

Contents

1	Math Libraries in ROOT	3
1.1	TMath	3
1.2	Random Numbers	4
1.2.1	TRandom	4
1.2.2	TRandom1	4
1.2.3	TRandom2	4
1.2.4	TRandom3	4
1.2.5	Seeding the Generators	5
1.2.6	Examples of Using the Generators	5
1.2.7	Random Number Distributions	5
1.2.8	UNURAN	6
1.2.9	Performances of Random Numbers	7
1.3	MathCore Library	7
1.4	Generic Vectors for 2, 3 and 4 Dimensions (GenVector)	8
1.4.1	Main Characteristics	8
1.4.2	Example: 3D Vector Classes	11
1.4.3	Example: 3D Point Classes	13
1.4.4	Example: LorentzVector Classes	14
1.4.5	Example: Vector Transformations	16
1.4.6	Example with External Packages	19
1.5	MathMore Library	19
1.6	Mathematical Functions	20
1.6.1	Special Functions in MathCore	20
1.6.2	Special Functions in MathMore	21
1.6.3	Probability Density Functions (PDF)	22
1.6.4	Cumulative Distribution Functions (CDF)	23
1.7	Linear Algebra: SMatrix Package	24
1.7.1	Example: Vector Class (SVector)	24
1.7.2	Example: Matrix Class (SMatrix)	25
1.7.3	Example: Matrix and Vector Functions and Operators	28
1.7.4	Matrix and Vector Functions	29
1.8	Minuit2 Package	29
1.9	ROOT Statistics Classes	30
1.9.1	Classes for Computing Limits and Confidence Levels	30
1.9.2	Specialized Classes for Fitting	30
1.9.3	Multi-variate Analysis Classes	31

Chapter 1

Math Libraries in ROOT

The aim of Math libraries in ROOT is to provide and to support a coherent set of mathematical and statistical functions. The latest developments have been concentrated in providing first versions of the **MathCore** and **MathMore** libraries, included in ROOT v5.08. Other recent developments include the new version of MINUIT, which has been re-designed and re-implemented in the C++ language. It is integrated in ROOT. In addition, an optimized package for describing small matrices and vector with fixed sizes and their operation has been developed (**SMatrix**). The structure is shown in the following picture.

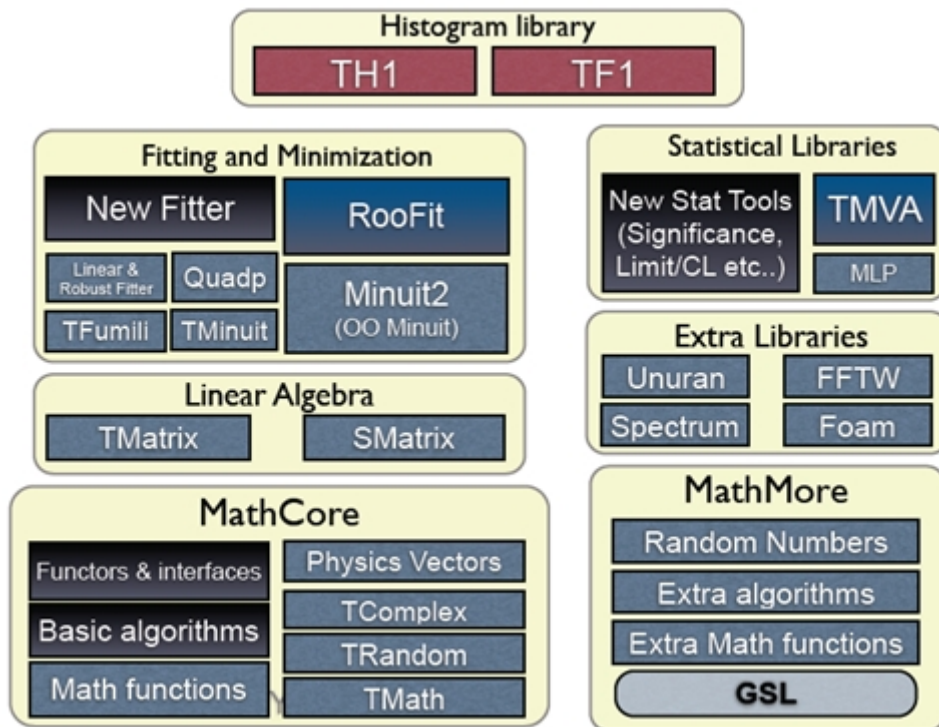


Figure 1.1: Math libraries and packages

1.1 TMath

In the namespace, **TMath** a collection of free functions is provided for the following functionality:

- numerical constants (like **pi**, **e**, **h**, etc.);
- elementary and trigonometric functions;
- functions to find **min** and **max** of arrays;
- statistic functions to find mean and **rms** of arrays of data;
- algorithms for binary search/ hashing sorting;

- special mathematical functions like `Bessel`, `Erf`, `Gamma`, etc.;
- statistical functions, like common probability and cumulative (quantile) distributions

For more details, see the reference documentation of **TMath** at <http://root.cern.ch/root/html/doc/TMath.html>.

1.2 Random Numbers

In ROOT pseudo-random numbers can be generated using the **TRandom** classes. 4 different types exist: **TRandom**, **TRandom1**, **TRandom2** and **TRandom3**. All they implement a different type of random generators. **TRandom** is the base class used by others. It implements methods for generating random numbers according to pre-defined distributions, such as Gaussian or Poisson.

1.2.1 TRandom

Pseudo-random numbers are generated using a linear congruential random generator. The multipliers used are the same of the BSD `rand()` random generator. Its sequence is:

$$x_{n+1} = (ax_n + c) \bmod m \text{ with } a = 1103515245, c = 12345 \text{ and } m = 2^{31}.$$

This type of generator uses a state of only a 32 bit integer and it has a very short period, 2^{31} , about 10^9 , which can be exhausted in just few seconds. The quality of this generator is therefore BAD and it is strongly recommended to NOT use for any statistical study.

1.2.2 TRandom1

This random number generator is based on the Ranlux engine, developed by M. Lüscher and implemented in Fortran by F. James. This engine has mathematically proven random proprieties and a long period of about 10^{171} . Various luxury levels are provided (1,2,3,4) and can be specified by the user in the constructor. Higher the level, better random properties are obtained at a price of longer CPU time for generating a random number. The level 3 is the default, where any theoretical possible correlation has very small chance of being detected. This generator uses a state of 24 32-bits words. Its main disadvantage is that is much slower than the others (see timing table). For more information on the generator see the following article:

- F. James, “RANLUX: A Fortran implementation of the high quality pseudo-random number generator of Lüscher”, Computer Physics Communication, 79 (1994) 111.

1.2.3 TRandom2

This generator is based on the maximally equidistributed combined Tausworthe generator by L’Ecuyer. It uses only 3 32-bits words for the state and it has a period of about 10^{26} . It is fast and given its small states, it is recommended for applications, which require a very small random number size. For more information on the generator see the following article:

- P. L’Ecuyer, “Maximally Equidistributed Combined Tausworthe Generators”, Mathematics of Computation, 65, 213 (1996), 203-213.

1.2.4 TRandom3

This is based on the Mersenne and Twister pseudo-random number generator, developed in 1997 by Makoto Matsumoto and Takuji Nishimura. The newer implementation is used, referred in the literature as MT19937. It is a very fast and very high quality generator with a very long period of 10^{6000} . The disadvantage of this generator is that it uses a state of 624 words. For more information on the generator see the following article:

- M. M. Matsumoto and T. Nishimura, “Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator”, ACM Trans. on Modeling and Computer Simulations, 8, 1, (1998), 3-20.

TRandom3 is the recommended random number generator, and it is used by default in ROOT using the global **gRandom** object (see chapter `gRandom`).

1.2.5 Seeding the Generators

The seeds for the generators can be set in the constructor or by using the `SetSeed` method. When no value is given the generator default seed is used, like 4357 for `TRandom3`. In this case identical sequence will be generated every time the application is run. When the 0 value is used as seed, then a unique seed is generated using a `TUUID`, for `TRandom1`, `TRandom2` and `TRandom3`. For `TRandom` the seed is generated using only the machine clock, which has a resolution of about 1 sec. Therefore identical sequences will be generated if the elapsed time is less than a second.

1.2.6 Examples of Using the Generators

The method `Rndm()` is used for generating a pseudo-random number distributed between 0 and 1 as shown in the following example:

```
// use default seed
// (same random numbers will be generated each time)
TRandom3 r; // generate a number in interval ]0,1] (0 is excluded)
r.Rndm();
double x[100];
r.RndmArray(100,x); // generate an array of random numbers in ]0,1]
TRandom3 rdm(111); // construct with a user-defined seed
// use 0: a unique seed will be automatically generated using TUUID
TRandom1 r1(0);
TRandom2 r2(0);
TRandom3 r3(0);
// seed generated using machine clock (different every second)
TRandom r0(0);
```

1.2.7 Random Number Distributions

The `TRandom` base class provides functions, which can be used by all the other derived classes for generating random variates according to predefined distributions. In the simplest cases, like in the case of the exponential distribution, the non-uniform random number is obtained by applying appropriate transformations. In the more complicated cases, random variates are obtained using acceptance-rejection methods, which require several random numbers.

```
TRandom3 r;
// generate a gaussian distributed number with:
// mu=0, sigma=1 (default values)
double x1 = r.Gaus();
double x2 = r.Gaus(10,3); // use mu = 10, sigma = 3;
```

The following table shows the various distributions that can be generated using methods of the `TRandom` classes. More information is available in the reference documentation for `TRandom`. In addition, random numbers distributed according to a user defined function, in a limited interval, or to a user defined histogram, can be generated in a very efficient way using `TF1::GetRandom()` or `TH1::GetRandom()`.

Distributions	Description
<code>Double_t Uniform(Double_t x1, Double_t x2)</code>	Uniform random numbers between <code>x1,x2</code>
<code>Double_t Gaus(Double_t mu, Double_t sigma)</code>	Gaussian random numbers. Default values: <code>mu=0, sigma=1</code>
<code>Double_t Exp(Double_t tau)</code>	Exponential random numbers with mean <code>tau</code> .
<code>Double_t Landau(Double_t mean, Double_t sigma)</code>	Landau distributed random numbers. Default values: <code>mean=0, sigma=1</code>
<code>Double_t BreitWigner(Double_t mean, Double_t gamma)</code>	Breit-Wigner distributed random numbers. Default values <code>mean=0, gamma=1</code>
<code>Int_t Poisson(Double_t mean)</code>	Poisson random numbers
<code>Double_t PoissonD(Double_t mean)</code>	
<code>Int_t Binomial(Int_t ntot, Double_t prob)</code>	Binomial Random numbers

<code>Circle(Double_t &x,Double_t &y,Double_t r)</code>	Generate a random 2D point (x,y) in a circle of radius r
<code>Sphere(Double_t &x,Double_t &y,Double_t &z,Double_t r)</code>	Generate a random 3D point (x,y,z) in a sphere of radius r
<code>Rannor(Double_t &a,Double_t &b)</code>	Generate a pair of Gaussian random numbers with mu=0 and sigma=1

1.2.8 UNURAN

An interface to a new package, UNU.RAN, (Universal Non Uniform Random number generator for generating non-uniform pseudo-random numbers) was introduced in ROOT v5.16.

UNU.RAN is an ANSI C library licensed under GPL. It contains universal (also called automatic or black-box) algorithms that can generate random numbers from large classes of continuous (in one or multi-dimensions), discrete distributions, empirical distributions (like histograms) and also from practically all standard distributions. An extensive online documentation is available at the UNU.RAN Web Site <http://statmath.wu-wien.ac.at/unuran/>

The ROOT class **TUnuran** is used to interface the UNURAN package. It can be used as following:

- With the UNU.RAN native, string API for pre-defined distributions (see UNU.RAN documentation for the allowed string values at <http://statistik.wu-wien.ac.at/unuran/doc/unuran.html>):

```
TUnuran unr;
// initialize unuran to generate normal random numbers using
// a "arou" method
unr.Init("normal()", "method=arou");
...
// sample distributions N times (generate N random numbers)
for (int i = 0; i < N; ++i)
    double x = unr.Sample();
```

- For continuous 1D distribution object via the class **TUnuranContDist** that can be created for example from a **TF1** function providing the pdf (probability density function) . The user can optionally provide additional information via **TUnuranContDist::SetDomain(min,max)** like the **domain()** for generating numbers in a restricted region.

```
// 1D case: create a distribution from two TF1 object
// pointers pdfFunc
TUnuranContDist dist( pdfFunc);
// initialize unuran passing the distribution and a string
// defining the method
unr.Init(dist, "method=hinv");
// sample distribution N times (generate N random numbers)
for (int i = 0; i < N; ++i)
    double x = unr.Sample();
```

- For multi-dimensional distribution via the class **TUnuranMultiContDist**, which can be created from a the multi-dimensional pdf.

```
// Multi- dimensional case from a TF1 (TF2 or TF3) objects
TUnuranMultiContDist dist( pdfFuncMulti);
// the recommended method for multi-dimensional function is "hitro"
unr.Init(dist, "method=hitro");
// sample distribution N times (generate N random numbers)
double x[NDIM];
for (int i = 0; i < N; ++i)
    unr.SampleMulti(x);
```

- For discrete distribution via the class **TUnuranDiscrDist**, which can be initialized from a **TF1** or from a vector of probabilities.

```
// Create distribution from a vector of probabilities
double pv[NSize] = {0.1,0.2,...};
TUnuranDiscrDist dist(pv,pv+NSize);
// the recommended method for discrete distribution is
unr.Init(dist, "method=dgt");
// sample N times (generate N random numbers)
for (int i = 0; i < N; ++i)
    int k = unr.SampleDiscr();
```

- For empirical distribution via the class **TUnuranEmpDist**. In this case one can generate random numbers from a set of un-bin or bin data. In the first case the parent distribution is estimated by UNU.RAN using a gaussian kernel smoothing algorithm. The **TUnuranEmpDist** distribution class can be created from a vector of data or from **TH1** (using the bins or from its buffer for un-binned data).

```
// Create distribution from a set of data
// vdata is an std::vector containing the data
TUnuranEmpDist dist(vdata.begin(),vdata.end());
unr.Init(dist);
// sample N times (generate N random numbers)
for (int i = 0; i<N; ++i)
    double x = unr.Sample();
```

- For some predefined distributions, like **Poisson** and **Binomial**, one can use directly a function in the **TUnuran** class. This is more convenient in passing distribution parameters than using directly the string interface.

```
TUnuran unr;
// Initialize unuran to generate normal random numbers from the
// Poisson distribution with parameter mu
unr.InitPoisson(mu);
...
// Sample distributions N times (generate N random numbers)
for (int i = 0; i<N; ++i)
    int k = unr.SampleDiscr();
```

Functionality is also provided via the C++ classes for using a different random number generator by passing a **TRandom** pointer when constructing the **TUnuran** class (by default the ROOT **gRandom** is passed to UNURAN).

1.2.9 Performances of Random Numbers

Here are the CPU times obtained using the four random classes on an **lxplus** machine with an Intel 64 bit architecture and compiled using gcc 3.4:

	TRandom (ns/call)	TRandom1 (ns/call)	TRandom2 (ns/call)	TRandom3 (ns/call)
Rndm()	-	-	6	9
Gaus()	31	161	35	42
Rannor()	116	216	126	130
Poisson(m=10)	147	1161	162	239
Poisson(m=10) UNURAN	80	294	89	99

1.3 MathCore Library

MathCore provides a collection of functions and C++ classes for numerical computing. This library includes only the basic mathematical functions and algorithms and not all the functionality required by the physics community. A more advanced mathematical functionality is provided by the **MathMore** library. The current set included classes are:

- Basic special functions like the gamma, beta and error function.

- Mathematical functions used in statistics, such as the probability density functions and the cumulative distributions functions (lower and upper integral of the pdf's).
- **GenVector**: physics and geometry vectors for 3 and 4 dimensions with their transformations (rotations and boost).
- Generic (**ROOT::Math::IFunction**) and parametric (**ROOT::Math::IParamFunction**) function interfaces for one and multi dimensions.

A detailed description for all **MathCore** classes is available in the [online reference documentation](#). The **MathCore** library presented in the ROOT distribution contains the Cling dictionary for I/O and interactive usage. For the template classes, the dictionary is provided for some of the possible types, such as those based on double and Double32_t. For the I/O or interactive use of other types, the dictionary must be first generated. An example on how to generate the required dictionary is provided in the tutorial `mathcoreVectorFloatIO.C` (in `$ROOTSYS/tutorials/math`). **MathCore** can also be built as an independent package using `configure/make`. In this case the library will not contain the dictionary information and cannot be used interactively in ROOT.

1.4 Generic Vectors for 2, 3 and 4 Dimensions (GenVector)

GenVector is a package intended to represent vectors and their operations and transformations, such as rotations and Lorentz transformations, in 3 and 4 dimensions. The 3D space is used to describe the geometry vectors and points, while the 4D space-time is used for physics vectors representing relativistic particles. These 3D and 4D vectors are different from vectors of the linear algebra package, which describe generic N-dimensional vectors. Similar functionality is currently provided by the CLHEP and packages and the ROOT Physics vector classes (See “Physics Vectors”). It also re-uses concepts and ideas from the CMS [Common Vector package](#). In contrast to CLHEP or the ROOT physics libraries, **GenVector** provides class templates for modeling the vectors. The user can control how the vector is internally represented. This is expressed by a choice of coordinate system, which is supplied as a template parameter when the vector is constructed. Furthermore, each coordinate system is itself a template, so that the user can specify the underlying scalar type.

The **GenVector** classes do not inherit from **TObject**, therefore cannot be used as in the case of the physics vector classes in ROOT collections.

In addition, to optimize performances, no virtual destructors are provided. In the following paragraphs, the main characteristics of **GenVector** are described. A more detailed description of all the **GenVector** classes is available also at <http://seal.cern.ch/documents/mathlib/GenVector.pdf>

1.4.1 Main Characteristics

1.4.1.1 Optimal Runtime Performances

We try to minimize any overhead in the run-time performance. We have deliberately avoided the use of any virtual function and even virtual destructors in the classes. In addition, as much as possible functions are defined as inline. For this reason, we have chosen to use template classes to implement the **GenVector** concepts instead of abstract or base classes and virtual functions. It is then recommended to avoid using the **GenVector** classes polymorphically and developing classes inheriting from them.

1.4.1.2 Points and Vector Concept

Mathematically vectors and points are two distinct concepts. They have different transformations, as vectors only rotate while points rotate and translate. You can add two vectors but not two points and the difference between two points is a vector. We then distinguish for the 3 dimensional case, between points and vectors, modeling them with different classes:

- **ROOT::Math::DisplacementVector2D** and **ROOT::Math::DisplacementVector3D** template classes describing 2 and 3 component direction and magnitude vectors, not rooted at any particular point;
- **ROOT::Math::PositionVector2D** and **ROOT::Math::PositionVector3D** template classes modeling the points in 2 and 3 dimensions.

For the 4D space-time vectors, we use the same class to model them, **ROOT::Math::LorentzVector**, since we have recognized a limited need for modeling the functionality of a 4D point.

1.4.1.3 Generic Coordinate System

The vector classes are based on a generic type of coordinate system, expressed as a template parameter of the class. Various classes exist to describe the various coordinates systems:

2D coordinate system classes:

- `ROOT::Math::Cartesian2D`, based on `(x,y)`;
- `ROOT::Math::Polar2D`, based on `(r,phi)`;

3D coordinate system classes:

- `ROOT::Math::Cartesian3D`, based on `(x,y,z)`;
- `ROOT::Math::Polar3D`, based on `(r,theta,phi)`;
- `ROOT::Math::Cylindrical3D`, based on `(rho,z,phi)`
- `ROOT::Math::CylindricalEta3D`, based on `(rho,eta,phi)`, where `eta` is the pseudo-rapidity;

4D coordinate system classes:

- `ROOT::Math::PxPyPzE4D`, based on based on `(px,py,pz,E)`;
- `ROOT::Math::PxPyPzM4D`, based on based on `(px,py,pz,M)`;
- `ROOT::Math::PtEtaPhiE4D`, based on based on `(pt,eta,phi,E)`;
- `ROOT::Math::PtEtaPhiM4D`, based on based on `(pt,eta,phi,M)`;

Users can define the vectors according to the coordinate type, which is the most efficient for their use. Transformations between the various coordinate systems are available through copy constructors or the assignment (=) operator. For maximum flexibility and minimize memory allocation, the coordinate system classes are templated on the scalar type. To avoid exposing templated parameter to the users, typedefs are defined for all types of vectors based on doubles. See in the examples for all the possible types of vector classes, which can be constructed by users with the available coordinate system types.

1.4.1.4 Coordinate System Tag

The 2D and 3D points and vector classes can be associated to a tag defining the coordinate system. This can be used to distinguish between vectors of different coordinate systems like global or local vectors. The coordinate system tag is a template parameter of the `ROOT::Math::DisplacementVector3D` and `ROOT::Math::PositionVector3D` (and also for 2D classes). A default tag exists for users who do not need this functionality, `ROOT::Math::DefaultCoordinateSystemTag`.

1.4.1.5 Transformations

The transformations are modeled using simple (non-template) classes, using double as the scalar type to avoid too large numerical errors. The transformations are grouped in rotations (in 3 dimensions), Lorentz transformations and Poincare transformations, which are translation/rotation combinations. Each group has several members which may model physically equivalent transformations but with different internal representations. Transformation classes can operate on all type of vectors by using the operator () or the operator * and the transformations can be combined via the operator *. The available transformations are:

- 3D rotation classes
- rotation described by a 3x3 matrix (`ROOT::Math::Rotation3D`)
- rotation described by Euler angles (`ROOT::Math::EulerAngles`)
- rotation described by a direction axis and an angle (`ROOT::Math::AxisAngle`)
- rotation described by a quaternion (`ROOT::Math::Quaternion`)

- optimized rotation around x (`ROOT::Math::RotationX`), y (`ROOT::Math::RotationY`) and z (`ROOT::Math::RotationZ`) and described by just one angle.
- 3D transformation: we describe the transformations defined as a composition between a rotation and a translation using the class `ROOT::Math::Transform3D`. It is important to note that transformations act differently on vectors and points. The vectors only rotate, therefore when applying a transformation (rotation + translation) on a vector, only the rotation operates while the translation has no effect. The `Transform3D` class interface is similar to the one used in the CLHEP Geometry package (class).
- Lorentz rotation:
 - generic Lorentz rotation described by a 4x4 matrix containing a 3D rotation part and a boost part (class `ROOT::Math::LorentzRotation`)
 - a pure boost in an arbitrary direction and described by a 4x4 symmetric matrix or 10 numbers (class `ROOT::Math::Boost`)
 - boost along the axis: x (`ROOT::Math::BoostX`), y (`ROOT::Math::BoostY`) and z (`ROOT::Math::BoostZ`).

1.4.1.6 Minimal Vector Classes Interface

We have tried to keep the interface to a minimal level by:

- Avoiding methods that provide the same functionality but use different names (like `getX()` and `x()`).
- Minimizing the number of setter methods, avoiding methods, which can be ambiguous and can set the vector classes in an inconsistent state. We provide only methods which set all the coordinates at the same time or set only the coordinates on which the vector is based, for example `SetX()` for a Cartesian vector. We then enforce the use of transformations as rotations or translations (additions) for modifying the vector contents.
- The majority of the functionality, which is present in the CLHEP package, involving operations on two vectors, is moved in separated helper functions (see `ROOT::Math::VectorUtil`). This has the advantage that the basic interface will remain more stable with time while additional functions can be added easily.

1.4.1.7 Naming Convention

As part of ROOT, the `GenVector` package adheres to the prescribed ROOT naming convention, with some (approved) exceptions, as described here:

- Every class and function is in the `ROOT::Math` namespace.
- Member function names start with upper-case letter, apart some exceptions (see the next section about CLHEP compatibility).

1.4.1.8 Compatibility with CLHEP Vector Classes

- For backward compatibility with CLHEP the vector classes can be constructed from a CLHEP `HepVector` or `HepLorentzVector`, by using a template constructor, which requires only that the classes implement the accessors `x()`, `y()`, and `z()` (and `t()` for the 4D).
- We provide vector member function with the same naming convention as CLHEP for the most used functions like `x()`, `y()` and `z()`.

1.4.1.9 Connection to Linear Algebra Package

In some use cases, like in track reconstruction, it is needed to use the content of the vector and rotation classes in conjunction with linear algebra operations. We prefer to avoid any direct dependency to any linear algebra package. However, we provide some hooks to convert to and from linear algebra classes. The vector and the transformation classes have methods which allow to get and set their data members (like `SetCoordinates` and `GetCoordinates`) passing either a generic iterator or a pointer to a contiguous set of data, like a C array. This allows an easy connection with the linear algebra package, which in turn, allows creation of matrices using C arrays (like the ROOT `TMatrix` classes) or iterators (`SMatrix` classes). Multiplication between linear algebra matrices and `GenVector` vectors is possible by using the template free functions `ROOT::Math::VectorUtil::Mult`. This function works for any linear algebra matrix, which implements the operator (i,j) and with first matrix element at $i=j=0$.

1.4.2 Example: 3D Vector Classes

To avoid exposing template parameter to the users, typedef's are defined for all types of vectors based on double's and float's. To use them, one must include the header file `Math/Vector3D.h`. The following typedef's, defined in the header file `Math/Vector3Dfwd.h`, are available for the different instantiations of the template class `ROOT::Math::DisplacementVector3D`:

- `ROOT::Math::XYZVector` vector based on `x,y,z` coordinates (Cartesian) in double precision
- `ROOT::Math::XYZVectorF` vector based on `x,y,z` coordinates (Cartesian) in float precision
- `ROOT::Math::Polar3DVector` vector based on `r,theta,phi` coordinates (polar) in double precision
- `ROOT::Math::Polar3DVectorF` vector based on `r,theta,phi` coordinates (polar) in float precision
- `ROOT::Math::RhoZPhiVector` vector based on `rho,z,phi` coordinates (cylindrical) in double precision
- `ROOT::Math::RhoZPhiVectorF` vector based on `rho,z,phi` coordinates (cylindrical) in float precision
- `ROOT::Math::RhoEtaPhiVector` vector based on `rho,eta,phi` coordinates (cylindrical using `eta` instead of `z`) in double precision
- `ROOT::Math::RhoEtaPhiVectorF` vector based on `rho,eta,phi` coordinates (cylindrical using `eta` instead of `z`) in float precision

1.4.2.1 Constructors and Assignment

The following declarations are available:

```
XYZVector      v1;           //an empty vector (x=0, y=0, z=0)
XYZVector      v2(1,2,3);    //vector with x=1, y=2, z=3;
Polar3DVector   v3(1,PI/2,PI); //vector with r=1, theta=PI/2, phi=PI
RhoEtaPhiVector v4(1,2, PI);  //vector with rho=1, eta=2, phi=PI
```

Note that each vector type is constructed by passing its coordinate representation, so a `XYZVector(1,2,3)` is different from a `Polar3DVector(1,2,3)`. In addition, the vector classes can be constructed by any vector, which implements the accessors `x()`, `y()` and `z()`. This can be another 3D vector based on a different coordinate system type. It can be even any vector of a different package, like the CLHEP `HepThreeVector` that implements the required signature.

```
XYZVector      v1(1,2,3);
RhoEtaPhiVector r2(v1);
CLHEP::HepThreeVector q(1,2,3);
XYZVector      v3(q);
```

1.4.2.2 Coordinate Accessors

All coordinate accessors are available through the class `ROOT::Math::DisplacementVector3D`:

```
//returns cartesian components for the cartesian vector v1
v1.X(); v1.Y(); v1.Z();
//returns cylindrical components for the cartesian vector v1
v1.Rho(); v1.Eta(); v1.Phi();
//returns cartesian components for the cylindrical vector r2
r2.X(); r2.Y(); r2.Z()
```

In addition, all the 3 coordinates of the vector can be retrieved with the `GetCoordinates` method:

```
double d[3];
//fill d array with (x,y,z) components of v1
v1.GetCoordinates(d);
//fill d array with (r,eta,phi) components of r2
r2.GetCoordinates(d);
std::vector vc(3);
//fill std::vector with (x,y,z) components of v1
v1.GetCoordinates(vc.begin(),vc.end());
```

See the reference documentation of `ROOT::Math::DisplacementVector3D` for more details on all the coordinate accessors.

1.4.2.3 Setter Methods

One can set only all the three coordinates via:

```
v1.SetCoordinates(c1,c2,c3); // (x,y,z) for a XYZVector
r2.SetCoordinates(c1,c2,c3); // r,theta,phi for a Polar3DVector
r2.SetXYZ(x,y,z); // 3 cartesian components for Polar3DVector
```

Single coordinate setter methods are available for the basic vector coordinates, like `SetX()` for a `XYZVector` or `SetR()` for a polar vector. Attempting to do a `SetX()` on a polar vector will not compile.

```
XYZVector v1;
v1.SetX(1); //OK setting x for a Cartesian vector
Polar3DVector v2;
v2.SetX(1); //ERROR: cannot set X for a Polar vector.
//Method will not compile
v2.SetR(1); //OK setting r for a Polar vector
```

In addition, there are setter methods from C arrays or iterator

```
double d[3] = {1.,2.,3.};
XYZVector v;
// set (x,y,z) components of v using values from d
v.SetCoordinates(d);
```

or, for example, from an `std::vector` using the iterator

```
std::vector w(3);
// set (x,y,z) components of v using values from w
v.SetCoordinates(w.begin(),w.end());
```

1.4.2.4 Arithmetic Operations

The following operations are possible between vector classes, even of different coordinate system types: (`v1,v2` are any type of `ROOT::Math::DisplacementVector3D` classes, `v3` is the same type of `v1`; `a` is a scalar value)

```
v1 += v2;
v1 -= v2;
v1 = - v2;
v1 *= a;
v1 /= a;
v2 = a * v1;
v2 = v1 / a;
v2 = v1 * a;
v3 = v1 + v2;
v3 = v1 - v2;
```

1.4.2.5 Comparison

For `v1` and `v2` of the same type (same coordinate system and same scalar type):

```
v1 == v2;
v1 != v2;
```

1.4.2.6 Dot and Cross Product

We support the dot and cross products, through the `Dot()` and `Cross()` method, with any vector (`q`) implementing `x()`, `y()` and `z()`.

```
XYZVector v1(x,y,z);
double s = v1.Dot(q);
XYZVector v2 = v1.Cross(q);
```

Note that the multiplication between two vectors using the operator `*` is not supported because it is ambiguous.

1.4.2.7 Other Methods

```
XYZVector u = v1.Unit(); //return unit vector parallel to v1
```

1.4.3 Example: 3D Point Classes

To use all possible types of 3D points one must include the header file `Math/Point3D.h`. The following type-def's defined in the header file `Math/Point3Dfwd.h`, are available for different instantiations of the template class `ROOT::Math::PositionVector3D`:

- `ROOT::Math::XYZPoint` point based on `x`, `y`, `z` coordinates (Cartesian) in double precision
- `ROOT::Math::XYZPointF` point based on `x`, `y`, `z` coordinates (Cartesian) in float precision
- `ROOT::Math::Polar3DPoint` point based on `r`, `theta`, `phi` coordinates (polar) in double precision
- `ROOT::Math::Polar3DPointF` point based on `r`, `theta`, `phi` coordinates (polar) in float precision
- `ROOT::Math::RhoZPhiPoint` point based on `rho`, `z`, `phi` coordinates (cylindrical using `z`) in double precision
- `ROOT::Math::RhoZPhiPointF` point based on `rho`, `z`, `phi` coordinates (cylindrical using `z`) in float precision
- `ROOT::Math::RhoEtaPhiPoint` point based on `rho`, `eta`, `phi` coordinates (cylindrical using `eta` instead of `z`) in double precision
- `ROOT::Math::RhoEtaPhiPointF` point based on `rho`, `eta`, `phi` coordinates (cylindrical using `eta` instead of `z`) in float precision

1.4.3.1 Constructors and Assignment

The following declarations are available:

```
XYZPoint      p1;           //an empty vector (x=0, y=0, z=0)
XYZPoint      p2(1,2,3);    //
```

Note that each point type is constructed by passing its coordinate representation, so a `XYZPoint(1,2,3)` is different from a `Polar3DPoint(1,2,3)`. In addition the point classes can be constructed by any vector, which implements the accessors `x()`, `y()` and `z()`. This can be another 3D point based on a different coordinate system type or even any vector of a different package, like the CLHEP `HepThreePoint` that implements the required signatures.

```
XYZPoint      p1(1,2,3);
RhoEtaPhiPoint r2(v1);
CLHEP::HepThreePoint q(1,2,3);
XYZPoint      p3(q);
```

1.4.3.2 Coordinate Accessors and Setter Methods

For the points classes we have the same getter and setter methods as for the vector classes. See “Example: 3D Vector Classes”.

1.4.3.3 Point-Vector Operations

The following operations are possible between points and vector classes: (`p1`, `p2` and `p3` are instantiations of the `ROOT::Math::PositionVector3D` objects with `p1` and `p3` of the same type; `v1` and `v2` are `ROOT::Math::DisplacementVector3D` objects).

```
p1 += v1;
p1 -= v1;
p3 = p1 + v1;    // p1 and p3 are the same type
p3 = v1 + p1;    // p3 is based on the same coordinate system as v1
p3 = p1 - v1;
p3 = v1 - p1;
v2 = p1 - p2;    // difference between points returns a vector v2
                  // based on the same coordinate system as p1
```

Note that the addition between two points is **NOT** possible and the difference between points returns a vector.

1.4.3.4 Other Operations

Exactly as for the 3D Vectors, the following operations are allowed:

- comparison of points
- scaling and division of points with a scalar
- dot and cross product with any type of vector

1.4.4 Example: LorentzVector Classes

As in the 3D case, typedef's are defined for user convenience. and can be used by including the header file `Math/Vector4D.h`. The following typedef's, defined in the header file `Math/Vector4Dfwd.h`, are available for the different instantiations of the template class `ROOT::Math::LorentzVector`:

- `ROOT::Math::XYZTVector` vector based on `x`, `y`, `z`, `t` coordinates (Cartesian) in double precision
- `ROOT::Math::XYZTVectorF` vector based on `x`, `y`, `z`, `t` coordinates (Cartesian) in float precision
- `ROOT::Math::PtEtaPhiEVector` vector based on `pt(rho)`, `eta`, `phi` and `E(t)` coordinates in double precision
- `ROOT::Math::PtEtaPhiMVector` vector based on `pt(rho)`, `eta`, `phi` and `M(t)` coordinates in double precision
- `ROOT::Math::PxPyPzMVector` vector based on `px`, `py`, `pz` and `M(mass)` coordinates in double precision

The metric used for all the LorentzVector is $(-, -, -, +)$.

1.4.4.1 Constructors and Assignment

The following declarations are available:

```
// create an empty vector (x=0, y=0, z=0, t=0)
XYZTVector    v1;
// vector with x=1, y=2, z=3, t=4
XYZTVector    v2(1,2,3,4);
// vector with pt=1, eta=2, phi=PI, E=5
PtEtaPhiEVector v3(1,2,PI,5);
```

Note that each type of vector is constructed by passing its coordinate representation, so a `XYZTVector(1,2,3,4)` is different from a `PtEtaPhiEVector(1,2,3,4)`. In addition, the Vector classes can be constructed by any vector, which implements the accessors `x()`, `y()`, `z()` and `t()`.

This can be another `ROOT::Math::LorentzVector` based on a different coordinate system or any vector of a different package, like the CLHEP `HepLorentzVector` that implements the required signature.

```

XYZTVector      v1(1,2,3,4);
PtEtaPhiEVector v2(v1);
CLHEP::HepLorentzVector q(1,2,3,4);
XYZTVector      v3(q);

```

1.4.4.2 Coordinate Accessors

All the same coordinate accessors are available through the interface of `ROOT::Math::LorentzVector`. For example:

```

//returns cartesian components for the cartesian vector v1
v1.X(); v1.X(); v1.Z(); v1.T();
//returns cartesian components for the cylindrical vector v2
v2.Px(); v2.Py(); v2.Pz(); v2.E();
//returns other components for the cartesian vector v1
v1.Pt(); v1.Eta(); v1.Phi(); v1.M()

```

In addition, all 4 vector coordinates can be retrieved with the `GetCoordinates` method:

```

double d[4];
//fill d array with (x,y,z,t) components of v1
v1.GetCoordinates(d);
//fill d array with (pt,eta,phi,e) components of v2
v2.GetCoordinates(d);
std::vector w(4);
//fill std::vector with (x,y,z,t)
v1.GetCoordinates(w.begin(),w.end());
//components of v1

```

To get information on all the coordinate accessors see the `ROOT::Math::LorentzVector` reference documentation.

1.4.4.3 Setter Methods

One can set only all the three coordinates via:

```

//sets the (x,y,z,t) for a XYZTVector
v1.SetCoordinates(c1,c2,c3,c4);
//sets pt,eta,phi,e for a PtEtaPhiEVector
v2.SetCoordinates(c1,c2,c3,c4);
//sets cartesian components for PtEtaPhiEVector
v2.SetXYZ(x,y,z,t);

```

Single coordinate setter methods are available for the basic vector coordinates, like `SetX()` for a `XYZTVector` or `SetPt()` for a `PtEtaPhiEVector`. Attempting to do a `SetX()` on a non-Cartesian vector will not compile.

```

XYZTVector      v1;
v1.SetX(1); //OK setting x for a cartesian vector
PtEtaPhiEVector v2;
v2.SetX(1); //ERROR: cannot set X for a non-cartesian
            //vector. Method will not compile.
v2.SetR(1)   // OK setting Pt for a PtEtaPhiEVector vector

```

In addition, there are setter methods from C arrays or iterators.

```

double d[4] = {1.,2.,3.,4.};
XYZTVector v;
//set (x,y,z,t) components of v using values from d
v.SetCoordinates(d);

```

or for example from an `std::vector` using the iterators

```

std::vector w(4);
//set (x,y,z,t) components of v using values from w
v.SetCoordinates(w.begin(),w.end());

```

1.4.4.4 Arithmetic Operations

The following operations are possible between Lorentz vectors classes, even of different coordinate system types: (v and w are two Lorentz vector of the same type, q is a generic Lorentz vector implementing $x()$, $y()$, $z()$ and $t()$, and a is a generic scalar type: double, float, int, etc.) .

```
v += q;
v -= q;
v = -q;
v *= a;
v /= a;
w = v + q;
w = v - q;
w = v * a;
w = a * v;
w = v / a;
```

1.4.4.5 Comparison

```
v == w;
v != w;
```

1.4.4.6 Other Methods

```
a = v.Dot(q); //dot product in metric(+,+,+,-) of 2 LorentzVectors
XYZVector s = v.Vect() //return the spatial components (x,y,z)
v.Beta(); //return beta and gamma value (vector must
v.Gamma() // be time-like otherwise result is meaningless)
XYZVector b = v.BoostToCM(); //return boost vector which will bring
//the Vector in its mas frame (P=0)
```

1.4.5 Example: Vector Transformations

Transformation classes are grouped in rotations (in three dimensions), Lorentz transformations and Poincarre transformations, which are translation/rotation combinations. Each group has several members which may model physically equivalent transformations but with different internal representations. All the classes are non-template and use double precision as the scalar type. The following types of transformation classes are defined:

3D rotations:

- `ROOT::Math::Rotation3D`, rotation described by a 3x3 matrix of doubles
- `ROOT::Math::EulerAngles` rotation described by the three Euler angles (`phi`, `theta` and `psi`) following the [Goldstein definition](#).
- `ROOT::Math::RotationZYX` rotation described by three angles defining a rotation first along the Z axis, then along the rotated Y' axis and then along the rotated X" axis.
- `ROOT::Math::AxisAngle`, rotation described by a vector (axis) and an angle
- `ROOT::Math::Quaternion`, rotation described by a quaternion (4 numbers)
- `ROOT::Math::RotationX`, specialized rotation along the X axis
- `ROOT::Math::RotationY`, specialized rotation along the Y axis
- `ROOT::Math::RotationZ`, specialized rotation along the Z axis

3D transformations (rotations + translations)

- `ROOT::Math::Transform3D`, (rotations and then translation) described by a 3x4 matrix (12 double numbers)
- `ROOT::Math::Translation3D` (only translation) described by a 3D Vector

Lorentz rotations and boosts

- `ROOT::Math::LorentzRotation`, 4D rotation (3D rotation plus a boost) described by a 4x4 matrix
- `ROOT::Math::Boost`, a Lorentz boost in an arbitrary direction and described by a 4x4 symmetric matrix (10 numbers)
- `ROOT::Math::BoostX`, a boost in the X axis direction
- `ROOT::Math::BoostY`, a boost in the Y axis direction
- `ROOT::Math::BoostZ`, a boost in the Z axis direction

1.4.5.1 Constructors

All rotations and transformations are default constructible (giving the identity transformation). All rotations are constructible taking a number of scalar arguments matching the number (and order of components).

```
Rotation3D  rI;           //a summy rotation (Identity matrix)
RotationX   rX(PI);       //a RotationX with an angle PI
EulerAngles rE(phi,theta,psi); //an Euler rotation with phi,
                           //theta,psi angles

XYZVector   u(ux,uy,uz);
AxisAngle   rA(u,delta);  //a rotation based on direction u,
                           //angle delta
```

In addition, all rotations and transformations (other than the axial rotations) and transformations are constructible from `(begin,end)` iterators or from pointers behave like iterators.

```
double      data[9];
//create a rotation from a rotation matrix
Rotation3D  r(data,data+9);
std::vector w(12);
//create Transform3D from std::vector content
Transform3D t(w.begin(),w.end());
```

All rotations, except the axial rotations, are constructible and assigned from any other type of rotation (including the axial):

```
//create a rotation 3D from a rotation along X axis of angle PI
Rotation3D  r(ROOT::Math::RotationX(PI));

//construct an Euler rotation from A Rotation3D
EulerAngles r2(r);

//assign an Axis rotation from an Euler Rotation
AxisAngle   r3; r3 = r2;
```

Transform3D (rotation + translation) can be constructed from a rotation and a translation vector:

```
Rotation3D  r;
XYZVector   v;
Transform3D t1(r,v);  //construct from rotation and then
                      //translation
Transform3D t2(v,r);  //construct inverse from first translation
                      //then rotation
Transform3D t3(r);    //construct from only a rotation
                      //(zero translation)
Transform3D t4(v);    //construct from only translation
                      //(identity rotation)
```

1.4.5.2 Operations

All transformations can be applied to vector and points using the operator `*` or using the operator `()`

```
XYZVector  v1(...);
Rotation3D r(...);
XYZVector  v2 = r*v1;           //rotate vector v1 using r
v2 = r(v1);                     //equivalent
```

Transformations can be combined using the operator `*`. Rotation, translation and **Transform3D** classes can be all combined with the operator `*`. The result of a combination of a rotation and a translation will be a **Transform3D** class. Note that the rotations are not commutative, the order is then important.

```
Rotation3D r1(...);
Rotation3D r2(...);
Rotation3D r3 = r2*r1; //a combine rotation r3 by
                       //applying first r1 then r2
```

We can combine rotations of different types, like **Rotation3D** with any other type of rotations. The product of two different axial rotations returns a **Rotation3D**:

```
RotationX  rx(1.);
RotationY  ry(2.);
Rotation3D r = ry * rx; //rotation along X and then Y axis
```

It is also possible to invert all the transformation or return their inverse:

```
Rotation3D r1(...);
r1.Invert(); //invert the rotation modifying its content
Rotation3D r2 =r1.Inverse(); //return the inverse in a new
                             //rotation class
```

We have used rotation as examples, but all these operations can be applied to all the transformation classes.

1.4.5.3 Set/GetComponents Methods

Common methods to all transformations are **Get** and **SetComponents**. They can be used to retrieve all the scalar values on which the transformation is based.

```
RotationX  rx;
rx.SetComponents(1.); //set aple of the X rotation
double d[9] = {...};
Rotation3D r;
r.SetComponents(d,d+9); //set 9 components of 3D rotation
double d[16];
LorentzRotation lr;
lr.GetComponents(d,d+16); //get 16 components of a LorentzRotation
TMatrixD(3,4) m;
Transform3D t;
t.GetComponens(m); //fill 3x4 matrix with components of t
```

The **GetComponents** and **SetComponents** methods can be used with a signature based iterators or by using any foreign matrix which implements the **operator(i,j)** or a different signatures depending on the transformation type. For more details on all methods see the reference documentation of any specific transformation class.

1.4.6 Example with External Packages

1.4.6.1 Connection to Linear Algebra Classes

It is possible to use the vector and rotation classes together with the linear algebra classes and to set and get the contents of any 3D or 4D vector from a linear algebra vector class which implements an iterator or something which behaves like an iterator. For example a pointer to a C array (double*) behaves like an iterator. It is then assumed that the coordinates, like (x,y,z) will be stored contiguously.

```
TVectorD   r2(N); //ROOT Linear Algebra Vector containing
               //many vectors

XYZVector  v2;
//construct vector from x=r[INDEX], y=r[INDEX+1], z=r[INDEX+2]
v2.SetCoordinates(&r2[INDEX],&r2[index]+3);
```

To fill a linear algebra vector from a 3D or 4D vector, with `GetCoordinates()` one can get the internal coordinate data.

```
HepVector   c(3);           //CLHEP Linear algebra vector
//fill HepVector c with c[0]=x, c[1]=y, c[2]=z
v2.GetCoordinates(&c[0],&c[index]+3)
```

or using `TVectorD`:

```
double      *data[3];
v2.GetCoordinates(data,data+3);
TVectorD    r1(3,data); //create a new Linear Algebra vector
                       //copying the data
```

In the case of transformations, constructor and method to set/get components exist with linear algebra matrices. The requisite is that the matrix data are stored, for example in the case of a Lorentz rotation, from (0,0) thru (3,3)

```
TMatrixD(4,4)   m;
LorentzRotation r(m);           //create Lorentz r
```

1.4.6.2 Connection to Other Vector Classes

The 3D and 4D vectors of the `GenVector` package can be constructed and assigned from any vector which satisfies the following requisites:

- for 3D vectors implementing the `x()`, `y()` and `z()` methods
- for Lorentz vectors implementing the `x()`, `y()`, `z()` and `t()` methods.

```
CLHEP::Hep3Vector hv;
XYZVector          v1(hv);           //create 3D vector from
                                     //CLHEP 3D Vector

HepGeom::Point3D   hp;
XYZPoint           p1(hp);           //create a 3D p
```

1.5 MathMore Library

The `MathMore` library provides an advanced collection of functions and C++ classes for numerical computing. This is an extension of the functionality provided by the `MathCore` library. The current set includes:

- Special functions (see Special Functions in MathMore)
- Mathematical functions used in statistics such as probability density functions, cumulative distributions functions and their inverse.

- Numerical algorithms for one dimensional functions based on implementation of the GNU Scientific Library (GSL):
- Numerical integration using the class `ROOT::Math::Integrator` which is based on the Adaptive integration algorithms of QUADPACK
- Numerical differentiation via `ROOT::Math::Derivator`
- Root finder via `ROOT::Math::RootFinder` which uses different solver algorithms from GSL
- Minimization via `ROOT::Math::Minimizer1D`
- Interpolation via `ROOT::Math::Interpolation`. All the GSL interpolation types are supported
- Function approximation based on Chebyshev polynomials via the class `ROOT::Math::Chebyshev`
- Random number generators and distributions
- Polynomial evaluation and root solvers

The mathematical functions are implemented as a set of free functions in the namespace `ROOT::Math`. The naming used for the special functions is the same proposed for the C++ standard (see C++ standard extension [proposal document](#)). The `MathCore` library is implemented wrapping in C++ the GNU Scientific Library (). Building `MathMore` requires a version of GSL larger or equal 1.8. The source code of `MathMore` is distributed under the GNU General Public License.

`MathMore` (and its ROOT Cling dictionary) can be built within ROOT whenever a GSL library is found in the system. The GSL library and header file location can be specified in the ROOT configure script, by doing:

```
./configure --with-gsl-incdir=... --with-gsl-libdir=...
```

`MathMore` can be built also a stand-alone library (without requiring ROOT) downloading the tar file from the Web at this link. In this case the library will not contain the dictionary information and therefore cannot be used interactively

More information on the classes and functions present in `MathMore` is available in the [online reference documentation](#).

1.6 Mathematical Functions

The mathematical functions are present in both `MathCore` and `MathMore` libraries. All mathematical functions are implemented as free functions in the namespace `ROOT::Math`. The most used functions are in the `MathCore` library while the others are in the `MathMore` library. The functions in `MathMore` are all using the implementation of the GNU Scientific Library (GSL). The naming of the special functions is the same defined in the C++ [Technical Report on Standard Library extensions](#). The special functions are defined in the header file `Math/SpecFunc.h`.

1.6.1 Special Functions in MathCore

- `ROOT::Math::beta(double x, double y)` - evaluates the beta function:

$$B(x, y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$$

- `double ROOT::Math::erf(double x)` - evaluates the error function encountered in integrating the normal distribution:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

- `double ROOT::Math::erfc(double x)` - evaluates the complementary error function:

$$\text{erfc}(x) = 1 - \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$$

- `double ROOT::Math::tgamma(double x)` - calculates the gamma function:

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$$

1.6.2 Special Functions in MathMore

- `double ROOT::Math::assoc_legendre(unsigned l,unsigned m,double x)` -computes the associated Legendre polynomials (with $m \geq 0$, $l \geq m$ and $|x| < 1$):

$$P_l^m(x) = (1-x^2)^{m/2} \frac{d^m}{dx^m} P_l(x)$$

- `double ROOT::Math::comp_ellint_1(double k)` - calculates the complete elliptic integral of the first kind (with $0 \leq k^2 \leq 1$):

$$K(k) = F(k, \pi/2) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1-k^2 \sin^2 \theta}}$$

- `double ROOT::Math::comp_ellint_2(double k)` - calculates the complete elliptic integral of the second kind (with $0 \leq k^2 \leq 1$):

$$E(k) = E(k, \pi/2) = \int_0^{\pi/2} \sqrt{1-k^2 \sin^2 \theta} d\theta$$

- `double ROOT::Math::comp_ellint_3(double n,double k)` - calculates the complete elliptic integral of the third kind (with $0 \leq k^2 \leq 1$):

$$\Pi(n, k, \pi/2) = \int_0^{\pi/2} \frac{d\theta}{(1-n \sin^2 \theta) \sqrt{1-k^2 \sin^2 \theta}}$$

- `double ROOT::Math::conf_hyperg(double a,double b,double z)` - calculates the confluent hyper-geometric functions of the first kind:

$${}_1F_1(a; b; z) = \frac{\Gamma(b)}{\Gamma(a)} \sum_{n=0}^{\infty} \frac{\Gamma(a+n)}{\Gamma(b+n)} \frac{z^n}{n!}$$

- `double ROOT::Math::conf_hypergU(double a,double b,double z)` - calculates the confluent hyper-geometric functions of the second kind, known also as Kummer function of the second type. It is related to the confluent hyper-geometric function of the first kind:

$$U(a, b, z) = \frac{\pi}{\sin \pi b} \left[\frac{{}_1F_1(a, b, z)}{\Gamma(a-b+1)} - \frac{z^{1-b} {}_1F_1(a-b+1, 2-b, z)}{\Gamma(a)} \right]$$

- `double ROOT::Math::cyl_bessel_i(double nu,double x)` - calculates the modified Bessel function of the first kind, also called regular modified (cylindrical) Bessel function:

$$I_\nu(x) = i^{-\nu} J_\nu(ix) = \sum_{k=0}^{\infty} \frac{(\frac{1}{2}x)^{\nu+2k}}{k! \Gamma(\nu+k+1)}$$

- `double ROOT::Math::cyl_bessel_j(double nu,double x)` - calculates the (cylindrical) Bessel function of the first kind, also called regular (cylindrical) Bessel function:

$$J_\nu(x) = \sum_{k=0}^{\infty} \frac{(-1)^k (\frac{1}{2}x)^{\nu+2k}}{k! \Gamma(\nu+k+1)}$$

- `double ROOT::Math::cyl_bessel_k(double nu,double x)` - calculates the modified Bessel function of the second kind, also called irregular modified (cylindrical) Bessel function for $x > 0$, $\nu > 0$:

$$K_\nu(x) = \frac{\pi}{2} i^{\nu+1} (J_\nu(ix) + iN_\nu(ix)) = \begin{cases} \frac{\pi}{2} \frac{I_{-\nu}(x) - I_\nu(x)}{\sin \nu \pi} & \text{for non-integral } \nu \\ \frac{\pi}{2} \lim_{\mu \rightarrow \nu} \mu \frac{I_{-\mu}(x) - I_\mu(x)}{\sin \mu \pi} & \text{for integral } \nu \end{cases}$$

- `double ROOT::Math::cyl_neumann(double nu,double x)` - calculates the (cylindrical) Bessel function of the second kind, also called irregular (cylindrical) Bessel function or (cylindrical) Neumann function:

$$N_\nu(x) = Y_\nu(x) = \begin{cases} \frac{J_\nu \cos \nu \pi - J_{-\nu}(x)}{\sin \nu \pi} & \text{for non-integral } \nu \\ \lim_{\mu \rightarrow \nu} \mu \frac{J_\mu \cos \mu \pi - J_{-\mu}(x)}{\sin \mu \pi} & \text{for integral } \nu \end{cases}$$

- `double ROOT::Math::ellint_1(double k,double phi)` - calculates incomplete elliptic integral of the first kind (with $0 \leq k^2 \leq 1$):

$$K(k) = F(k, \pi/2) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1-k^2 \sin^2 \theta}}$$

- `double ROOT::Math::ellint_2(double k, double phi)` - calculates the complete elliptic integral of the second kind (with $0 \leq k^2 \leq 1$):

$$E(k) = E(k, \pi/2) = \int_0^{\pi/2} \sqrt{1 - k^2 \sin^2 \theta} d\theta$$

- `double ROOT::Math::ellint_3(double n, double k, double phi)` - calculates the complete elliptic integral of the third kind (with $0 \leq k^2 \leq 1$):

$$\Pi(n, k, \pi/2) = \int_0^{\pi/2} \frac{d\theta}{(1 - n \sin^2 \theta) \sqrt{1 - k^2 \sin^2 \theta}}$$

- `double ROOT::Math::expint(double x)` - calculates the exponential integral:

$$Ei(x) = - \int_{-x}^{\infty} \frac{e^{-t}}{t} dt$$

- `double ROOT::Math::hyperg(double a, double b, double c, double x)` - calculates Gauss' hyper-geometric function:

$${}_2F_1(a, b; c; x) = \frac{\Gamma(c)}{\Gamma(a)\Gamma(b)} \sum_{n=0}^{\infty} \frac{\Gamma(a+n)\Gamma(b+n)}{\Gamma(c+n)} \frac{x^n}{n!}$$

- `double ROOT::Math::legendre(unsigned l, double x)` - calculates the Legendre polynomials for $l \geq 0$, $|x| \leq 1$ in the Rodrigues representation:

$$P_l(x) = \frac{1}{2^l l!} \frac{d^l}{dx^l} (x^2 - 1)^l$$

- `double ROOT::Math::riemann_zeta(double x)` - calculates the Riemann zeta function:

$$\zeta(x) = \begin{cases} \sum_{k=1}^{\infty} k^{-x} & \text{for } x > 1 \\ 2^x \pi^{x-1} \sin\left(\frac{1}{2}\pi x\right) \Gamma(1-x) \zeta(1-x) & \text{for } x < 1 \end{cases}$$

- `double ROOT::Math::sph_bessel(unsigned n, double x)` - calculates the spherical Bessel functions of the first kind (also called regular spherical Bessel functions):

$$j_n(x) = \sqrt{\frac{\pi}{2x}} J_{n+1/2}(x)$$

- `double ROOT::Math::sph_neumann(unsigned n, double x)` - calculates the spherical Bessel functions of the second kind (also called irregular spherical Bessel functions or spherical Neumann functions):

$$n_n(x) = y_n(x) = \sqrt{\frac{\pi}{2x}} N_{n+1/2}(x)$$

1.6.3 Probability Density Functions (PDF)

Probability density functions of various distributions. All the functions, apart from the discrete ones, have the extra location parameter `x0`, which by default is zero. For example, in the case of a gaussian pdf, `x0` is the mean, `mu`, of the distribution. All the probability density functions are defined in the header file `Math/DistFunc.h` and are part of the `MathCore` libraries. The definition of these functions is documented in the [reference doc for statistical functions](#):

```
double ROOT::Math::beta_pdf(double x, double a, double b);
double ROOT::Math::binomial_pdf(unsigned int k, double p, unsigned int n);
double ROOT::Math::breitwigner_pdf(double x, double gamma, double x0=0);
double ROOT::Math::cauchy_pdf(double x, double b=1, double x0=0);
double ROOT::Math::chisquared_pdf(double x, double r, double x0=0);
double ROOT::Math::exponential_pdf(double x, double lambda, double x0=0);
double ROOT::Math::fdistribution_pdf(double x, double n, double m, double x0=0);
double ROOT::Math::gamma_pdf(double x, double alpha, double theta, double x0=0);
double ROOT::Math::gaussian_pdf(double x, double sigma, double x0=0);
double ROOT::Math::landau_pdf(double x, double s, double x0=0);
double ROOT::Math::lognormal_pdf(double x, double m, double s, double x0=0);
double ROOT::Math::normal_pdf(double x, double sigma, double x0=0);
double ROOT::Math::poisson_pdf(unsigned int n, double mu);
double ROOT::Math::tdistribution_pdf(double x, double r, double x0=0);
double ROOT::Math::uniform_pdf(double x, double a, double b, double x0=0);
```

1.6.4 Cumulative Distribution Functions (CDF)

For all the probability density functions, we have the corresponding cumulative distribution functions and their complements. The functions with extension `_cdf` calculate the lower tail integral of the probability density function:

$$D(x) = \int_{-\infty}^x p(x') dx'$$

while those with the `cdf_c` extension calculate the upper tail of the probability density function, so-called in statistics the survival function. For example, the function:

```
double ROOT::Math::gaussian_cdf(double x, double sigma, double x0=0);
```

evaluates the lower tail of the Gaussian distribution:

$$D(x) = \int_{-\infty}^x \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x'-x_0)^2/2\sigma^2} dx'$$

while the function:

```
double ROOT::Math::gaussian_cdf_c(double x, double sigma, double x0=0);
```

evaluates the upper tail of the Gaussian distribution:

$$D(x) = \int_x^{+\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x'-x_0)^2/2\sigma^2} dx'$$

The cumulative distributions functions are defined in the header file `Math/ProbFunc.h`. The majority of the CDF's are present in the `MathCore`, apart from the `chisquared`, `fdistribution`, `gamma` and `tdistribution`, which are in the `MathMore` library.

1.6.4.1 Inverse of the Cumulative Distribution Functions(Quantiles)

For almost all the cumulative distribution functions (`_cdf`) and their complements (`_cdf_c`) present in the library, we provide the inverse functions. The inverse of the cumulative distribution function is called in statistics quantile function. The functions with the extension `_quantile` calculate the inverse of the cumulative distribution function (lower tail integral of the probability density function), while those with the `quantile_c` extension calculate the inverse of the complement of the cumulative distribution (upper tail integral). All the inverse distributions are in the `MathMore` library and are defined in the header file `Math/ProbFuncInv.h`.

The following picture illustrates the available statistical functions (PDF, CDF and quantiles) in the case of the normal distribution.

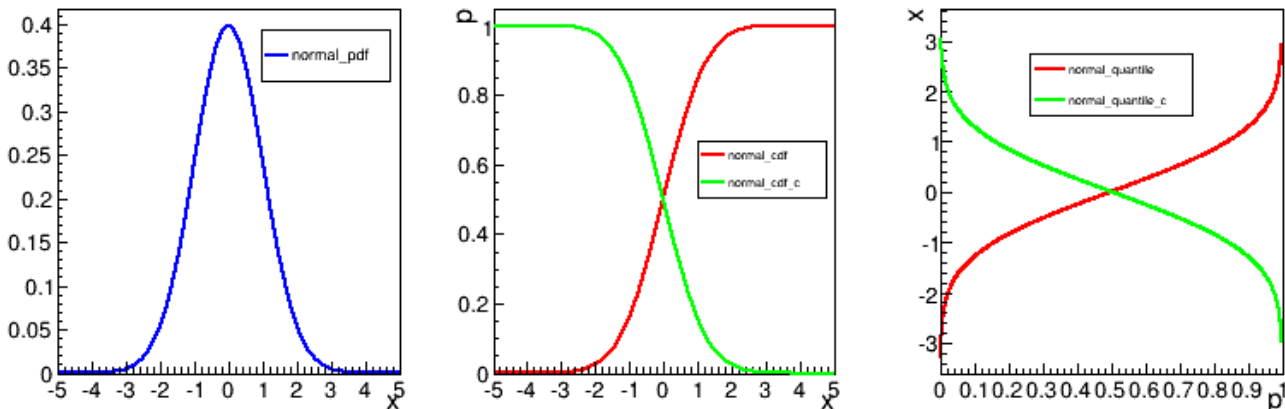


Figure 1.2: PDF, CDF and quantiles in the case of the normal distribution

1.7 Linear Algebra: SMatrix Package

The ROOT Linear algebra package is documented in a separate chapter (see “Linear Algebra in ROOT”). **SMatrix** is a C++ package, for high performance vector and matrix computations. It has been introduced in ROOT v5.08. It is optimized for describing small matrices and vectors and It can be used only in problems when the size of the matrices is known at compile time, like in the tracking reconstruction of physics experiments. It is based on a C++ technique, called expression templates, to achieve an high level optimization. The C++ templates can be used to implement vector and matrix expressions such that these expressions can be transformed at compile time to code which is equivalent to hand optimized code in a low-level language like FORTRAN or C (see for example T. Veldhuizen, Expression Templates, C++ Report, 1995).

The **SMatrix** has been developed initially by T. Glebe in Max-Planck-Institut, Heidelberg, as part of the **HeraB** analysis framework. A subset of the original package has been now incorporated in the ROOT distribution, with the aim to provide a stand-alone and high performance matrix package. The API of the current package differs from the original one, in order to be compliant to the ROOT coding conventions.

SMatrix contains the generic **ROOT::Math::SMatrix** and **ROOT::Math::SVector** classes for describing matrices and vectors of arbitrary dimensions and of arbitrary type. The classes are templated on the scalar type and on the size, like number of rows and columns for a matrix. Therefore, the matrix/vector dimension has to be known at compile time. An advantage of using the dimension as template parameters is that the correctness of dimension in the matrix/vector operations can be checked at compile time.

SMatrix supports, since ROOT v5.10, symmetric matrices using a storage class (**ROOT::Math::MatRepSym**) which contains only the $N*(N+1)/2$ independent element of a $N \times N$ symmetric matrix. It is not in the mandate of this package to provide complete linear algebra functionality. It provides basic matrix and vector functions such as matrix-matrix, matrix-vector, vector-vector operations, plus some extra functionality for square matrices, like inversion and determinant calculation. The inversion is based on the optimized Cramer method for squared matrices of size up to 6×6 .

The **SMatrix** package contains only header files. Normally one does not need to build any library. In the ROOT distribution a library, **libSMatrix** is produced with the C++ dictionary information for squared and symmetric matrices and vectors up to dimension 7 and based on **Double_t**, **Float_t** and **Double32_t**. The following paragraphs describe the main characteristics of the matrix and vector classes. More detailed information about the **SMatrix** classes API is available in the [online reference documentation](#).

1.7.1 Example: Vector Class (SVector)

The template class **ROOT::Math::SVector** represents n -dimensional vectors for objects of arbitrary type. This class has 2 template parameters, which define at compile time, its properties: 1) type of the contained elements (for example *float* or *double*); 2) size of the vector. The use of this dictionary is mandatory if one want to use **Smatrix** in Cling and with I/O.

1.7.1.1 Creating a Vector

The following constructors are available to create a vector:

- Default constructor for a zero vector (all elements equal to zero).
- Constructor (and assignment) from a vector expression, like $v = p * q + w$. Due to the expression template technique, no temporary objects are created in this operation.
- Constructor by passing directly the elements. This is possible only for vectors up to size 10.
- Constructor from an iterator copying the data referred by the iterator. It is possible to specify the *begin* and *end* of the iterator or the *begin* and the *size*. Note that for the Vector the iterator is not generic and must be of type T^* , where T is the type of the contained elements.

In the following example we assume that we are using the namespace **ROOT::Math**

```
//create an empty vector of size 3 ( v[0]=v[1]=v[2]=0)
SVector<double,3> v;
double d[3] = {1,2,3};
SVector<double,3> v(d,3); //create a vector from a C array
```


1.7.1.2 Accessing and Setting Methods

The single vector elements can be set or retrieved using the `operator[i]`, `operator(i)` or the iterator interface. Notice that the index starts from zero and not from one as in FORTRAN. Also no check is performed on the passed index. The full vector elements can be set also by using the `SetElements` function passing a generic iterator.

```
double x = m(i);           // return the i-th element
x = *(m.begin()+i);       // return the i-th element
v[0] = 1;                 // set the first element
v(1) = 2;                 // set the second element
*(v.begin()+3) = 3;       // set the third element
std::vector<double> w(3);

// set vector elements from a std::vector<double>::iterator
v.SetElements(w.begin(),w.end());
```

In addition there are methods to place a sub-vector in a vector. If the size of the sub-vector is larger than the vector size a static assert (a compilation error) is produced.

```
SVector<double,N> v;
SVector<double,M> w;
// M <= N otherwise a compilation error is obtained later
// place a vector of size M starting from
// element ioff, v[ioff+i]=w[i]
v.Place_at(w,ioff);
// return a sub-vector of size M starting from
// v[ioff]: w[i]=v[ioff+i]
w = v.Sub < SVector<double,M> > (ioff);
```

For the vector functions see later in the Matrix and Vector Operators and Functions paragraph.

1.7.2 Example: Matrix Class (SMatrix)

The template class `ROOT::Math::SMatrix` represents a matrix of arbitrary type with `nrows` x `ncol` dimension. The class has 4 template parameters, which define at compile time, its properties:

- type of the contained elements, `T`, for example float or double;
- number of rows;
- number of columns;
- representation type. This is a class describing the underlined storage model of the Matrix. Presently exists only two types of this class:
- `ROOT::Math::MatRepStd` for a general `nrows` x `ncols` matrix. This class is itself a template on the contained type `T`, the number of rows and the number of columns. Its data member is an array `T[nrows*ncols]` containing the matrix data. The data are stored in the row-major C convention. For example, for a matrix `M`, of size `3x3`, the data `{a0,a1,...,a8}` are stored in the following order:

$$M = \begin{pmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{pmatrix}$$

- `ROOT::Math::MatRepSym` for a symmetric matrix of size `NxN`. This class is a template on the contained type and on the symmetric matrix size `N`. It has as data member an array of type `T` of size `N*(N+1)/2`, containing the lower diagonal block of the matrix. The order follows the lower diagonal block, still in a row-major convention. For example for a symmetric `3x3` matrix the order of the 6 independent elements `{a0,a1,...,a5}` is:

$$M = \begin{pmatrix} a_0 & a_1 & a_3 \\ a_1 & a_2 & a_4 \\ a_3 & a_4 & a_5 \end{pmatrix}$$

1.7.2.1 Creating a Matrix

The following constructors are available to create a matrix:

- Default constructor for a zero matrix (all elements equal to zero).
- Constructor of an identity matrix.
- Copy constructor (and assignment) for a matrix with the same representation, or from a different one when possible, for example from a symmetric to a general matrix.
- Constructor (and assignment) from a matrix expression, like $D=A*B+C$. Due to the expression template technique, no temporary objects are created in this operation. In the case of an operation like $A=A*B+C$, a temporary object is needed and it is created automatically to store the intermediary result in order to preserve the validity of this operation.
- Constructor from a generic STL-like iterator copying the data referred by the iterator, following its order. It is both possible to specify the begin and end of the iterator or the begin and the size. In case of a symmetric matrix, it is required only the triangular block and the user can specify whether giving a block representing the lower (default case) or the upper diagonal part.

Here are some examples on how to create a matrix. We use typedef's in the following examples to avoid the full C++ names for the matrix classes. Notice that for a general matrix the representation has the default value, `ROOT::Math::MatRepStd`, and it is not needed to be specified. Furthermore, for a general square matrix, the number of column may be as well omitted.

```
// typedef definitions used in the following declarations
typedef ROOT::Math::SMatrix<double,3>      SMatrix33;
typedef ROOT::Math::SMatrix<double,2>      SMatrix22;
typedef ROOT::Math::SMatrix<double,3,3>    SMatrix33;
ROOT::Math::MatRepSym<double,3>> SMatrixSym3;
typedef ROOT::Math::SVector<double,2>      SVector2;
typedef ROOT::Math::SVector<double,3>      SVector3;
typedef ROOT::Math::SVector<double,6>      SVector6;
SMatrix33  m0;                               // create a zero 3x3 matrix
// create an 3x3 identity matrix
SMatrix33  i = ROOT::Math::SMatrixIdentity();
double  a[9] = {1,2,3,4,5,6,7,8,9};         // input matrix data
// create a matrix using the a[] data
SMatrix33  m(a,9);                          // this will produce the 3x3 matrix
//      ( 1  2  3 )
//      ( 4  5  6 )
//      ( 7  8  9 )
```

Example to fill a symmetric matrix from an `std::vector`:

```
std::vector<double> v(6);
for (int i = 0; i<6; ++i) v[i] = double(i+1);
SMatrixSym3  s(v.begin(),v.end())           // this will produce the
// symmetric matrix
//      ( 1  2  4 )
//      ( 2  3  5 )
//      ( 4  5  6 )
// create a general matrix from a symmetric matrix (the opposite
// will not compile)
SMatrix33  m2 = s;
```

1.7.2.2 Accessing and Setting Methods

The matrix elements can be set using the `operator()(irow,icol)`, where `irow` and `icol` are the row and column indexes or by using the iterator interface. