User Guide: STK++ Arrays

Serge Iovleff

October 25, 2022

Abstract

This user guide gives a general review of the STK++ arrays and of their usages. After giving a general review of the capability of the library, we detail the various accessors/mutators available. We give a complete list of constructors. We focus on the Array2D class and give some hints about their capabilities. We then present the different visitors, appliers and functors implemented in STK++ allowing to compute quantities or arrays of interests. We then explain how it is possible to use or modify slices of arrays. Finally we end this document by detailing the way to construct reference arrays.

1 Introduction

The containers/arrays you use in order to store and process the data in your application greatly influence the speed and the memory usage of your application. STK++ proposes a large choice of containers/arrays and methods that you can used in conjunction with them.

There are mainly two kinds of arrays and a wrapper you can use in STK++:

- The Array2D family classes which are the classes defined in the oldest versions of STK++,
- the CArray family classes which have been introduced in version 0.4 of STK++ library.
- For rtkore users, the RVector and RMatrix wrap the R SEXP stucture.

Before explaining the usage and differences between the different arrays, we first introduce some vocabulary. The terminology used in STK++ project for the arrays is the following:

- An array is often called a matrix.
- In the case where a matrix has 1 column, such matrix is called column-vector, often abbreviated just as *vector*,
- in the other case, where a matrix has 1 row, it is called row-vector, often abbreviated just as *point*.

The word *point* is borrowed from the statistical vocabulary where a row of a data array is often named a point.

The Array2D classes are very flexible if you need to add, insert, remove, resize,... quickly rows or columns to your container. On the other hand, the storing scheme of the the CArray classes allow you to used them easily with other linear algebra libraries (e.g. lapack).

2 A detailed introductory example

Consider the following example:

Listing 1: Introductory example

```
#include "STKpp.h"
using namespace STK;
int main(int argc, char** argv)
{
  // create a matrix of Real of dynamic size (3,5)
  Array2D<Real> a(3, 4); a << 1.,2.,3.,4.
                          , 1.,2.,3.,4.
                          , 1.,1.,1.,1.;
  std::cout << "a=\n" << a;
  // create a vector of Real of dynamic size 3 with all coefficients
      equal to 0
  Array2DVector<Real> b(3, 0);
  b[2] = 1.;
  std::cout << "b=\n" << b;
  // create an unitialized CArray of Real of fixed size (3,4)
  CArray<Real, 3, 4> c;
  // create a CArray of Real of dynamic size (3,4) with initial value
      -2.
  CArray<Real> d(3, 4, -2.);
  // compute c = -a - d + 3 (Id)
  c= -a - d  + 3.;  c(2,2) = 5.;
  std::cout << "c=\n" << c;
  // create an unitialized CVector of Real of fixed size 3
  CArrayVector<Real, 3> e;
  e = -2.; e[1] = 5.;
  std::cout << "e=\n" << e;
  // create an initialized CPoint of Real of fixed size 3
  CArrayPoint<Real, 3> f;
  f = -2.; f[1] = 5.;
  std::cout << "f= " << f;
  return 0;
}
```

Listing 1: Output

```
a=
1 2 3 4
1 2 3 4
1 1 1 1
b=
0
0
1
c=
4 3 2 1
4 3 2 1
4 4 5 4
e=
-2
 5
-2
f= -2 5 -2
```

2.1 Accessors

The primary coefficient accessors and mutators in STK++ are the overloaded parenthesis operators. For matrices, the row index is always passed first.

The `operator[]` is also overloaded for index-based access in vectors, but keep in mind that C++ doesn't allow `operator[]` to take more than one argument. The `operator[]` is thus restricted to vectors/points/diagonal matrices. For vectors, just pass one index in a bracket.

Indexing starts at 0 by default. This behavior can be modify by defining the STKBASEARRAYS macro at compile time using the directive -DSTKBASEARRAYS=1. Enabling this macro allow user to get 1 based arrays like in FORTRAN. If you want to build code independent of the first index you should use the `beginCols()`, `beginRows()`, `endRows()`, `endCols()`, `lastIdxCols()` and `lastIdxRows()` methods of the arrays and the `begin()`, `end()`, `lastIdx()` methods of the Points (Row-vectors), Vectors, square matrices and diagonal matrices.

Here an example

Listing 2: Accessors

```
#include "STKpp.h"
using namespace STK;
int main(int argc, char** argv)
{
  ArrayXX t(5, 5); // array of size 5x5
  for (int i=t.beginRows(); i<t.endRows(); i++)
  {
    PointX r(t.row(i), true); // create a reference on the i-th row of
        t
    // fill the i-th row of t with the number i
    r= i; // same as for (int j=r.begin(); j<r.end(); j++) { r[j] = i;}
  }
  std::cout << "t =\n" << t;
  return 0;
}
```

Listing 2: Output

```
t =
0 0 0 0 0
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
```

2.2 Expression template

Assume that `c, a, d` are arrays of the same size and consider the line of code

```
c= -a - d  + 3.;
```

It is an expression involving matrix operations. All these expressions are encoded in an expression template and are completely inlined at compile time. It means there is no temporary objects created when these expressions are evaluated.

2.3 Constructors

The `Array2D` family have only one mandatory template parameter: the type of the data that will be stored. On the other hand arrays in `CArray` family have four template parameters:

1. the type of the data that will be stored,
2. the number of rows (`UnknownSize` if it is not known at compile time),
3. the number of columns (`UnknownSize` if it is not known at compile time),
4. the orientation storage scheme of the data (by row or by column).

