Statistics and Algebra Reference Guide

Serge Iovleff

October 1, 2018

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 (which provides statistical tools) and the "Algebra" project (which provide, mainly, matrix decomposition/inversion tools).

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.

The creation of a factors using as input a vector or a matrix is detailed in section (2)

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.pdf(2)= "      << l.pdf(2)      << "\n";
  stk_cout << "l.lpdf(2)= "     << l.lpdf(2)     << "\n";
  stk_cout << "l.cdf(3.96)= "   << l.cdf(3.96)   << "\n";
  stk_cout << "l.cdfc(3.96)= "  << l.cdfc(3.96)  << "\n";
  stk_cout << "l.lcdf(3.96)= "  << l.lcdf(3.96)  << "\n";
  stk_cout << "l.lcdfc(3.96)= "<< l.lcdfc(3.96)<< "\n";
  stk_cout << "l.icdf(0.975)= "<< l.icdf(0.975)<< "\n";
  stk_cout << "l.rand()= "      << l.rand()      << "\n";
  CArray<Real, 2, 3> a;  a = 0.5;
  stk_cout << "------------\n";
  stk_cout << "a=\n"         << a;
  stk_cout << "a.pdf(l)=\n"  << a.pdf(l);
  stk_cout << "a.lpdf(l)=\n" << a.lpdf(l);
  stk_cout << "a.cdf(l)=\n"  << a.cdf(l);
  stk_cout << "a.lcdf(l)=\n" << a.lcdf(l);
  stk_cout << "a.cdfc(l)=\n" << a.cdfc(l);
  stk_cout << "a.lcdfc(l)=\n"<< a.lcdfc(l);
  stk_cout << "a.icdf(l)=\n" << a.icdf(l);
}
```

Listing 1: Output

```
l.pdf(2)= 0.398942
l.lpdf(2)= -0.918939
l.cdf(3.96)= 0.975002
l.cdfc(3.96)= 0.0249979
l.lcdf(3.96)= -3.68896
l.lcdfc(3.96)= -3.68896
l.icdf(0.975)= 3.95996
l.rand()= 2.89849
-----------
a=
0.5 0.5 0.5
0.5 0.5 0.5
a.pdf(l)=
0.129518 0.129518 0.129518
0.129518 0.129518 0.129518
a.lpdf(l)=
-2.04394 -2.04394 -2.04394
-2.04394 -2.04394 -2.04394
a.cdf(l)=
0.0668072 0.0668072 0.0668072
0.0668072 0.0668072 0.0668072
a.lcdf(l)=
-0.0691435 -0.0691435 -0.0691435
-0.0691435 -0.0691435 -0.0691435
a.cdfc(l)=
0.933193 0.933193 0.933193
0.933193 0.933193 0.933193
a.lcdfc(l)=
-0.0691435 -0.0691435 -0.0691435
-0.0691435 -0.0691435 -0.0691435
a.icdf(l)=
2 2 2
2 2 2
```

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;                // generate a Cauchy random variate
    virtual Real pdf( Real const& x) const;   // probability distribution function (pdf)
    virtual Real lpdf( Real const& x) const; // log-pdf
    virtual Real cdf( Real const& t) const;   // cumulative distribution function (cdf)
    virtual Real cdfc( Real const& t) const; // complementary cdf
    virtual Real lcdf( Real const& t) const; // log-cdf
    virtual Real lcdfc( Real const& t) const;// log-cdfc
    virtual Real icdf( Real const& p) const; // inverse cdf (quantiles)
    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& 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 pdf value $f(x)$,
2. `lpdf(x)` return the log-pdf value $\log(f(x))$,
3. `rand()` return a random variate with pdf f,
4. `cdf(t)` return the cumulative distribution function (cdf) value $F(t) = \int_{-\infty}^{t} f(x)dx$
5. `lcdf(t)` return the log-cdf value $\log F(t)$
6. `cdfc(t)` return the complementary cdf value $G(t) = \int_{t}^{+\infty} f(x)dx$

7. `cdfc(t)` return the log-complementary cdf value $\log G(t)$

8. `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 $\lambda e^{-\lambda x}$
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`

All Distribution laws methods can be applied to a vector/array/expression using the corresponding methods. An example is given below.

Listing 3: Compute the log-complementary cdf in a logistic regression

