Quick Reference guide of the STK++ Arrays

Serge Iovleff

December 14, 2023

Abstract

This quick reference guide resume all possibilities available for arrays, vectors, points (row-vectors) and expressions.

1 R wrappers

The R structure SEXP can be wrapped and used like any other STK++ array using the following template classes

```
template <typename Type > class RVector;
template <typename Type > class RMatrix;
```

The constructors for these objects are the following

```
/** Default Constructor. */
RVector<Type>();
/** Constructor with given dimension. */
RVector<Type>(int length);
/** Constructor with SEXP. */
RVector<Type>( SEXP robj);
/** Constructor with Rcpp vector*/
RVector<Type>( Rcpp::Vector<Rtype_> vector)
/** Copy Constructor. @c ref is only there for compatibility.
 *  By default all R object are copied by reference.
 **/
RVector<Type>( RVector robj, bool ref);

/** Default Constructor */
RMatrix<Type>();
/** Constructor with given dimension */
RMatrix(int nRow, int nCol);
/** Constructor with SEXP */
RMatrix<Type>( SEXP robj);
/** Copy constructor */
RMatrix<Type>( Rcpp::Matrix<Rtype_> matrix)
/** Copy Constructor. @c ref is only there for compatibility.
 *  By default all R object are copied by reference.
 **/
RMatrix<Type>( RMatrix const& matrix, bool ref)
```

All these arrays can be converted back as SEXP using the methods

```
  return v.vector(); // return wrapped SEXP
  return m.matrix(); // return wrapped SEXP
```

2 STK++ Arrays and Vectors

2.1 Containers

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. Thes containers are grouped in two families of arrays : CArray and Array2D. All theses classes are in namespace STK.

Data can be stored in one of the following array

```
typedef CArray<Type, SizeRows, SizeCols, Orient> MyCArray;
typedef CArraySquare<Type, Size, Orient> MyCSquare;
typedef CArrayVector<Type, SizeRows, Orient> MyCVector;
typedef CArrayPoint<Type, SizeCols, Orient> MyCPoint;

typedef Array2D<Type> MyArray2D;
typedef Array2DVector<Type> MyVector2D;
typedef Array2DPoint<Type> MyPoint2D;
typedef Array2DUpperTriangular<Type> MyUpperTriangular2D;
typedef Array2DLowerTriangular<Type> MyLowerTriangular2D;
typedef Array2DDiagonal<Type> MyDiagonal2D;
```

- `Type` is the type of the elements like `double`, `float`, etc.
- `Orient` can be either `STK::Arrays::by_col_` (= 1, the default template) or `Arrays::by_row_` (= 0)
- If you don't know at compile time the size of your array/vector just use `STK::UnknownSize` (the default template).

> **Note:**
> Only the first template argument is mandatory in `CArray` family.

2.2 Constant Arrays

It is possible to use constant arrays with all values equal to 1 using the type predefined in the `namespace STK::Const`

```
Const::Identity<Type, 10> i; // size fixed at compile time
Const::Identity<Type> i(10); // size fixed at runtime
Const::Square<Type, Size> s;
Const::Square<Type> s(10);
Const::Vector<Type, Size> v;
Const::Point<Type, Size> p;
Const::Array< Type, SizeRows, SizeCols> a;
Const::Array< Type> a(10, 20);
Const::UpperTriangular< Type, SizeRows, SizeCols> u;
Const::LowerTriangular< Type, SizeRows, SizeCols> l;
```

As usual, only the first template parameter is mandatory.

