.4) to the end of the set of probabilities $p_1, \dots, p_n$.

- `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 $p = Prob(Z \leq z)$.

2.2.3 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 $z_1, \dots, z_n$ 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 (see Section1.1.12)

Members

The leading `Categ_non_param_cdf::` is omited 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 (see Section1.1.4).

- `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 (see Section1.1.4).

- `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 (see Section1.1.4). 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 (see Section1.1.4). 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 (see Section1.1.4). 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 (see Section1.1.4) 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 (see Section1.1.4) to the end of set $z_1, \ldots, z_n$.

- `p_iterator non_param_cdf::p_begin()`

  Returns a model of Forward Iterator (see Section1.1.4) 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 (see Section1.1.4) 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 $p = Prob(Z \leq z)$.

2.3 Function Object Classes

2.3.1 Monte Carlo Sampler

`Monte_carlo_sampler<random_number_generator>`

`Monte_carlo_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 (see Section1.2.1)

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

- `Monte_carlo_sampler::Monte_carlo_sampler()`

  Default constructor.

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

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

-
  ```
  template<class Geovalue, class Cdf>
  typename cdf::value_type
  operator()(Geovalue& g, const Cdf& f)
  ```

  Function call operator. Type `cdf` is a model of CDF1.1.11. Draws a value from `f`.

2.3.2 Simple Kriging Constraints

`SK_constraints`

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

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

or, if secondary variables are accounted for:

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

`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 Constraint1.2.3

Type Requirements

See 2.4 for a thorough description of the requirements on the matrix library.

Members

- `SK_constraints::SK_constraints()`

  Default constructor.

-
  ```
  template<class InputIterator, class Location, class Matrix, class Vector>
  unsigned int
  SK_constraints::operator()(Matrix& A, Vector& b,
                             const Location& u
                             InputIterator first_neigh, InputIterator last_neigh)
  ```

Function call operator. `first_neigh,last_neigh)` is a range of Neighborhood (see Section1.1.9), and `Location` is a model of Location (see Section1.1.7). `A` is the kriging matrix and `b` the right hand side 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 2.4.

2.3.3 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:

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

`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 Constraint1.2.3

Type Requirements

See 2.4 for a thorough description of the requirements on the matrix library.

Members

- `OK_constraints::OK_constraints()`

  Default constructor.

-
  ```
  template<class InputIterator, class Location, class Matrix, class Vector>
  unsigned int
  OK_constraints::operator()(Matrix& A, Vector& b,
                             const Location& u,
                             InputIterator first_neigh, InputIterator last_neigh)
  ```

  Function call operator. `first_neigh,last_neigh)` is a range of Neighborhood (see Section1.1.9), and `Location` is a model of Location (see Section1.1.7). `A` is the kriging matrix and `b` the right hand side 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 2.4.

2.3.4 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 $\mathbf{u}$.

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

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

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 (see Section1.1.4).

Model of

Kriging Constraint1.2.3

Type Requirements

See 2.4 for a thorough description of the requirements on the matrix library.

Members

- `KT_constraints::KT_constraints(forward_iterator begin, forward_iterator end)`

  Constructs a `KT_constraint`. The range `[begin,end)` contains the functions f_i, $i = 1, \dots, K$ 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 (see Section1.1.7).

-
  ```
  template<class InputIterator, class Location, class Matrix, class Vector>
  unsigned int
  KT_constraints::operator()(Matrix& A, Vector& b,
                             const Location& u,
                             InputIterator first_neigh, InputIterator last_neigh)
  ```

  Function call operator. `first_neigh,last_neigh)` is a range of Neighborhood (see Section1.1.9), and `Location` is a model of Location (see Section1.1.7). `A` is the kriging matrix and `b` the right hand side 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 2.4.

2.3.5 LMC Covariance

`LMC_covariance<covariance_matrix>`

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

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 represents a matrix. Expression `mat(i,j)` must be valid and return element (i,j) of the matrix (i and j are greater than or equal to 1), and the matrix must have a copy constructor. The elements of the covariance matrix must be models of Covariance1.2.2

Model of

Covariance Set1.2.5

Type Requirements

The elements of the matrix must be models of Covariance1.2.2

Members

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

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

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

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

2.3.6 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 $(\mathbf{u}_1, \mathbf{u}_2) \longmapsto \mathrm{C}_{\mathrm{i,j}}(\mathbf{u}_1, \mathbf{u}_2)$ 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:

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

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

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 Covariance1.2.2
`matrix`	is an object that represents a matrix. Expression `mat(i,j)` must be valid and return element (i,j) of the matrix (i and j are greater than or equal to 1), and the matrix must have a copy constructor. The elements of `matrix` are of type convertible to `double`.

Model of

Covariance Set1.2.5

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 MM1_covariance::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.