```
// logis is logistic distribution
STK::Law::Logistic logis;
// y is a (R) matrix of size (n,p) and beta is a (R) vector of size p
STK::RMatrix<double> y;
STK::RVector<double> beta;
// compute log-complementary cdf
(y_*beta).lcdfc(logis);
```

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 (`/* */` for optional arg)

`min(m)`
`minByCol(m)`
`minByRow(m)`
	`min(m, w)`
	`minByCol(m, w)`
	`minByRow(m, w)`
		`minSafe(m/*,w*/)`
		`minSafeByCol(m/*,w*/)`
		`minSafeByRow(m/*,w*/)`

`max(m)`
`maxByCol(m)`
`maxByRow(m)`
	`max(m, w)`
	`maxByCol(m, w)`
	`maxByRow(m, w)`
		`maxSafe(m/*,w*/)`
		`maxSafeByCol(m/*,w*/)`
		`maxSafeByRow(m/*,w*/)`

`sum(m)`
`sumByCol(m)`
`sumByRow(m)`
	`sum(m, w)`
	`sumByCol(m, w)`
	`sumByRow(m, w)`
		`sumSafe(m/*,w*/)`
		`sumSafeByCol(m/*,w*/)`
		`sumSafeByRow(m/*,w*/)`

`mean(m)`
`meanByCol(m)`
`meanByRow(m)`
	`mean(m, w)`
	`meanByCol(m, w)`
	`meanByRow(m, w)`
		`meanSafe(m/*,w*/)`
		`meanSafeByCol(m/*,w*/)`
		`meanSafeByRow(m/*,w*/)`

`variance(m, unbiased)`
`varianceByCol(m, unbiased)`
`varianceByRow(m, unbiased)`
	`variance(m, w, unbiased)`
	`varianceByCol(m, w, unbiased)`
	`varianceByRow(m, w, unbiased)`
		`varianceSafe(m/*,w*/, unbiased)`
		`varianceSafeByCol(m/*,w*/, unbiased)`
		`varianceSafeByRow(m/*,w*/, unbiased)`

`varianceWithFixedMean(m, mu, unbiased)`
`varianceWithFixedMeanByCol(m, mu, unbiased)`
`varianceWithFixedMeanByRow(m, mu, unbiased)`
	`variance*(m, w, mu, unbiased)`
	`variance*ByCol(m, w, mu, unbiased)`
	`variance*ByRow(m, w, mu, unbiased)`
		`variance*Safe(m/*,w*/, mu, unbiased)`
		`variance*SafeByCol(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 4: 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 4: Output

```
True sigma=
   2  0.8 0.36
 0.8    2  0.8
0.36  0.8    1
Estimated sigma=
 2.48966  1.24486 0.437199
 1.24486  1.95422 0.731717
0.437199 0.731717   1.0612
```

1.3 Miscellaneous statistical functions

Given an array m, it is possible to center it, to standardize it and to perform the reverse operations. They are listed in table (5) given below. These methods are illustrated in the following example

Listing 5: Example

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

  return 0;
}
```

Listing 5: Output

```
mean(A)=
0.183092 1.57987 1.65149 0.816282
     1.97431 1.66366 1.97361
    0.761082
variance(A)=
1.37938 3.69946 3.08708 0.661574
    3.03347 9.26716 3.60422
    12.7621

mean(A)=
-1.11022e-17 -1.33227e-16 2.22045
    e-17 6.66134e-17 1.77636e-16
     1.11022e-17 -3.33067e-17
    4.44089e-17
variance(A)=
1 1 1 1 1 1 1 1

mean(A)=
0.183092 1.57987 1.65149 0.816282
     1.97431 1.66366 1.97361
    0.761082
variance(A)=
1.37938 3.69946 3.08708 0.661574
    3.03347 9.26716 3.60422
    12.7621
```

Function	weighted version
center(m, mean) centerByCol(m, mean) centerByRow(m, mean)	center(m, w, mean) centerByCol(m, w, mean) centerByRow(m, w, mean)
standardize(m, std, unbiased) standardizeByCol(m, std, unbiased) standardizeByRow(m, std, unbiased)	standardize(m, w, std, unbiased) standardizeByCol(m, w, std, unbiased) standardizeByRow(m, w, std, unbiased)
standardize(m, mean, std, unbiased) standardizeByCol(m, mean, std, unbiased) standardizeByRow(m, mean, std, unbiased)	standardize(m, w, mean, std, unbiased) standardizeByCol(m, w, 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 utilities 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.

> **Note:**
> By default, all functions applied on an array are applied column by column.

2 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 arrays). These classes are runners and you have to use the run method in order to trigger the computation of the factors.

