STK++ Utilities reference guide

Serge Iovleff

July 5, 2017

Abstract

This reference guide gives a general review of the capabilities offered by the STK++ library. The library is divided in various *projects*. The "Arrays" project is described in detail in the vignette "STK++ Arrays, User Guide" ([1]) and the quick reference guide. This vignette focus on the "STatistiK" project and the "Algebra" project.

Contents

1 Statistical functors, methods and functions (STatistiK project)

This section describe the main features provided by the STatistiK project. Mainly

1. the probability classes (1.1),
2. the descriptive statistical methods,
3. the utilities related methods.

1.1 Probabilities

All the probabilities handled by R are available in `rtkore`. In the stand-alone STK++ library, only a subset of theses probabilities are implemented. Probability distribution classes are defined in the `namespace Law` and can be used as in this example

Listing 1: Example

```
#include "STKpp.h"
using namespace STK;
/** @ingroup tutorial */
int main(int argc, char** argv)
{
  Law::Normal l(2, 1);
  stk_cout << "l.cdf(3.96)= "    << l.cdf(3.96)  << _T("\n");
  stk_cout << "l.icdf(0.975)= " << l.icdf(0.975)  << _T("\n");
  stk_cout << "l.pdf(2)= "      << l.pdf(2)  << _T("\n");
  stk_cout << "l.lpdf(2)= "     << l.lpdf(2)  << _T("\n");
  stk_cout << "l.rand()= "      << l.rand()  << _T("\n");
  CArray33 a;
  a.rand(1);
  stk_cout << "a=\n"      << a  << _T("\n");
}
```

Listing 1: Output

```
l.cdf(3.96)= 0.975002
l.icdf(0.975)= 3.95996
l.pdf(2)= 0.398942
l.lpdf(2)= -0.918939
l.rand()= 2.62646
a=
0.272322  0.185485  1.41288
 1.65554   1.87354 0.168805
 2.46967 -0.266933  0.19432
```

All probability distribution classes have a similar prototype like the one given below

Listing 2: Prototype of probability distribution class (example taken from Cauchy class)

```
class Cauchy : public IUnivLaw<Real>
{
  public:
    Cauchy( Real const& mu=0, Real const& scale=1);
    virtual ~Cauchy() {}
    Real const& mu() const;
    Real const& scale() const;
    void setMu( Real const& mu);
    void setScale( Real const& scale);
    virtual Real rand() const;
    virtual Real pdf( Real const& x) const;
    virtual Real lpdf( Real const& x) const;
    virtual Real cdf( Real const& t) const;
    virtual Real icdf( Real const& p) const;
    static Real rand( Real const& mu, Real const& scale);
    static Real pdf( Real const& x, Real const& mu, Real const& scale);
    static Real lpdf( Real const& x, Real const& mu, Real const& scale);
    static Real cdf( Real const& t, Real const& mu, Real const& scale);
    static Real icdf( Real const& p, Real const& mu, Real const& scale);
  protected:
    Real mu_;
    Real scale_;
}
```

If f denote the density of some probability distribution function (pdf) on $\mathbb{R}$, the methods have the following meaning

1. `pdf(x)` return the value $f(x)$,
2. `lpdf(x)` return the value $\log(f(x))$,
3. `rand()` return a a realization of the density f,
4. `cdf(t)` return the cumulative distribution function value $F(t) = \int_{-\infty}^{t} f(x)dx$
5. `icdf(p)` return the inverse cumulative distribution function value $F^{-1}(p)$.

The table 1 gives the list of available probability distribution.