Only the first template parameter is mandatory.

Listing 3: Constructors

```
#include "STKpp.h"
using namespace STK;
int main(int argc, char** argv)
{
  // Array2D constructors
  Array2DDiagonal<Real> a(3); a= 1.; // same as Array2DDiagonal<Real> a(3, 1.);
  Array2DSquare<Real> b(3); b = 0.; // same as Array2DSquare<Real> b(3, 0.);
  Array2DUpperTriangular<Real> c(3, 3); c= 2.; // same as Array2DUpperTriangular<Real> c(3, 3, 2.);
  Array2DLowerTriangular<Real> d(3, 3); d = -2.;// same as Array2DLowerTriangular<Real> d(3, 3, -2.);
  Array2DVector<Real> e(3); e= 5.; // same as Array2DVector<Real> e(3, 5.);
  Array2DPoint<Real> f(3); f = 6.; // same as Array2DPoint<Real> f(3, 6.);
  // CArray constructors
  CArraySquare<Real> g(3); g= 0.; // same as CArraySquare<Real> g(3, 0.);
  CArrayVector<Real> j(3); j= 5.; // same as CArrayVector<Real> j(3, 5.);
  CArrayPoint<Real> k(3); k = 6.; // same as CArrayPoint<Real> k(3, 6.);
  return 0;
}
```

2.4 Accessing rows/columns/parts of an array

You can access rows, columns and sub-part of STK++ arrays easily. Here is an example:

Listing 4: Example

```
#include "STKpp.h"
using namespace STK;
int main(int argc, char** argv)
{
  VectorX b(3, 0);
  std::cout << "b=\n" << b;
  b.sub(Range(2)) = 1.;
  std::cout << "b=\n" << b << "\n";
  ArrayXX a(3, 4); a << 1.,2.,3.,4.
                     , 1.,2.,3.,4.
                     , 1.,1.,1.,1.;
  std::cout << "a=\n" << a;
  a.col(1) = b;
  a.row(1, Range(2,2)) = 0.;
  a.sub(Range(1,2), Range(2,2)) = 9.;
  std::cout << "a=\n" << a;
  return 0;
}
```

Listing 4: Output

```
b=
0
0
0
b=
1
1
0

a=
1 2 3 4
1 2 3 4
1 1 1 1
a=
1 1 3 4
1 1 9 9
1 0 9 9
```

Applied to a vector/point/diagonal matrix the method sub require only one parameter: the Range of the data we want to access/mutate.

In the general case, you can use the following methods in order to access/mutate columns/rows/part of an array:

Listing 5: Review of the main accessors

```
// access to the ith row
 a.row(i);
// access to the element [first, first+size-1] of the ith row
 a.row(i, Range(first, size));
// access to the jth column
 a.col(j);
// access to the element [first, first+size-1] of the jth columns
 a.col(j, Range(first, size));
// access to the element [first, first+size-1] of the jth columns
 a.col(j, Range(first, size));
// Create ranges
 Range I(3,2), J(4,2);
// access to the sub-array formed by the range I,J.
 a.sub(I, J);
// access to the row i, in the range J = 4:5
 a(i,J);
 // access to the column j in the Range I=3:4
 a(I,j);
 // access to a subarray in the range (3:4, 4:5)
 a(I,J);
```

2.5 Using references and move

In some cases, you may want to conserve an access to some part of an array for some work. For this purpose, it is possible to create reference array, that is array that wrap (part of) another array

Listing 6: reference and move

```
#include "STKpp.h"
using namespace STK;
/** @ingroup tutorial */
int main(int argc, char** argv)
{
  ArraySquareX a(4); a << 1.,2.,3.,4.
                        , 1.,2.,3.,4.
                        , 1.,1.,1.,1.
                        , 1.,1.,1.,1.;
  // create a reference
  ArrayXX b(a.sub(Range(2), Range(3)), true);
  b = -1.;
  std::cout << "Modified a=\n" << a << "\n";

  PointX m;
  m.move(Stat::mean(a));
  std::cout << "m= " << m;
  return 0;
}
```

Listing 6: Output

```
Modified a=
-1 -1 -1 4
-1 -1 -1 4
 1  1  1 1
 1  1  1 1

m= 0 0 0 2.5
```

The `Stat::mean` function return an `Array2D` *by value*. In order to avoid a useless copy, we use the `move` function. The following piece of code

```
a.move(b);
```

perform the operations:

- if `a` contains data, the memory is released,
- `a` become the owner of the data contain by `b`,
- `b` become a reference,
- If `b` was a reference, then `a` is also a reference.

> **Note:**
> Many more examples can be found in the test files coming with the STK++ library.

3 Arrays classes and their constructors

As say in the Introduction, there is two kind of arrays provides by the STK++ library: the arrays part of the Array2D family and the arrays part of the CArray family. Constructors depends of the structure of the array you want to obtain. In this part we will review all the existing arrays and the different way to create them.

3.1 Review of the classes

3.1.1 Array2D and CArray classes

The Array2D and CArray classes are the most general classes for storing scalars. The values are stored in an array of range [beginRows:endRows) × [beginCols:endCols).

```
template<typename Type> class \code{Array2D};
template< typename Type, int SizeRows_ = UnknownSize, int SizeCols_ = UnknownSize, bool Orient_ = Arrays
    ::by_col_>
class CArray;
```

3.1.2 Array2DVector, Array2DPoint, CArrayVector and CArrayPoint classes

The Array2DVector, Array2DPoint, CArrayVector and CArrayPoint classes allow to store scalars in column and row vectors. The values are stored in arrays of range [begin:end).

```
template<typename Type> \code{Array2D}Vector;
template<typename Type> \code{Array2D}Point;
template< typename Type, int SizeRows_=UnknownSize, bool Orient_ = Arrays::by_col_> class CArrayVector;
template< typename Type, int SizeCols_=UnknownSize, bool Orient_ = Arrays::by_row_> class CArrayPoint;
```

> **Note:**
> Only the first template parameter is mandatory.

3.1.3 Array2DSquare and CArraySquare classes

The Array2DSquare and CArraySquare classes are general classes for storing scalars in a square matrix. The values are stored in an array of range [begin:end) × [begin:end).

```
template<typename Type> \code{Array2D}Square;
template< typename Type_, int Size_ = UnknownSize, bool Orient_ = Arrays::by_col_>
class CArraySquare;
```

> **Note:**
> Only the first template parameter is mandatory.

3.1.4 Array2DDiagonal class

The Array2DDiagonal class is a general class for storing scalars in a diagonal matrix. The values are stored in an array of range [begin:end) × [begin:end) with zero outside the diagonal.

```
template<typename Type> \code{Array2D}Diagonal;
```

> **Note:**
> Only the diagonal values are effectively stored in an Array2DDiagonal array.

3.1.5 `Array2DUpperTriangular` and `Array2DLowerTriangular` classes

The `Array2DUpperTriangular` and `Array2DLowerTriangular` classes are general classes for storing scalars in triangular matrices. The values are stored in an array of range `[beginRows:endRows)` × `[beginCols:endCols)` with zero in the respectively lower and upper part of the array.

```
template<typename Type> \code{Array2D}UpperTriangular;
template<typename Type> \code{Array2D}LowerTriangular;
```

Note:
Only the non-zeros values are effectively stored in the arrays and the upper part (respectively the lower part) is the part of the array such that $j > i$ (resp. $i > j$), where i represents the index of the row and j the index of the column.

3.2 Constructors for the `Array2D` family containers

3.2.1 Default constructors

All default constructors create an empty Array of scalar `Type`

```
Array2D<Type> a;
Array2DVector<Type> v;
Array2DPoint<Type> p;
Array2DSquare<Type> s;
Array2DDiagonal<Type> d;
Array2DLowerTriangular<Type> l;
Array2DUpperTriangular<Type> u;
```

In the following example

```
Array2D<Real> a; // a is an empty array of Real (double by default)
a.resize(10,10)=0.; // a is now an array of size (10,10) with the value 0.
Array2DVector<Real> b; // a is an empty column vector of Real
b.resize(10)=0.; // b is now an column-vector of size 10 with the value 0.
```

3.2.2 Constructors with given dimensions

It is possible to declare an array with fixed ranges for the rows and the columns and Optionally an initial value

```
Array2D<Type>( Range const& I, Range const& J);
Array2D<Type>( Range const& I, Range const& J, Type const& v);
Array2DVector<Type>( Range const& I);
Array2DVector<Type>( Range const& I, Type const& v);
Array2DPoint<Type>( Range const& J);
Array2DPoint<Type>( Range const& J, Type const& v);
Array2DSquare<Type>( Range const& I);
Array2DSquare<Type>( Range const& I, Type const& v);
Array2DDiagonal<Type>( Range const& I);
Array2DDiagonal<Type>( Range const& I, Type const& v);
Array2DLowerTriangular<Type>( Range const& I, Range const& J);
Array2DLowerTriangular<Type>( Range const& I, Range const& J, Type const& v);
Array2DUpperTriangular<Type>( Range const& I, Range const& J);
Array2DUpperTriangular<Type>( Range const& I, Range const& J, Type const& v);
```

The following code build an array of `Real` (double by default) with rows in the range $2:4$ and columns in the range $0:9$ (this is the default, but could be $1:10$ if the macro STKBASEARRAYS=1 is activated). The array is initialized with the value 2

```
Array2D<Real> a(Range(2,3), 10, 2.);
```

The following code build a vector of `Real` (double by default) with columns in the range $0:9$ (this is the default, but could be $1:10$ if the macro `STKBASEARRAYS=1` is activated). The array is initialized with the value 2

```
Array2DVector<Real> a( 10, 2.);
```

3.2.3 Copy constructors

All arrays have a copy constructor. In the following examples, the resulting array is either a copy of the array `T` (if `ref == false`) or a reference of the array `T` (if `ref == true`)

```
Array2D<Type>( \code{Array2D}<Type> const& T, bool ref=false) a; // a is a physical copy of T
Array2DVector<Type>( \code{Array2D}Vector<Type> const& T, bool ref=false) v;  // v is a physical copy of
    T
Array2DPoint<Type>( \code{Array2D}Point<Type> const& T, bool ref=false);
Array2DSquare<Type>( \code{Array2D}Square<Type> const& T, bool ref=false);
Array2DDiagonal<Type>( \code{Array2D}Diagonal<Type> const& T, bool ref=false);
Array2DLowerTriangular<Type>( \code{Array2D}LowerTriangular<Type> const& T, bool ref=false);
Array2DUpperTriangular<Type>( \code{Array2D}UpperTriangular<Type> const& T, bool ref=false);
```

The following code copy the array `a` in a new array `b` and create a reference of the rows and columns of `c` with range I for the rows and range J for the columns.

```
Array2D<Real> a(10, 8);
Array2D<Real> b(a); // b is now a copy of a
// c is a reference, if an element of c is modified, the corresponding element of a is modified
Array2D<Real> c(a.sub(I,J), true);
```

The following shows how to get a reference on a column and a reference on a row of an `Array2D`.

```
Array2D<Real> a(10, 8);
Array2DVector<Real> c(a.col(1), true); // c is now a reference on the column 1 of a
Array2DPoint<Real> p(a.row(1), true); // p is now a reference on the row 1 of a
```

Warning:
It is the responsibility of the user to verify that a reference is valid.