An example is given below

Listing 6: Example

```
#include "STKpp.h"
using namespace STK;
int main(int argc, char** argv)
{
  CArray<char, 13, 3> fac;
  fac <<'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(fac.col(1));
  f1d.run();
  stk_cout << "nbLevels= "  << f1d.nbLevels() << "\n";
  stk_cout << "asInteger=\n"<< f1d.asInteger().transpose() << "\n";
  stk_cout << "Levels: "     << f1d.levels().transpose();
  stk_cout << "Levels counts: " << f1d.counts().transpose();

  Stat::MultiFactor<CArray<char, 13, 3> > f2d(fac);
  f2d.run();
  stk_cout << "nbLevels= "  << f2d.nbLevels() << "\n";
  stk_cout << "asInteger=\n"<< f2d.asInteger().transpose() << "\n";
  for (int i=f2d.levels().begin(); i<f2d.levels().end(); ++i)
    stk_cout <<"Levels "<< i <<": "<< f2d.levels()[i].transpose();
  stk_cout << "\n";
  for (int i=f2d.levels().begin(); i<f2d.levels().end(); ++i)
    stk_cout <<"Counts "<< i << ": "<< f2d.counts()[i].transpose();
  return 0;
}
```

Listing 6: 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 1 0 1 2 0 2 1 0 2 0 2 1
0 1 0 0 0 1 2 2 2 1 0 0 2
0 0 0 0 0 0 1 1 1 1 1 1 1

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

Counts 0: 5 4 4
Counts 1: 6 3 4
Counts 2: 6 7
```

3 Linear Algebra classes, methods and functions (Algebra project)

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 back-ends for dense matrix matrix decomposition (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 (or Qr)	Qr(data, ref = false) Qr(data)	if data is an ArrayXX and ref is true, data will be overwritten by Q
lapack::Svd (or Svd)	Svd(data, ref = false, withU=true, withV=true) Svd(data)	if ref is true, data will be overwritten by Q
lapack::SymEigen (or SymEigen)	SymEigen(data, ref = false) SymEigen(data)	if data is a SquareArray and ref is true, data will be overwritten by Q
lapack::MultiLeastSquare (or MultiLeastSquare)	MultiLeastSquare(b, a, isBref = false, isAref=false) MultiLeastSquare(b, a)	

3.1 Matrix decomposition

All methods for matrix decomposition are enclosed in classes. Computations are launched using the run method.

3.1.1 QR decomposition

QR decomposition (also called a QR factorization) of a matrix is a decomposition of a matrix A into a product $A = QR$ of an orthogonal matrix Q and an upper triangular matrix R.

QR decomposition can be achieved using either the class STK::Qr or if lapack is available STK::lapack::Qr. In later case, code have to be compiled using -DSTKUSElapack flag and linked using -llapack (at least for GNU-like compilers).

Listing 7: Example

```
#include <STKpp.h>
using namespace STK;
/** @ingroup tutorial */
int main(int argc, char** argv)
{
  Array2D<Real> a(5,4);
  a << 0, 1, 2, 3,
       2, 3, 4, 5,
       2, 1, 6, 7,
       0, 3,-1, 2,
       3,-1, 1, 1;
  stk_cout << _T("STK++ QR decomposition:\n");
  stk_cout << _T("-----------------------\n");
  Qr q(a); q.run();
  stk_cout << _T("R matrix:\n");
  stk_cout << q.R();
  ArrayXX QR = q.R();
  applyLeftHouseholderArray(QR, q.Q());
  stk_cout << _T("QR matrix:\n");
  stk_cout << QR;
  stk_cout << _T("\nlapack QR decomposition:\n");
  stk_cout << _T("-------------------------\n");
  lapack::Qr lq(a); lq.run();
  stk_cout << _T("R matrix:\n");
  stk_cout << lq.R();
  QR = lq.R();
  applyLeftHouseholderArray(QR, lq.Q());
  stk_cout << _T("QR matrix:\n");
  stk_cout << QR;
  return 0;
}
```

Listing 7: Output