Name	Constructor	R functions	Notes
Bernoulli	`Law::Bernouilli(p)`	-	
Binomial	`Law::Binomial(n,p)`	`*binom`	
Beta	`Law::Beta(alpha,beta)`	`*beta`	
Categorical	`Law::Categorical(p)`	-	p can be any STK++ vector
Cauchy	`Law::Cauchy(m,s)`	`*cauchy`	
ChiSquared	`Law::ChiSquared(n)`	`*chisq`	
Exponential	`Law::Exponential(lambda)`	`*exp`	Parameterization $\frac{e^{-x/\lambda}}{\lambda}$
FisherSnedecor	`Law::FisherSnedecor(df1,df2)`	`*f`	
Gamma	`Law::Gamma(a,b)`	`*gamma`	Parameterization $\frac{x^{a-1}}{\beta^a \Gamma(a)} e^{-x/\beta}$
Geometric	`Law::Geometric(p)`	`*geom`	
HyperGeometric	`Law::HyperGeometric(m,n,k)`	`*hyper`	
Logistic	`Law::Logistic(mu,scale)`	`*logis`	
LogNormal	`Law::LogNormal(mulog,sdlog)`	`*lnorm`	
NegativeBinomial	`Law::NegativeBinomial(size,prob,mu)`	`*nbinom`	
Normal	`Law::Normal(mu,sigma)`	`*norm`	
Poisson	`Law::Poisson(lambda)`	`*poiss`	
Student	`Law::Student(df)`	`*t`	
Uniform	`Law::Uniform(a,b)`	`*unif`	
UniformDiscrete	`Law::UniformDiscrete(a,b)`	-	
Weibull	`Law::Weibull(a)`	`*weibull`	

Table 1: List of the available probability distribution in `rtkore`

1.2 Statistical Methods and global functions

STK++ provides a lot of methods, functions and functors in order to compute usual statistics.

1.2.1 Methods

Let m be any kind of array (square, vector, point, etc...). it is possible to compute the min, max, mean, variance of the elements. These computations can be safe (i.e. discarding N.A. and infinite values) or unsafe and weighted

Method	weigthed version	safe versions	Notes
`m.norm()`	`m.wnorm(w)`	`m.normSafe(); m.wnormSafe(w)`	$\sqrt{\sum \lvert m_{ij}\rvert^2}$
`m.norm2()`	`m.wnorm2(w)`	`m.norm2Safe(); m.wnorm2Safe(w)`	$\sum \lvert m_{ij}\rvert^2$
`m.normInf()`	`m.wnormInf(w)`	`m.normInfSafe(); m.wnormInfSafe(w)`	$\sup \lvert m_{ij}\rvert$
`m.sum()`	`m.wsum(w)`	`m.sumSafe(); m.wsumSafe(w)`	$\sum m_{ij}$
`m.mean()`	`m.wmean(w)`	`m.meanSafe(); m.wmeanSafe(w)`	$\frac{1}{n}\sum m_{ij}$
`m.variance()`	`m.wvariance(w)`	`m.varianceSafe(); m.wvarianceSafe(w)`	$\frac{1}{n}\sum (m_{ij}-\bar{m})^2$
`m.variance(mu)`	`m.wvariance(mu,w)`	`m.varianceSafe(mu); m.wvarianceSafe(mu,w)`	$\frac{1}{n}\sum (m_{ij}-\mu)^2$

Table 2: List of the available statistical methods for the arrays. n represents the number of elements of m. For safe versions, n represents the number of available observations in m.

1.2.2 Statistical functions

For two dimensional arrays, there exists global functions allowing to compute the usual statistics by column or by row. By default all global functions are computing the statistics columns by columns. For example, if m is an array, `sum(m)` return a row-vector of range `m.cols()` containing the sum of each columns. The alias `sumByCol(m)` can also be used. The sum of each rows can be obtained using the function `sumByow(m)` which return an array of range `m.rows()`.

These computations can be safe (i.e. discarding N.A. and infinite values) or unsafe and/or weighted.

Function	weigthed version	safe versions
`min(m)`	`min(m, w)`	`minSafe(m/*,w*/)`
`minByCol(m)`	`minByCol(m, w)`	`minSafeByCol(m/*,w*/)`
`minByRow(m)`	`minByRow(m, w)`	`minSafeByRow(m/*,w*/)`
`max(m)`	`max(m, w)`	`maxSafe(m/*,w*/)`
`maxByCol(m)`	`maxByCol(m, w)`	`maxSafeByCol(m/*,w*/)`
`maxByRow(m)`	`maxByRow(m, w)`	`maxSafeByRow(m/*,w*/)`
`sum(m)`	`sum(m, w)`	`sumSafe(m/*,w*/)`
`sumByCol(m)`	`sumByCol(m, w)`	`sumSafeByCol(m/*,w*/)`
`sumByRow(m)`	`sumByRow(m, w)`	`sumSafeByRow(m/*,w*/)`
`mean(m)`	`mean(m, w)`	`meanSafe(m/*,w*/)`
`meanByCol(m)`	`meanByCol(m, w)`	`meanSafeByCol(m/*,w*/)`
`meanByRow(m)`	`meanByRow(m, w)`	`meanSafeByRow(m/*,w*/)`
`variance(m, unbiased)`	`variance(m, w, unbiased)`	`varianceSafe(m/*,w*/, unbiased)`
`varianceByCol(m, unbiased)`	`varianceByCol(m, w, unbiased)`	`varianceSafeByCol(m/*,w*/, unbiased)`
`varianceByRow(m, unbiased)`	`varianceByRow(m, w, unbiased)`	`varianceSafeByRow(m/*,w*/, unbiased)`
`varianceWithFixedMean(m, mu, unbiased)`	`variance*(m, w, mu, unbiased)`	`variance*Safe(m/*,w*/, mu, unbiased)`
`varianceWithFixedMeanByCol(m, mu, unbiased)`	`variance*ByCol(m, w, mu, unbiased)`	`variance*SafeByCol(m/*,w*/, mu, unbiased)`
`varianceWithFixedMeanByRow(m, mu, unbiased)`	`variance*ByRow(m, w, mu, unbiased)`	`variance*SafeByRow(m/*,w*/, mu, unbiased)`

Table 3: List of the available global statistical functions for arrays. m is the array, w the vector of weights. `unbiased` is a Boolean to set to `true` if unbiased variance (divided by $n-1$) is desired.

The covariance can be computed in two ways : using two vectors of same size, or using all the columns (rows) of a two-dimensional array. In the first case the functions return the value of the covariance, in the second case, they return a `CSquareArray`.

Function	weigthed version	safe versions
`covariance(v1, v2, unbiased)`	`covariance(v1, v2, w, unbiased)`	`covarianceSafe(v1, v2, unbiased)` `covarianceSafe(v1, v2, w, unbiased)`
`covarianceWithFixedMean(v1, v2, mean, unbiased)`	`covariance*(v1, v2, w, mean, unbiased)`	`covariance*Safe(v1, v2, unbiased)`
`covariance(m, unbiased)` `covarianceByRow(m, unbiased)`	`covariance(m, w, unbiased)` `covarianceByRow(m, w, unbiased)`	
`covarianceWithFixedMean(m, mean, unbiased)` `covarianceWithFixedMeanByRow(m, mean, unbiased)`	`covariance*(m, w, mean, unbiased)` `covariance*ByRow(m, w, mean, unbiased)`	

Table 4: List of the available covariance functions for vectors and arrays. v_1 and v_2 are vectors, m is an array, w a vector of weights. `unbiased` is a Boolean to set to `true` if unbiased covariance (divided by $n-1$) is desired. The first set of covariance functions return a scalar, the second set of covariance functions return a `CSquareArray`

The following example illustrate the use of the covariance function:

Listing 3: Example

```
#include "STKpp.h"
using namespace STK;
/** @ingroup tutorial */
int main(int argc, char** argv)
{
  // create coovariance matrix and its Cholesky decomposition
  CArraySquare<Real, 3> s; s << 2.0, 0.8, 0.36,
                                0.8, 2.0, 0.8,
                                0.36,0.8, 1.0;
  Array2DLowerTriangular<Real> L; Array2DDiagonal<Real> D;
  cholesky(s, D, L);
  // create correlated data set
  CArray<Real, 100, 3> a;
  a = a.randGauss() * D.sqrt() * L.transpose() + 1;
  stk_cout << "True sigma=\n" << s;
  stk_cout << "Estimated sigma=\n" << Stat::covariance(a);
  return 0;
}
```

Listing 3: Output

```
True sigma=
   2 0.8 0.36
 0.8   2  0.8
0.36 0.8    1
Estimated sigma=
 1.87474 0.809571 0.291616
0.809571  2.22951 0.890907
0.291616 0.890907  1.10475
```

1.3 Miscellaneous statistical functions

Given an array m, it is possible to center it, to standardize it and to perform the reverse operations as illustrated in the following exemple

Listing 4: Example

```
#include "STKpp.h"
using namespace STK;
int main(int argc, char** argv)
{
  CArray<Real, 5, 8> A;
  CArrayPoint<Real, 8> mu, std, aux;
  Law::Normal law(1,2);
  A.rand(law);
  stk_cout << _T("mean(A) =") << (mu=Stat::mean(A));
  stk_cout << _T("variance(A) =") << Stat::varianceWithFixedMean(A, mu, false)
      << _T("\n\n");
  Stat::standardize(A, mu, std);
  stk_cout << _T("mean(A) =") << (aux=Stat::mean(A));
  stk_cout << _T("variance(A) =") << Stat::varianceWithFixedMean(A, aux, false)
       << _T("\n\n");
  Stat::unstandardize(A, mu, std);
  stk_cout << _T("mean(A) =") << (aux=Stat::mean(A));
  stk_cout << _T("variance(A) =") << Stat::varianceWithFixedMean(A, aux, false)
       << _T("\n\n");

  return 0;
}
```

Listing 4: Output

```
mean(A) =1.17981 2.05352 0.339851
     1.25085 0.760685 0.878685 0.722511
     0.0667084
variance(A) =1.64512 2.2913 3.95029
     3.88707 8.99953 1.1543 1.38419
     4.13585


mean(A) =5.55112e-17 -2.22045e-17
     -2.22045e-17 0 3.26128e-17 2.22045e
     -17 0 8.88178e-17
variance(A) =1 1 1 1 1 1 1 1


mean(A) =1.17981 2.05352 0.339851
     1.25085 0.760685 0.878685 0.722511
     0.0667084
variance(A) =1.64512 2.2913 3.95029
     3.88707 8.99953 1.1543 1.38419
     4.13585
```

Function	weigthed version
`center(m, mean)`	`center(m, w, mean)`
`centerByCol(m, mean)`	`centerByCol(m, w, mean)`
`centerByRow(m, mean)`	`centerByRow(m, w, mean)`
`standardize(m, std, unbiased)`	`standardize(m, w, std, unbiased)`
`standardizeByCol(m, std, unbiased)`	`standardizeByCol(m, w, std, unbiased)`
`standardizeByRow(m, std, unbiased)`	`standardizeByRow(m, w, std, unbiased)`
`standardize(m, mean, std, unbiased)`	`standardize(m, w, mean, std, unbiased)`
`standardizeByCol(m, mean, std, unbiased)`	`standardizeByCol(m, w, mean, std, unbiased)`
`standardizeByRow(m, mean, std, unbiased)`	`standardizeByRow(m, w, mean, std, unbiased)`
`uncenter(m, mean)`	
`uncenterByCol(m, mean)`	
`uncenterByRow(m, mean)`	
`unstandardize(m, std)`	
`unstandardizeByCol(m, std)`	
`unstandardizeByRow(m, std)`	
`unstandardize(m, mean, std)`	
`unstandardizeByCol(m, mean, std)`	
`unstandardizeByRow(m, mean, std)`	

Table 5: List of the available utilites functions for centering and/or standardize an array. m is an array, w a vector of weights. If used on the columns, `mean` and `std` have to be points (row-vectors), if used by rows, `mean` and `std` have to be vectors.`unbiased` is a boolean to set to `true` if unbiased covariance (divided by $n-1$) is desired.

1.4 Computing factors

Given a finite collection of object in a vector or an array/expression, it is possible to encode it as factor using the classes `Stat::Factor` (for vectors) and `Stat::MultiFactor` (for array). These classes are runners and you have to use the `run` method in order to trigger the computation of the factors.

An exemple is given below

Listing 5: Example

```
#include "STKpp.h"
using namespace STK;
int main(int argc, char** argv)
{
  CArray<char, 13, 3> datai;
  datai << 'b', 'a', 'a',   'c', 'b', 'a',   'b', 'a', 'a',   'c', 'a', 'a',
       'd', 'a', 'a',
            'b', 'b', 'a',   'd', 'c', 'b',   'c', 'c', 'b',   'b', 'c', 'b',
                 'd', 'b', 'b',
            'b', 'a', 'b',   'd', 'a', 'b',   'c', 'c', 'b';

  Stat::Factor<CArrayVector<char, 13> > f1d(datai.col(1));
  f1d.run();
  stk_cout << _T("nbLevels= ")  << f1d.nbLevels() << _T("\n");
  stk_cout << _T("asInteger= ") << f1d.asInteger().transpose() << _T("\n");
  stk_cout <<_T("Levels: ")     << f1d.levels().transpose();
  stk_cout <<_T("Levels counts: ") << f1d.counts().transpose();

  Stat::MultiFactor<CArray<char, 13, 3> > f2d(datai);
  f2d.run();
  stk_cout << _T("nbLevels= ")   << f2d.nbLevels() << _T("\n");
  stk_cout << _T("asInteger=\n") << f2d.asInteger() << _T("\n");
  for (int i=f2d.levels().begin(); i<f2d.levels().end(); ++i)
  { stk_cout <<_T("Levels variable ")<< i << _T(": ")<< f2d.levels()[i].
       transpose();}
  stk_cout << _T("\n");
  for (int i=f2d.levels().begin(); i<f2d.levels().end(); ++i)
  { stk_cout <<_T("Levels counts ")<< i << _T(": ")<< f2d.counts()[i].transpose
       ();}
  stk_cout << _T("\n");

 return 0;
}
```

Listing 5: Output

```
nbLevels= 3
asInteger= 0 1 0 0 0 1 2 2 2 1 0 0 2

Levels: a b c
Levels counts: 6 3 4
nbLevels= 3 3 2

asInteger=
0 0 0
1 1 0
0 0 0
1 0 0
2 0 0
0 1 0
2 2 1
1 2 1
0 2 1
2 1 1
0 0 1
2 0 1
1 2 1

Levels variable 0: b c d
Levels variable 1: a b c
Levels variable 2: a b

Levels counts 0: 5 4 4
Levels counts 1: 6 3 4
Levels counts 2: 6 7
```

2 Linear Algebra classes, methods and functions

STK++ basic linear operation as product, dot product, sum, multiplication by a scalar,... are encoded in template expressions and optimized.

Since STK++ version 0.9 and later, LAPACK library can be used as backends for dense matrix matrix decompositions (QR, Svd, eigenvalues) and least square regression. In order to use lapack, you must activate its usage by defining the following macros -DSTKUSELAPACK at compilation time and by linking your code with your installed lapack library using -llapack (at least in *nix operating systems).

Class	constructor	Note
lapack::Qr	Qr(data, ref = false) Qr(data)	if data is an ArrayXX and ref is true, data will be overwritten by Q
lapack::Svd	Svd(data, ref = false, withU=true, withV=true) Svd(data)	if ref is true, data will be overwritten by Q
lapack::SymEigen	SymEigen(data, ref = false) SymEigen(data)	if data is a SquareArray and ref is true, data will be overwritten by Q
lapack::MultiLeastSquare	MultiLeastSquare(b, a, isBref = false, isBref=false) MultiLeastSquare(b, a)	

To be continued...

References

[1] Serge Iovleff. *STK++ Arrays, User Guide*, 2016. R package version 1.0.2.