3.3 Constructors for the `CArray` family containers

3.3.1 Default constructors

All default constructors create an empty Array of scalar `Type`

```
CArray<typename Type, SizeRows_, SizeCols_ = UnknownSize, bool Orient_ = Arrays::by_col_ >();
Array2DVector<typename Type, int Size_ = UnknownSize, bool Orient_ = Arrays::by_col_>();
CArrayPoint<typename Type, int Size_ = UnknownSize, bool Orient_ = Arrays::by_row_>();
CArraySquare<typename Type, int Size_ = UnknownSize, bool Orient_ = Arrays::by_col_>();
```

Note:
The only mandatory template argument is `Type`.

Here after are some more examples

```
CArray<Real> a; // a is an empty array of Real (double by default)
a.resize(10,10) =0.; // a is now an array of size (10,10) with the value 0.
CArrayVector<Real> b; // a is an empty column vector of Real
b.resize(10) =0.; // b is now a column-vector of size 10 with the value 0.
```

3.3.2 Constructors with given dimensions

It is possible to construct an array with fixed ranges for the rows and the columns and optionally an initial value.

```
CArray<Type>( Range const& I, Range const& J);
CArray<Type>( Range const& I, Range const& J, Type const& v);
CArrayVector<Type>( Range const& I);
CArrayVector<Type>( Range const& I, Type const& v);
CArrayPoint<Type>( Range const& J);
CArrayPoint<Type>( Range const& J, Type const& v);
CArraySquare<Type>( Range const& I);
CArraySquare<Type>( Range const& I, Type const& v);
```

If the dimension is known at compile time, you can use the template arguments

```
CArray<Type, 10, 10> a; // uninitialized array of fixed size (10,10)
CArray<Type, 10, 10> a(0,0,1)); // array of fixed size (10,10) with initial value 1
```

The following code build an array of Real (double by default) with rows in the range $2:4$ and columns in the range $0:9$ (this is the default, but could be $1:10$ if the macro STKBASEARRAYS=1 is activated). The array is initialized with the value 2.

```
CArray<Real> a(Range(2,3), 10, 2.);
```

The following code build a vector of Real (double by default) with columns in the range $0:9$ (this is the default, but could be $1:10$ if the macro STKBASEARRAYS=1 is activated). The array is initialized with the value 2.

```
CArrayVector<Real> a( 10, 2.);
```

3.3.3 Copy constructors

In the following examples, the resulting array is either a copy of the array T (if ref == false) or a reference of the array T (if ref == true)

```
CArray<Type>( const CArray<Type> const& T, bool ref=false);
CArrayVector<Type>( CArrayVector<Type> const& T, bool ref=false);
CArrayPoint<Type>( CArrayPoint<Type> const& T, bool ref=false);
CArraySquare<Type>( CArraySquare<Type> const& T, bool ref=false);
```

The following code copy the array a in a new array b and create a reference of the rows and columns of c with range I for the rows and range J for the columns

```
CArray<Real> a(10, 8);
CArray<Real> b(a); // b is now a copy of a
// c is a reference, if an element of c is modified, the corresponding element of a is modified
CArray<Real> c(a.sub(I,J), true);
```

The following show how to get a reference on a column and a reference on a row of a CArray.

```
CArray<Real> a(10, 8);
CArrayVector<Real> c(a.col(1), true); // c is now a reference on the column 1 of a
CArrayPoint<Real> p(a.row(1), true); // p is now a reference on the row 1 of a
```

4 Using Array2D

This section explains how you can add and/or remove columns/rows to an array of the Array2D family and how to merge arrays with the same number of rows without duplicating data.

4.1 General features of the Array2D

The Array2D is a family of array/vectors/points proposing much functionality allowing to add/remove rows/-columns. It is also possible to resize them in a conservative way. It is also possible to merge them if the dimensions agreed.

Each columns in an Array2D is in a contiguous memory block. Thus, all operations on the columns are fast as it does not involve data copy. This is not the case for the rows. Inserting or appending rows will cause in the worse cases, that is if the number of rows required is greater than the capacities of the columns, a re-allocation of a part (all) the columns.

4.2 Adding rows and columns to Array2D

All the Array2D (That is the arrays deriving from the interface class IArray2D) can add, insert, remove rows or columns to an existing data set.