```
STK++ QR decomposition:
-----------------------
R matrix:
4.12311 1.21268 5.57832  6.54846
      0 4.41921 2.08981  4.99158
      0       0   4.745  4.22321
      0       0       0 -1.53827
      0       0       0        0
QR matrix:
0   1   2 3
2   3   4 5
2   1   6 7
0   3  -1 2
3  -1   1 1

lapack QR decomposition:
-------------------------
R matrix:
-4.12311  -1.21268  -5.57832  -6.54846
       0  -4.41921  -2.08981  -4.99158
       0         0    -4.745  -4.22321
       0         0         0  -1.53827
       0         0         0         0
QR matrix:
0   1   2 3
2   3   4 5
2   1   6 7
0   3  -1 2
3  -1   1 1
```

Note:

By default the matrix Q is represented as a product of elementary reflectors $Q = H_1 H_2 ... H_k$, where $k = \min(m, n)$ each H_i has the form $H_i = I - \tau v v'$. It is possible to get the Q matrix (of size (m, m)) by using the `compQ()` method. Q will be overwritten.

It is possible to update a QR decomposition when a column is added or removed to the original matrix. In the following example, we remove the column number 2 of a matrix and then insert a column with value 1 after the column number 1.

Listing 8: Example

```
#include <STKpp.h>
using namespace STK;
/** @ingroup tutorial */
int main(int argc, char** argv)
{
  ArrayXX a(5,4), QR;
  a << 0, 1, 2, 3,
       2, 3, 4, 5,
       2, 1, 6, 7,
       0, 3,-1, 2,
       3,-1, 1, 1;
  stk_cout << "lapack QR decomposition:\n";
  lapack::Qr lq(a); lq.run();
  // remove column
  lq.eraseCol(2);
  stk_cout << "R matrix:\n";
  stk_cout << lq.R();
  QR = lq.Q() * lq.R();
  stk_cout << "QR matrix:\n";
  stk_cout << QR;
  // insert constant column with value 1 in column 2
  lq.insertCol(Const::Vector<Real>(5), 2);
  stk_cout << "\nR matrix:\n";
  stk_cout << lq.R();
  QR = lq.Q() * lq.R();
  stk_cout << "QR matrix:\n";
  stk_cout << QR;
  return 0;
}
```

Listing 8: Output

```
lapack QR decomposition:
R matrix:
-4.12311 -1.21268 -6.54846
       0 -4.41921 -4.99158
       0        0  4.49464
       0        0        0
       0        0        0
QR matrix:
0  1 3
2  3 5
2  1 7
0  3 2
3 -1 1

R matrix:
-4.12311 -1.21268 -1.69775 -6.54846
       0 -4.41921 -1.11811 -4.99158
       0        0 0.931381  1.39707
       0        0        0   -4.272
       0        0        0        0
QR matrix:
0  1 1 3
2  3 1 5
2  1 1 7
0  3 1 2
3 -1 1 1
```

3.1.2 SVD decomposition

The singular-value decomposition of an (m, n) real (or complex) matrix M is a factorization of the form $\mathbf{U\Sigma V}^*$, where U is an (m, n) real (or complex) unitary matrix, $\mathbf{\Sigma}$ is an (m, n) rectangular diagonal matrix with non-negative real numbers on the diagonal, and $\mathbf{V}$ is an (n, n) real (or complex) unitary matrix.

The diagonal entries σ_i of $\mathbf{\Sigma}$ are known as the singular values of M.

Listing 9: Example

```
#include <STKpp.h>
using namespace STK;
/** @ingroup tutorial */
int main(int argc, char** argv)
{
  ArrayXX a(5,4), usvt;
  a << 0, 1, 2, 3,
       2, 3, 4, 5,
       2, 1, 6, 7,
       0, 3,-1, 2,
       3,-1, 1, 1;
  stk_cout << _T("STK++ Svd decomposition:\n");
  stk_cout << _T("------------------------\n");
  Svd<ArrayXX> s(a); s.run();
  stk_cout << _T("Singular values:\n");
  stk_cout << s.D();
  stk_cout << _T("\nUDV^T matrix:\n");
  stk_cout << s.U()*s.D()*s.V().transpose();
  stk_cout << _T("\nlapack Svd decomposition:\n");
  stk_cout << _T("--------------------------\n");
  lapack::Svd ls(a); ls.run();
  stk_cout << _T("Singular values:\n");
  stk_cout << ls.D().transpose();
  stk_cout << _T("\nUDV^T matrix:\n");
  stk_cout << ls.U()*ls.D().diagonalize()*ls.V().transpose();
  return 0;
}
```

Listing 9: Output

```
STK++ Svd decomposition:
------------------------
Singular values:
12.6179       0       0       0
      0 4.17637       0       0
      0       0 2.51817       0
      0       0       0 1.00223

UDV^T matrix:
 2.5327e-16   1   2 3
          2   3   4 5
          2   1   6 7
5.13478e-16   3  -1 2
          3  -1   1 1

lapack Svd decomposition:
--------------------------
Singular values:
12.6179 4.17637 2.51817 1.00223