3 Manipulating Arrays and Vectors

3.1 Basic operations

Constructors are detailed in the document Arrays Constructors. After creation `arrays` can be initialized using comma initializer

```
CVector3  v;     v << 1, 2, 3;
CSquareXX m(3);  m << 1, 2, 3,
                      2, 3, 4,
                      3, 4, 5;
```

and elements can be accessed using the parenthesis (for arrays), the brackets (for vectors) and the `elt` methods

```
v[0] = 3; v.elt(1) = 1; v.at(2) = 2;        // at(.) check index range
m(0,0) = 5; m.elt(1,1) = 1; m.at(2,2) = 3; // at(.) check indexes range
```

The whole array/vector can be initialized using a value, either at the construction or during execution by using

```
v.setZeros()             // fill v with value 0
m.setOnes();             // fill m with value 1
Array2D<int> a(3, 3, 1); // arrays of size (3,3) with all the elements equal to 1
a.setValue(2);           // fill a with value 2
```

3.2 Getting parts of Arrays and Vectors

It is possible to access to a row, a column, a sub-part of an array using slicing operators.

Listing 1: Slicing

```
m.row(i);              // get the ith row
m.col(j);              // get the jth row
m.row(Trange<4>(3,4)); // get sub-array m[3:6,]
m.col(Trange<4>(1,4)); // get sub-array m[,1:3]
m.sub(Trange<4>(3,4), Trange<4>(1,4)); // get sub-array m[3:6,1:3]
```

3.3 Arrays and vectors basic informations

For all the containers it is possible to get the status (reference or array owning its data) the range, the beginning, the end and the size using

```
CArrayXX m(3,4);
bool mf = m.isRef(); // mf is false

typename CArrayXX::ColRange mc= m.cols();
int mbc= m.beginCols(), mec= m.endCols(), msc= m.sizeCols();

typename CArrayXX::RowRange mr= m.rows();
int mbr= m.beginRows(), mer= m.endRows(), msr= m.sizeRows();

CArrayVectorX v(m.col(0), true); // v is a reference on the column 0 of m
bool vf = v.isRef(); // vf is true

CArrayVectorX::RowRange vr= m.range();
int vb= m.begin(), ve= m.end(), vs= m.size();
```

For the `Array2D` family of container, it is also possible to get informations about the allocated memory of the containers. It can be interested in order to known if the container will have to reallocate the memory if you try to resize it.

```
ArrayXX m(3,4);
m.availableCols();                  // number of available columns
Array1D<int> a = m.availableRows(); // vector with the number of available rows of each
    column

m.capacityCol(1);                   // capacity of the column 1
Array1D< Range > r = m.rangeCols () // vector with used range of each columns (think to
    triangular matrices)
```

The `Array2D` family of container allocate a small amount of supplementary memory so that in case of a `resize`, it is possible to expand the container without data transfer.

3.4 Visitors and Appliers

Visitors and visit to an array/vector/point/expression and are automatically unrolled if the array is of fixed (small) size.

Listing 2: Visitors

```
int  i= m.count();          // count the number of true values
bool f= m.any();            // is there any true value ?
bool f= m.all();            //
Type r= m.minElt(i,j);      // can be v.minElt(i) or m.minElt()
Type r= m.maxElt(i,j);      // can be v.maxElt(i) or m.maxElt()
Type r= m.minEltSafe(i,j); // can be v.minEltSafe(i) or m.minEltSafe()
Type r= m.maxEltSafe(i,j); // can be v.maxEltSafe(i) or m.maxEltSafe()
Type r= m.sum(),  s= m.sumSafe();
Type r= m.mean(), s= m.meanSafe();
```

Appliers modifies the values of the container and thus cannot be used with expressions.

Listing 3: Appliers

```
m.randUnif();  // fill m with uniform random numbers between 0 and 1
m.randGauss(); // fill m with standardized Gaussian random numbers
m.setOnes();   // fill m with value 1
m.setValues(2);// fill m with value 2
m.setZeros();  // fill m with value 0
Law::Gamma law(1,2); // create a Gamma distribution
m.rand(law);         // fill m with gamma(1,2) random numbers
```

The next methods use visitors in order to compute the result, eventually safely

```
m.norm();  m.normSafe();
m.norm2(); m.norm2Safe(); // norm without sqrt
m.normInf();              // superior norm
m.variance(); m.varianceSafe();
// variance with fixed mean
m.variance(mean); m.varianceSafe(mean);
// weighted visitors
m.wsum(weights);  m.wsumSafe(weights);
m.wnorm(weights);  m.wnormSafe(weights);
m.wnorm2(weights); m.wnorm2Safe(weights);
m.wmean(weights);  m.wmeanSafe(weights);
m.wvariance(weights); m.wvarianceSafe(weights);
m.wvariance(mean, weights); m.wvarianceSafe(mean, weights);
```

4 Functors on Arrays/Vectors and expressions

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 a safe computation by row add SafeByRow.

All functors applied on an array will return a number if applied on a vector or a point (a row-vector), a `STK::Array2DPoint<Type>` if applied by column on an array, and a `STK::Array2DVector<Type>` if applied by row. `Type` is the type of the data stored in the array.

4.1 Statistical Functors

All the functors applied on arrays are currently in the `namespace STK::Stat`.

Listing 4: Statistical functors by column

```
Stat::min(a); Stat::minSafe(a); Stat::min(a, w); Stat::minSafe(a, w);
Stat::max(a); Stat::maxSafe(a); Stat::max(a, w); Stat::maxSafe(a, w);
Stat::sum(a); Stat::sumSafe(a); Stat::sum(a, w); Stat::sumSafe(a, w);
Stat::mean(a); Stat::meanSafe(a); Stat::mean(a, w); Stat::meanSafe(a, w);
// could also be
Stat::minByCol(a); Stat::minSafeByCol(a); Stat::minByCol(a, w); Stat::minSafeByCol(a, w);
// .. etc
// unbiased variance (division by n-1) when unbiased is true, false is the default
Stat::variance(a, false); Stat::varianceSafe(a, false);
Stat::variance(a, w, false); Stat::varianceSafe(a, w, false);
// fixed mean. Must be a vector/point for arrays and a number for vectors/points
// unbiased (false in examples below) has to be given
Stat::varianceWithFixedMean(a, mean, false);
Stat::varianceWithFixedMean(a, w, mean, false);
Stat::varianceWithFixedMeanSafe(a, mean, false);
Stat::varianceWithFixedMeanSafe(a, w, mean, false);
```

Listing 5: Statitica functors by row

```
Stat::minByRow(a); Stat::minSafeByRow(a); Stat::minByRow(a, w); Stat::minSafeByRow(a, w);
Stat::maxByRow(a); Stat::maxSafeByRow(a); Stat::maxByRow(a, w); Stat::maxSafeByRow(a, w);
Stat::sumByRow(a); Stat::sumSafeByRow(a); Stat::sumByRow(a, w); Stat::sumSafeByRow(a, w);
Stat::meanByRow(a); Stat::meanSafeByRow(a); Stat::meanByRow(a, w); Stat::meanSafeByRow(a, w)
    ;
// unbiased variance (division by n-1) when unbiased is true, false is the default
Stat::varianceByRow(a, false); Stat::varianceSafeByRow(a, false);
Stat::varianceByRow(a, w, false); Stat::varianceSafeByRow(a, w, false);
// fixed mean. Must be a vector/point for arrays and a number for vectors/points
// unbiased (false in examples below) has to be given
Stat::varianceWithFixedMeanByRow(a, mean, false);
Stat::varianceWithFixedMeanByRow(a, w, mean, false);
```

```
Stat::varianceWithFixedMeanSafeByRow(a, mean, false);
Stat::varianceWithFixedMeanSafeByRow(a, w, mean, false);
```

5 Arithmetic Operations on arrays/vectors and expressions

Available operations on arrays/vectors and expressions are summarized in the next table. Most operations are similar to the operations furnished by the Eigen library .