In this first example, we add columns to an Array2D

Listing 7: Adding columns

```
#include "STKpp.h"
using namespace STK;
int main(int argc, char *argv[])
{
  ArrayXX A(4, Range(0,2));
  Law::Normal law(1.,2.);
  A.rand(law);
  stk_cout << _T("A =\n") << A;
  // Adding a column with value 5
  A.pushFrontCols(Const::VectorX(4)*5);
  // insert an uninitialized columns at place 2
  A.insertCols(2, 2);
  // set x^2 and x^3 with x in columns 1
  A.col(2) = A.col(1).square();
  A.col(3) = A.col(1).cube();
  // add an uninitialized columns at the end
  A.pushBackCols(1);
  // set x^2 with x in columns 4
  A.col(5) = A.col(4).square();
  stk_cout << _T("A =\n") << A;
  // do the same with a vector
  VectorX B(4); B << 1, 3, 5, 6;
  stk_cout << _T("B = ") << B.transpose();
  // insert 1 uninitialized element in place 2
  B.insertElt(2); B[2] = 3; // set value to 3
  stk_cout << _T("B = ") << B.transpose();
}
```

Listing 7: Output

```
A =
0.0615321  -0.617549
 -1.11315    2.73537
 -1.55495    1.97633
  3.92467    2.21783
A =
5  0.0615321  0.00378619  0.000232972  -0.617549  0.381367
5   -1.11315     1.23911     -1.37932    2.73537   7.48223
5   -1.55495     2.41786     -3.75965    1.97633    3.9059
5    3.92467      15.403      60.4518    2.21783   4.91877
B = 1 3 5 6
B = 1 3 3 5 6
```

And in this second example, we add rows to an `Array2D`

Listing 8: Adding rows

```
#include "STKpp.h"
using namespace STK;
int main(int argc, char *argv[])
{
  ArrayXX A(Range(0,2), 4);
  Law::Normal law(1.,2.);
  A.rand(law);
  stk_cout << _T("A =\n") << A;
  // Adding a column with 1
  A.pushFrontRows(Const::PointX(4));
  // Insert 2 uninitialize rows at place 2
  A.insertRows(2, 2);
  // Set x^2 and x^3 with x in row 1
  A.row(2) = A.row(1).square();
  A.row(3) = A.row(1).cube();
  // Add 2 unitialized rows
  A.pushBackRows(2);
  // Set x^2 and x^3 with x in row 4
  A.row(5) = A.row(4).square();
  A.row(6) = A.row(4).cube();
  stk_cout << _T("A =\n") << A;

  PointX B(4); B << 1, 3, 5, 6;
  stk_cout << _T("B =\n") << B;
  // insert an unitialized element at place 2
  B.insertElt(2); B[2] = 3; // set value to 3
  stk_cout << _T("B =\n") << B;
}
```

Listing 8: Output

```
A =
  5.01666    4.1416   1.91137  0.0289455
-0.330411  0.837067  0.452249    4.28104
A =
          1         1          1           1
    5.01666    4.1416    1.91137   0.0289455
    25.1669   17.1529    3.65332 0.000837844
    126.254   71.0404    6.98284 2.42518e-05
  -0.330411  0.837067   0.452249     4.28104
   0.109172  0.700682   0.204529     18.3273
 -0.0360715  0.586518  0.0924981     78.4598
B =
1 3 5 6
B =
1 3 3 5 6
```

Note:

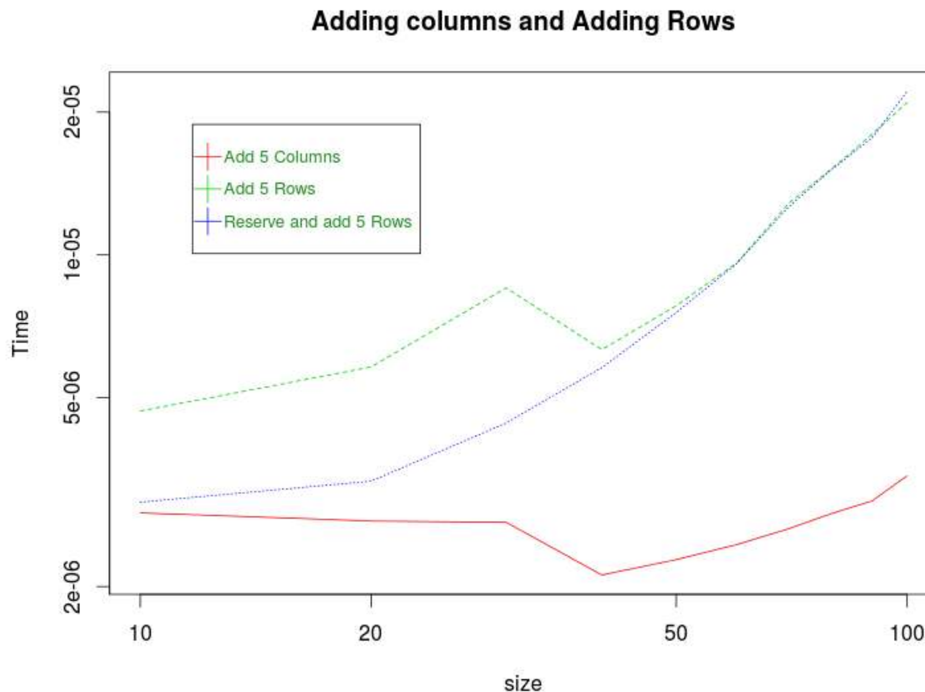
It is important to observe that adding/inserting columns to an `Array2D` does not produce any copy of the data stored, while the same operations involving the rows of the array will produce data movement.

The next graphic give the execution time of the following code

```
A.pushFrontCols(Const::VectorX(m));
A.insertCols(2, 2);
A.col(2) = A.col(1).square();
A.col(3) = A.col(1).cube();
```

```
    A.pushBackCols(2);
    A.col(5) = A.col(4).square();
    A.col(6) = A.col(4).cube();
```

and of the same code by replacing "Cols" by "Rows". The size of the array `A` was (m, m). It is easily observed that adding rows is time consuming. Observe also that a small amount of space is added at the creation of the object and that the usage of the `reserve` method can be wise in order to save time.