2.3.7 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 (`i` is greater than or equal to 0). The elements of the vector must be models of Covariance1.2.2. `covariance_vector` must also have a copy constructor.
`matrix`	is an object that represents a matrix. Expression `mat(i,j)` must be valid and return element (i,j) of the matrix (i and j are greater than or equal to 1), and the matrix must have a copy constructor. The elements of `matrix` are of type convertible to `double`.

Model of

Covariance Set1.2.5

Type Requirements

- The elements of the matrix are of type convertible to `double`.
- `covariance_vector` contains pointers to models of Covariance1.2.2.

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 MM2_covariance::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.

2.3.8 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 Covariance1.2.2
`kriging_constraints`	is model of Kriging Constraint1.2.3. 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 (see Section1.2.6)

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 GaussianCdf>
  int operator()(const Location& u, const Neighborhood& neighbors,
                 GaussianCdf& ccdf)
  ```

  Function call operator. It estimates the gaussian cdf parameters and modifies `ccdf` accordingly. `location` is a model of Location (see Section1.1.7), and `neighborhood` is a model of Neighborhood (see Section1.1.9).

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

  `neighbors` is the neighborhood of location `u`. The function returns 0 if no problem occurred.

2.3.9 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 Set1.2.5
`kriging_constraints`	is model of Kriging Constraint1.2.3. 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 (see Section1.2.7)

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 InputIterator, class GaussianCdf>
  int operator()(const Location& u,
                 InputIterator first_neigh, InputIterator last_neigh,
                 GaussianCdf& ccdf)
  ```

  Function call operator. It estimates the gaussian cdf parameters and modifies `ccdf` accordingly. `location` is a model of Location (see Section1.1.7), and `[first_neigh,last_neigh)` is a range of models of Neighborhood (see Section1.1.9).

  `u` is the location at which the Gaussian conditional cdf is estimated. The function returns 0 if no problem occurred.

2.3.10 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 $Prob(Z(\mathbf{u}) \leq \mathrm{z_i})$.

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 (see Section1.1.4)
`constraints_iterator`	is model of Forward Iterator (see Section1.1.4)
`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 (see Section1.2.7)

Type Requirements

- `covar_iterator` iterates on a set of pointers to covariance functions, i.e. pointers to objects that are models Covariance1.2.2.

- `constraints_iterator` iterates on a set of pointers to kriging constraints, i.e. pointers to objects that are models of Kriging Constraint1.2.3.

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 (see Section1.1.7), and
  `neighborhood` is a model of Neighborhood (see Section1.1.9).

  `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, $i = 1, \ldots, n$.

2.3.11 Search Tree

`search_tree<T, allocator>`

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 $\mathbf{u}$, is the sequence of values:

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

$\mathbf{u}$ 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 later, Strebelle 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},. . . ,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>`

Template Parameters

`T`	is the type used to represent a category. It is a "discrete type", e.g. `int`, `bool`, . . .
`allocator`	is an object that manages the memory allocation for the search tree. Two models of allocator are available: a **pool allocator** and a **standard allocator**. The **standard allocator** allocates memory only when it is needed. This allows to use as little memory as possible, but may not be efficient in term of speed. Memory allocation is a slow task, and significant speed improvement can be achieved by decreasing the number of times memory is allocated. This is what the **pool allocator** does: memory is allocated by large chunks or pools, and when new space is needed, it is taken from the pool without requiring the system to allocate some new memory. The **standard allocator** should be used when memory is an issue, while the **pool allocator** is useful to improve performance. **pool alocator** is the default allocator.

Model of

Single Variable Cdf Estimator (see Section1.2.6)

Members

-
  ```
  template<class window_neighborhood, class forward_iterator>
  search_tree::search_tree(forward_iterator begin, forward_iterator end
                           window_neighborhood& neighbors,
                           int window_size, int nb_of_categories)
  ```

  Constructs a `search_tree`.

  `forward_iterator` is a model of Forward Iterator (see Section1.1.4). It iterates on a set of geo-values, the training image.

  `neighborhood` is a model of Window Neighborhood (see Section1.1.10). It is used to define the data event associated to each geo-value in range `[begin,end)`.

  `window_size` is the maximum number of geovalues `neighbors` can contain. `nb_of_categories` is the number of categories (e.g. "sand", "mud") that we want to work with.

-
  ```
  template<class location, class non_param_cdf>
  int search_tree::operator()(const location& u,
                              const window_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 (see Section1.1.7), and `window_neighborhood` is a model of Window Neighborhood (see Section1.1.10).

  `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 $Z(\mathbf{u}+\mathbf{h_i})$. The function returns 0 if no problem occurred.

2.4 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 G_STL 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. 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 G_STL 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 2.4.1.

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

```
template<class T>
struct TNT_lib{

  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);
  }
```

It has two nested types: `Matrix`, and `Vector` which are defined by *TNT*, and two functions: a Cholesky factorization, a LU factorization, 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>`. What has been done in the case of *TNT* can serve as a model.

2.4.1 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 (see Section1.1.4). 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.