5.1 Operation between arrays/expressions/numbers

Listing 6: add, subtract, divide, multiply arrays element by element

```
m= m1+m2; m+= m1;
m= m1-m2; m-= m1;
m= m1/m2; m/= m1;
m= m1.prod(m2); // don't use m1*m2 if you want a product element by element
```

Listing 7: add, subtract, divide, multiply by a number

```
Real s;
m= m1+s; m= s+m1; m+= s;
m= m1-s; m= s-m1; m-= s;
m= m1/s; m= s/m1; m/= s;
m= m1*s; m= s*m1; m*= s;
m= m1%s; m= s%m1; m%= s; // work only with integers
bool f;
m= m1&&f; m= m1||f;       // logical operations
int r;
m= m1&r; m=m1|r; m=m1^r; // bitwise operations, work only with integers
```

Listing 8: matrix by matrix/vector products

```
CArrayXX m2(5,4), m1(4,5), m; PointX p2(5), p1(4), p; VectorX v2(4), v1(5), v;
Real s= p2*v1; // dot product
v= m2*v2; v= m1p2.transpose();  // get a vector
p= p2*m2; p= v1.transpose()*m2; // get a point (row-vector)
m= m2*m1; m= m1.transpose()*m;  // matrix multiplication
m= m2.prod(m1.transpose());     // product element by element
m2*= m1;                        // m2 will be resized and filled with m2*m1 product
```

Listing 9: Comparisons operators

```
Real s;
// count for each columns the number of true comparisons
(m1 < m2).count();  (m1 > m2).count();  (m1 < s).count();  (m1 > s).count();
(m1 <= m2).count(); (m1 >= m2).count(); (m1 <= s).count(); (m1 >= s).count();
(m1 == m2).count(); (m1 != m2).count(); (m1 == s).count(); (m1 != s).count();
```

5.2 Operations on Arrays

All these operations return an array of the same size from the original array.

Listing 10: Probabilities and related operations

```
Law::Normal l(0,1);
// these methods return an array of the same size as m
m.pdf(l);   // compute pdf values to m using distribution l
m.lpdf(l);  // compute log-pdf values to m using distribution l
```

```
m.cdf(l);   // compute cdf values using distribution l
m.lcdf(l);  // compute log-cdf values using distribution l
m.cdfc(l);  // compute complementary cdf values using distribution l
m.lcdfc(l); // compute log-complementary cdf values using distribution l
m.icdf(l);  // compute inverse cdf values using distribution l
```

Listing 11: Mathematical functions

```
m.isNa();        // boolean expression with true if m(i,j) is a NA value
m.isFinite();    // boolean expression with true if m(i,j) is a finite value
m.isInfinite(); // boolean expression with true if m(i,j) is an infinite value
m.min(m2);       // Type expression with min(m(i,j), m2(i,j))
m.max(m2);       // Type expression with max(m(i,j), m2(i,j))
m.prod(m2);      // Type expression with m(i,j)*m2(i,j)
m.neg();         // Type expression with !m(i,j)
m.abs();         // Type expression with abs(m(i,j))
m.sqrt();        // Type expression with sqrt(m(i,j))
m.log();         // Type expression with log(m(i,j))
m.exp();         // Type expression with exp(m(i,j))
m.pow(number);  // Type expression with m(i,j)^number
m.square();      // Type expression with m(i,j)*m(i,j)
m.cube();        // Type expression with m(i,j)*m(i,j)*m(i,j)
m.inverse();     // Type expression with 1./m(i,j)
m.sin();         // Type expression with sin(m(i,j))
m.cos();         // Type expression with cos(m(i,j))
m.tan();         // Type expression with tan(m(i,j))
m.asin();        // Type expression with asin(m(i,j))
m.acos();        // Type expression with acos(m(i,j))
```

Listing 12: Miscellaneous functions

```
m.cast<OtherType>();        // expression with static_cast<OtherType>(m(i,j)))
m.funct0<Functor>();        // expression with Functor(m(i,j)))
m.funct1<Functor>(functor); // same thing with an instance of Functor given
```

6 Reshaping Arrays

These operations does not copy data but transform the structure of the array.

Listing 13: Reshaping operations

```
m.transpose(); //transpose m
v.diagonalize(); // transform a vector/point to a diagonal matrix
m.getDiagonal(); // get diagonal of a square matrix
m.upperTriangularize(); // transform m to a upper triangular matrix (only upper part is used
    )
m.lowerTriangularize(); // transform m to a lower triangular matrix (only lower part is used
    )
m.symmetrize();         // m is known to be symmetric
m.uppersymmetric(); // m is symmetric and stored in the upper part
m.lowersymmetric(); // m is symmetric and stored in the lower part
```

7 Convenience typedef

There exists predefined typedef for the arrays that can be used. The predefined type STK::Real can be either @c double (the default) or @c float. STK::Real as float can be enabled at compile time by the directive `-DSTKREALAREFLOAT`.