4.3 Removing Rows and Columns

Rows and columns can be also removed from arrays of the `Array2D` family using `remove` and `popBack` methods.

Listing 9: Removing rows and columns

```
#include "STKpp.h"
using namespace STK;
int main(int argc, char *argv[])
{
  ArrayXX A(5, 6);
  Law::Normal law(1.,2.);
  A.rand(law);
  stk_cout << _T("A =\n") << A;
  // Remove 2 columns
  A.popBackCols(2);
  // erase two rows in position 1
  A.eraseRows(1,2);
  stk_cout << _T("A =\n") << A;

  PointX B(6); B << 1, 2, 3, 4, 5, 6;
  stk_cout << _T("B =\n") << B;
  // erase three elements at the position 2
  B.erase(2,3);
  stk_cout << _T("B =\n") << B;
}
```

Listing 9: Output

```
A =
-0.368484 -0.816883   2.29464 0.0968061 0.467136
     1.46384
  1.44368    1.44231   2.00604 -0.447036   3.26891
      -1.92511
  4.45443    3.14678   1.56722     2.2803   1.86343
     1.84961
 -2.27432 -0.504987  -0.80423   -1.12455 0.381285
     1.07171
 -2.01589    5.72782 0.884305   -3.11897   1.19921
     3.89621
A =
-0.368484 -0.816883   2.29464 0.0968061
 -2.27432 -0.504987  -0.80423   -1.12455
 -2.01589    5.72782 0.884305   -3.11897
B =
1 2 3 4 5 6
B =
1 2 6
```

4.4 Merging two arrays

It is possible to merge an array with an other array or vector without copying the data using the merge method.

Listing 10: Merge an array with an array and a vector

```
#include "STKpp.h"
using namespace STK;
int main(int argc, char *argv[])
{
  ArrayXX A(3, 2), B(3, 3);
  A << 1, 2, 3, 4, 5, 6; B << 1, 2, 3, 4, 5, 6, 7, 8, 9;
  stk_cout << _T("B.isRef() = ") << B.isRef() << _T('\n');
  // merge B with A
  A.merge(B);
  stk_cout << _T("A =\n") << A << _T('\n');
  stk_cout << _T("B.isRef() = ") << B.isRef() << _T('\n');
  stk_cout << _T("B =\n") << B<< _T('\n');
  // merge C with A
  VectorX C(3); C << 4, 5, 6;
  A.merge(C);
  stk_cout << _T("A =\n") << A << _T('\n');
  // B has been invalidate by A.merge(C) and cannot be used anymore
  // stk_cout << _T("B =\n") << B; produce an error at execution
  stk_cout << _T("C.isRef() = ") << C.isRef() << _T('\n');
  stk_cout << _T("C =\n") << C;
}
```

Listing 10: Output

```
B.isRef() = 0
A =
1 2 1 2 3
3 4 4 5 6
5 6 7 8 9

B.isRef() = 1
B =
1 2 3
4 5 6
7 8 9

A =
1 2 1 2 3 4
3 4 4 5 6 5
5 6 7 8 9 6

C.isRef() = 1
C =
4
5
6
```

5 Using Visitors, Appliers and Functors

This page explains STK++'s visitors and appliers and how they can be used on the arrays and in expressions.

5.1 Visitors

A visitor is a constant function applied to an expression or array returning a single value. One of the most useful visitor is `ExprBase::sum()`, returning the sum of all the coefficients of a given expression or array.

Visitors can also be used to obtain the location of an element inside an Expression or Array. This is the case of `ExprBase::maxElt(i,j)`, `ExprBase::maxElt(i)`, `ExprBase::minElt(i,j)` and `ExprBase::minElt(i)`, which can be used to find the location of the greatest or smallest coefficient in an Expression or an Array.

The `minElt` method should not be confounded with the `min` method which compute the minimum between two arrays.

The next example shows different usage of visitors

Listing 11: Usage of the visitors

```
#include "STKpp.h"
using namespace STK;
/** @ingroup tutorial */
int main(int argc, char** argv)
{
 CArray22 a;
 a << 0, 1,
      2, 3;
 stk_cout << "a.sum()= " << a.sum()   << _T("\n");
 stk_cout << "a.minElt()= " << a.minElt() << _T("\n");
 stk_cout << "a.maxElt()= " << a.maxElt() << _T("\n");
 stk_cout << "(a > 0).count()= " << (a > 0).count() << _T("\n");
 stk_cout << "(a > 0).all()= " << (a > 0).all() << _T("\n");
 stk_cout << "(a > 0).any()= " << (a > 0).any() << _T("\n");
 int i,j;
 stk_cout << "a.minElt(i,j)= " << a.minElt(i,j) << _T("\n");
 stk_cout << "i= " << i << ", j= " << j << _T("\n");
 return 0;
}
```

Listing 11: Output

```
a.sum()= 6
a.minElt()= 0
a.maxElt()= 3
(a > 0).count()= 3
(a > 0).all()= 0
(a > 0).any()= 1
a.minElt(i,j)= 0
i= 0, j= 0
```

5.2 Slicing Visitors

A visitor can also be applied on each column/row of an Expression or Array by using global functions in the `STK` namespace. All the visitors can be used as slicing visitors except the `minElt` with arguments.

Listing 12: Slicing visitors

```
#include "STKpp.h"
using namespace STK;
/** @ingroup tutorial */
int main(int argc, char** argv)
{
 CArray2X a(2,4);
 a << 0, 1, 2, 3
    , 4, 5, 6, 7;
 stk_cout << "min(a)       = " << min(a);
 stk_cout << "minByRow(a) = " << minByRow(a);
 stk_cout << "mean(a) = "     << mean(a);
 stk_cout << "count(a > 0)= " << count(a > 0);
 stk_cout << "all(a > 0)   = " << all(a > 0);
 stk_cout << "any(a > 0)   = " << any(a > 0);
 a(1,1) = a(1,3) = Arithmetic<Real>::NA();
 stk_cout << "count(a.isNA())= "   << count(a.isNA());
 stk_cout << "countByRow(a.isNA())= "   << countByRow(a.isNA());
 return 0;
}
```

Listing 12: Output

```
min(a)       = 0 1 2 3
minByRow(a) = 0
4
mean(a) = 2 3 4 5
count(a > 0)= 1 2 2 2
all(a > 0)   = 0 1 1 1
any(a > 0)   = 1 1 1 1
count(a.isNA())= 0 1 0 1
countByRow(a.isNA())= 0
2
```

5.3 Appliers

An applier is like a visitor except that it can only be applied to arrays as it will modify the content of the array. Slicing appliers does not have any meaning and thus an applier can only be used on a whole expression. If you need to use an applier on a row or column use the `col` or `row` method.

Listing 13: Appliers

```
#include "STKpp.h"
using namespace STK;
/** @ingroup tutorial */
int main(int argc, char** argv)
{
 CArray2X a(2, 5);
 stk_cout << "a.randUnif() =\n"   <<  a.randUnif();
 stk_cout << "a.randGauss()=\n" << a.randGauss();
 Law::Gamma law(1, 1.5);
 stk_cout << "a.rand(law)  =\n" << a.rand(law);
 stk_cout << "a.setValue(1)=\n" << a.setValue(1);
 stk_cout << "a+=2 =\n" << (a+=2);
 stk_cout << "a*=2 =\n" << (a*=2);
 stk_cout << "a/=2 =\n" << (a/=2);
 stk_cout << "a-=2 =\n" << (a-=2);
 return 0;
}
```

Listing 13: Output

```
a.randUnif() =
 0.137596  0.88743  0.169496 0.368936 0.647935
0.0491916 0.203792 0.0451589 0.117904 0.866746
a.randGauss()=
-0.381627  1.89977   1.56497 1.48796 -0.136965
 0.993396 0.125887 -0.575037 2.36021  -1.04943
a.rand(law)  =
0.099326  0.817579 0.702912 0.937509 0.813065
  3.2304 0.0500886  2.33403  1.48067  3.72204
a.setValue(1)=
1 1 1 1 1
1 1 1 1 1
a+=2 =
3 3 3 3 3
3 3 3 3 3
a*=2 =
6 6 6 6 6
6 6 6 6 6
a/=2 =
3 3 3 3 3
3 3 3 3 3
a-=2 =
1 1 1 1 1
1 1 1 1 1
```

5.4 Functors

Functors can fundamentally be used in the same way that slicing visitors. The only difference is that they return an `Array2DPoint` if they are applied by column or an `Array2DVector` if they are applied by rows. Recall that it is possible to use the `move` method in order to store the results of the computation without data copy.

All the functors are in the `Stat` namespace and compute the usual statistics by columns or rows. If there is the possibility of missing/NaN values add the word Safe to the name of the functor. If the mean by row is needed, just add ByRow to the name of the functor. If you want both add SafeByRow.

Listing 14: Functors

```
#include "STKpp.h"
using namespace STK;
int main(int argc, char** argv)
{
 CArray3X a(3,5);  a.randGauss();
 stk_cout << "Stat::mean(a)=\n"     << Stat::mean(a);
 stk_cout << "Stat::meanByRow(a)=\n"<< Stat::meanByRow
      (a);
 stk_cout << "Stat::variance(a)=\n" << Stat::variance(
      a);
 // compute the biased variance (divided by N=3) with
      mean fixed to 0
 stk_cout << "Stat::varianceWithFixedMean(a,0,false)=\
      n"
          << Stat::varianceWithFixedMean(a, Const::
             VectorX(5)*0, false);
 a(1,2) = a(2,2) = Arithmetic<Real>::NA();
 stk_cout << "Stat::meanSafe(a)=\n"      << Stat::
      meanSafe(a);
 stk_cout << "Stat::meanSafeByRow(a)=\n" << Stat::
      meanSafeByRow(a);
 return 0;
}
```

Listing 14: Output

```
Stat::mean(a)=
-0.911327 0.0749711 1.16178 0.318061  -0.42087
Stat::meanByRow(a)=
 0.360791
-0.369855
 0.142633
Stat::variance(a)=
0.00584228 0.45772 0.472917 1.84638 0.821848
Stat::varianceWithFixedMean(a,0,false)=
0.00584228 0.45772 0.472917 1.84638 0.821848
Stat::meanSafe(a)=
-0.911327 0.0749711 1.76343 0.318061  -0.42087
Stat::meanSafeByRow(a)=
 0.360791
-0.842989
 0.128485
```

6 Slicing arrays and expressions

This part explains the slicing operations. A slice is a rectangular part of an array or expression. Arrays slices can be used both as `rvalue` and as `lvalue` while Expression slice can only be used as `rvalue`.

6.1 Rows and Columns

Individual columns and rows of arrays and expressions can be accessed using `ExprBase::col()` and `ExprBase::row()` as `rvalue` and using `ArrayBase::col()` and `ArrayBase::row()` methods as `lvalue`. The argument for `col()` and `row()` is the index of the column or row to be accessed.

Listing 15: Rows and columns slice

```
#include "STKpp.h"
using namespace STK;
int main(int argc, char *argv[])
{
  CArrayXX A(4, 6); CArray2X B(2,4, -1.);
  A << 1, 2, 3, 4, 1, 2,
       4, 3, 1, 3, 2, 4,
       1, 3, 4, 2, 1, 4,
       2, 3, 1, 4, 3, 2;
  stk_cout << _T("A =\n") << A;
  stk_cout << _T("B =\n") << B;
  A.sub(B.rows(), B.cols()) = B;  // copy B in A
       (0:1,0:3)
  stk_cout << _T("A =\n") << A;
  A.row(3,B.cols()) = meanByCol(B); // copy a row-
       vector in row 3 of A
  stk_cout << _T("A =\n") << A;
}
```

Listing 15: Output

```
A =
1 2 3 4 1 2
4 3 1 3 2 4
1 3 4 2 1 4
2 3 1 4 3 2
B =
-1 -1 -1 -1
-1 -1 -1 -1
A =
-1 -1 -1 -1 1 2
-1 -1 -1 -1 2 4
 1  3  4  2 1 4
 2  3  1  4 3 2
A =
-1 -1 -1 -1 1 2
-1 -1 -1 -1 2 4
 1  3  4  2 1 4
-1 -1 -1 -1 3 2
```

6.2 sub-Arrays and sub-Vectors

The most general sub operation is named `sub()`. There are two versions, one for arrays, an other for vectors. The sub-Arrays and sub-vectors can be employed as `rvalue` or `lvalue`, meaning you can assign or reference a sub-array or sub-vector.

Listing 16: sub arrays and vectors

```
#include "STKpp.h"
using namespace STK;
int main(int argc, char** argv)
{
  VectorX b(3, 0);
  std::cout << "b=\n" << b;
  b.sub(Range(2)) = 1.;
  std::cout << "b=\n" << b << "\n";
  ArrayXX a(3, 4); a << 1.,2.,3.,4.
                      , 1.,2.,3.,4.
                      , 1.,1.,1.,1.;
  std::cout << "a=\n" << a;
  a.col(1) = b;
  a.row(1, Range(2,2)) = 0.;
  a.sub(Range(1,2), Range(2,2)) = 9.;
  std::cout << "a=\n" << a;
  return 0;
}
```

Listing 16: Output

```
b=
0
0
0
b=
1
1
0

a=
1 2 3 4
1 2 3 4
1 1 1 1
a=
1 1 3 4
1 1 9 9
1 0 9 9
```

7 Using reference arrays

This part explains how to create reference on existing arrays.

7.1 Creating reference to sub-arrays

Any slice of an existing array can be referenced by an other array and modified using this "reference" container. This is achieved using the copy constructor with reference as `true`.

Listing 17: Referencing sub-arrays

```
#include "STKpp.h"
using namespace STK;
/** @ingroup tutorial */
int main(int argc, char** argv)
{
  ArraySquareX a(4); a << 1.,2.,3.,4.
                        , 1.,2.,3.,4.
                        , 1.,1.,1.,1.
                        , 1.,1.,1.,1.;
  // create a reference
  ArrayXX b(a.sub(Range(2), Range(3)), true);
  b = -1.;
  std::cout << "Modified a=\n" << a << "\n";

  CArray<Real, 3, 4> c; c << 1.,2.,3.,4.
                           , 1.,2.,3.,4.
                           , 1.,1.,1.,1.;
  // create a reference
  CVector3 d(c.col(1), true);
  d = -1.;
  std::cout << "Modified c=\n" << c;
  return 0;
}
```

Listing 17: Output

```
Modified a=
-1 -1 -1 4
-1 -1 -1 4
 1  1  1 1
 1  1  1 1

Modified c=
1 -1 3 4
1 -1 3 4
1 -1 1 1
```

This method allows to get a safe access to some sub-part of a big array that you want to modify.

Warning:
It is the responsibility of the user to verify that a reference is valid.

7.2 Creating reference to raw data

It is possible to map a single pointer `Type*` or a double pointer `Type**` in respectively `CArray` and `Array2D` using the dedicated constructors.

In this constructor, we wrap the data pointed by `q` in an array with rows in the range I and columns in the range J.

Listing 18: Wrapping raw data in Array2D

```
Array2D<Type>( Type** q, Range const& I, Range const& J);
```

In this example we create an array of size (10,5) and wrap it in an `Array2D`.

Listing 19: Example of wrapping by Array2D

```
Real** q = new Real*[10];
for (int j=0; j<10; j++) { q[j] = new Real[5];}
//wrap q. Note that q will not be freed when a is deleted
Array2D<Real> a(q, Range(0,10), Range(0,5));
```

In this constructor, we wrap the data pointed by `q` in an array with rows and columns of size m and n.

Listing 20: Wrapping raw data in CArray

```
CArray<Type>( Type* q, int m, int n);
```

In this example we create an array of size (10, 5) and wrap it in a `CArray`.

```
Real* q = new Real[10];
for (int j=0; j<10; j++) { q[j] = j;}
//wrap q in a fixed size array of size (2,5).
// Note that q will not be freed when a is deleted
CArray<Real, 2, 5> a(q, 2, 5);
```