UDV^T matrix:
 1.07206e-15   1   2 3
           2   3   4 5
           2   1   6 7
-2.35922e-16   3  -1 2
           3  -1   1 1
```

Note:

Singular values are stored in a vector in `lapack` method and in a diagonal matrix in `STK++` method. It is also possible to compute only U and/or V matrix.

3.1.3 Eigenvalues decomposition

Let A be a square (n, n) matrix with n linearly independent eigenvectors, q_i $(i = 1, \dots, N)$. Then A can be factorized as

$$\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1}$$

where Q is the square (n, n) matrix whose i-th column is the eigenvector q_i of A and Λ is the diagonal matrix whose diagonal elements are the corresponding eigenvalues, i.e., $\Lambda_{ii} = \lambda_i$.

If A is a symmetric matrix then Q is an orthogonal matrix.

STK provide native and `lapack` interface classes allowing to compute the eigenvalue decomposition of a *symmetric* square matrix.

Listing 10: Example

```cpp
#include <STKpp.h>
using namespace STK;
/** @ingroup tutorial */
int main(int argc, char** argv)
{
  CSquareX a(5);
  a << 0, 1, 2, 3, 4,
       2, 3, 4, 5, 6,
       2, 1, 6, 7, 8,
       0, 3,-1, 2, 3,
       3,-1, 1, 1, 0;
  stk_cout << "STK++ Eigen decomposition:\n";
  stk_cout << "-------------------------\n";
  SymEigen<CSquareX> s(a.upperSymmetrize()); s.run();
  stk_cout << "D matrix:\n";
  stk_cout << s.eigenValues();
  CSquareX QDQt = s.eigenVectors() * s.eigenValues().diagonalize()
       * s.eigenVectors().transpose();
  stk_cout << "QDQ^T matrix:\n";
  stk_cout << QDQt;
  stk_cout << "\nlapack Eigen decomposition:\n";
  stk_cout << "-------------------------\n";
  lapack::SymEigen<CSquareX> ls(a); ls.run();
  stk_cout << "D matrix:\n";
  stk_cout << ls.eigenValues();
  QDQt = ls.eigenVectors() * ls.eigenValues().diagonalize() * ls.
      eigenVectors().transpose();
  stk_cout << "QDQ^T matrix:\n";
  stk_cout << QDQt;
  return 0;
}
```

Listing 10: Output

```
STK++ Eigen decomposition:
-------------------------
D matrix:
 20.8996
0.340126
-0.39673
-1.64329
-8.19967
QDQ^T matrix:
-2.33147e-15 1 2 3           4
             1 3 4 5           6
             2 4 6 7           8
             3 5 7 2           3
             4 6 8 3 1.64313e-14

lapack Eigen decomposition:
-------------------------
D matrix:
-8.19967
-1.64329
-0.39673
0.340126
 20.8996
QDQ^T matrix:
-6.88338e-15 1 2 3            4
             1 3 4 5            6
             2 4 6 7            8
             3 5 7 2            3
             4 6 8 3 -7.10543e-15
```

Note:

`STK++` eigenvalues computation need a full symmetric matrix as input while `lapack` version use only upper part of the input data. It is also possible to use the lower part of the matrix in the `lapack` version.

3.2 Solving least square problems

In linear regression, the observations are assumed to be the result of random deviations from an underlying relationship between the dependent variables y and independent variable x.

Given a data set

$$\{y_{i1}, \dots, y_{id}, x_{i1}, \dots, x_{ip}\}_{i=1}^{n}$$

of n statistical units, a linear regression model assumes that the relationship between the dependent variable y and the regressors x is linear. This relationship is modeled through a disturbance term that adds "noise" to the linear relationship between the dependent variable and regressors. Thus the model takes the form

$$\mathbf{y}_i = \mathbf{x}_i^{\mathsf{T}}\boldsymbol{\beta} + \varepsilon_i, \qquad i = 1, \dots, n.$$

Often these n equations are stacked together and written in matrix notation as

$$\mathbf{Y} = X\boldsymbol{\beta} + \boldsymbol{\varepsilon},$$

STK++ provide native and lapack interface classes allowing to solve the least square regression problem.

Listing 11: Example

```
#include <STKpp.h>
using namespace STK;
/** @ingroup tutorial */
int main(int argc, char** argv)
{
  CArrayXX y(1000,3), x(1000,5), beta(5,3);
  Law::Normal l(0,1);
  x.rand(l);
  beta << 0, 1, 2,
          2, 3, 4,
          2, 1, 6,
          0, 3,-1,
          3,-1, 1;
  y = x * beta + CArrayXX(1000, 3).rand(l);
  stk_cout << "STK++ MultiLeastSquare:\n";
  stk_cout << "-----------------------\n";
  STK::MultiLeastSquare<CArrayXX, CArrayXX> ols(y,x); ols.run();
  stk_cout << "beta matrix:\n";
  stk_cout << ols.x();
  stk_cout << "\nlapack MultiLeastSquare:\n";
  stk_cout << "-----------------------\n";
  STK::lapack::MultiLeastSquare<CArrayXX, CArrayXX> ls(y,x); ls.
      run();
  stk_cout << "beta matrix:\n";
  stk_cout << ls.x();
  return 0;
}
```

Listing 11: Output

```
STK++ MultiLeastSquare:
-----------------------
beta matrix:
  0.0247875    1.02139   1.99183
    1.96609    3.01792    4.0079
    2.04481    1.03293   6.03462
-0.00848336    2.99758  -1.05867
    3.02074  -0.996975   1.00954

lapack MultiLeastSquare:
------------------------
beta matrix:
  0.0247875    1.02139   1.99183
    1.96609    3.01792    4.0079
    2.04481    1.03293   6.03462
-0.00848336    2.99758  -1.05867
    3.02074  -0.996975   1.00954
```

Note:
It is also possible to solve weighted least-square regression problems.

3.3 Inverting matrices

Matrices can be inverted using either the templated functor STK::InvertMatrix or the templated function STK::invert. The first example below deals with general square and/or symmetric matrices.

Listing 12: Example

```
#include <STKpp.h>
using namespace STK;
/** @ingroup tutorial */
int main(int argc, char** argv)
{
  CArray<Real, 4, 4> a(4,4);
  a << 0, 1, 2, 3,
       2, 3, 4, 5,
       2, 1, 6, 7,
       0, 3,-1, 2;
  stk_cout << "Inverse general matrix:\n";
  stk_cout << "-----------------------\n";
  stk_cout << a*invert(a);
  stk_cout << "\nInverse upper-symmetric matrix:\n";
  stk_cout << "-------------------------------\n";
  stk_cout << a.upperSymmetrize();
  stk_cout << a.upperSymmetrize()*invert(a.
      upperSymmetrize());
  stk_cout << "\nInverse lower-symmetric matrix:\n";
  stk_cout << "-------------------------------\n";
  stk_cout << a.lowerSymmetrize();
  stk_cout << a.lowerSymmetrize()*invert(a.
      lowerSymmetrize());
  return 0;
}
```

Listing 12: Output

```
Inverse general matrix:
-----------------------
           1 0 0 0
           0 1 0 0
-8.88178e-16 0 1 0
           0 0 0 1

Inverse upper-symmetric matrix:
-------------------------------
0 1 2 3
1 3 4 5
2 4 6 7
3 5 7 2
           1 0            0            0
           0 1 -2.22045e-16 -2.22045e-16
 6.66134e-16 0            1 -2.22045e-16
-2.10942e-15 0  1.11022e-16            1

Inverse lower-symmetric matrix:
-------------------------------
0 2  2  0
2 3  1  3
2 1  6 -1
0 3 -1  2
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

The second example deals with lower and upper triangular arrays.

Listing 13: Example

```
#include <STKpp.h>
using namespace STK;
/** @ingroup tutorial */
int main(int argc, char** argv)
{
  Array2DLowerTriangular<Real> a(5,5);
  a << 1,
       1, 2,
       3, 4, 3,
       4, 5, 6, 6,
       7, 8, 2, 3, 2;
  stk_cout << "Inverse lower-triangular matrix:\n";
  stk_cout << "--------------------------------\n";
  stk_cout << a*invert(a);
  stk_cout << "\nInverse upper-triangular matrix:\n";
  stk_cout << "--------------------------------\n";
  stk_cout << a.transpose()*invert(a.transpose());
  return 0;
}
```

Listing 13: Output

```
Inverse lower-triangular matrix:
--------------------------------
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1

Inverse upper-triangular matrix:
--------------------------------
1 0 0 -1.11022e-16 0
0 1 0            0 0
0 0 1            0 0
0 0 0            1 0
0 0 0            0 1
```

Note:
If the matrix is not inversible, the result provided will be a generalized inverse.

References

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