7.1 CArray

Listing 14: CArray family

```
// CArray with Real (double by default) data
typedef CArray<Real, UnknownSize, UnknownSize, Arrays::by_col_> CArrayXX;
typedef CArray<Real, UnknownSize, 2, Arrays::by_col_>           CArrayX2;
typedef CArray<Real, UnknownSize, 3, Arrays::by_col_>           CArrayX3;
typedef CArray<Real, 2, UnknownSize, Arrays::by_col_>           CArray2X;
typedef CArray<Real, 3, UnknownSize, Arrays::by_col_by>         CArray3X;
typedef CArray<Real, 2, 2, Arrays::by_col_>                     CArray22;
typedef CArray<Real, 3, 3, Arrays::by_col_>                     CArray33;

// CArray with double data (add d to the type name)
typedef CArray<double, UnknownSize, UnknownSize, Arrays::by_col_>CArrayXXd;
...
// CArray with int data (add i to the typename)
typedef CArray<int, UnknownSize, UnknownSize, Arrays::by_col_>  CArrayXXi;
...
// Arrays with data stored by rows, the same as above with ByRow added
typedef CArray<Real, UnknownSize, UnknownSize, Arrays::by_row_> CArrayByRowXX;
...
// CArraySquare (like CArray with the same number of rows and columns)
typedef CArraySquare<Real, UnknownSize, Arrays::by_col_>   CSquareX;
typedef CArraySquare<Real, 2, Arrays::by_col_>             CSquare2;
..
typedef CArraySquare<int, 3, Arrays::by_col_>              CSquare3i;
...
typedef CArraySquare<Real, UnknownSize, Arrays::by_row_>   CSquareByRowX;
```

Some of the predefined typedef for the `Array2D` class are given hereafter

Listing 15: Array2D family

```
  typedef Array2D<Real>    ArrayXX;
  typedef Array2D<double>  ArrayXXd;
  typedef Array<int>       ArrayXXi;
  typedef Array<float>     ArrayXXf;
```

For the other kind of containers there exists also predefined types

Listing 16: CArrayVector and CArrayPoint family

```
typedef CArrayVector<Real, UnknownSize, Arrays::by_col_>   CVectorX;
typedef CArrayVector<Real, 2, Arrays::by_col_>             CVector2;
typedef CArrayVector<Real, 3, Arrays::by_col_>             CVector3;
typedef CArrayVector<double, UnknownSize, Arrays::by_col_> CVectorXd;
typedef CArrayVector<double, 2, Arrays::by_col_>           CVector2d;
typedef CArrayVector<double, 3, Arrays::by_col_>           CVector3d;
typedef CArrayVector<int, UnknownSize, Arrays::by_col_>    CVectorXi;
typedef CArrayVector<int, 2, Arrays::by_col_>              CVector2i;
typedef CArrayVector<int, 3, Arrays::by_col_>              CVector3i;

// CArrayPoint
typedef CArrayPoint<Real, UnknownSize, Arrays::by_col_>   CPointX;
typedef CArrayPoint<Real, 2, Arrays::by_col_>             CPoint2;
typedef CArrayPoint<Real, 3, Arrays::by_col_>             CPoint3;
...
typedef CArrayPoint<int, 3, Arrays::by_col_>              CPoint3i;
```

Listing 17: Array2DVector Array2DPoint and Array2DDiagonal

```
  typedef Array2DPoint<Real>     PointX;
  typedef Array2DPoint<double>   PointXd;
  typedef Array2DVector<Real>    VectorX;
  typedef Array2DVector<double>  VectorXd;
  typedef Array2DDiagonal<Real>  ArrayDiagonalX;
  typedef Array2DDiagonal<int>   ArrayDiagonalXi;
```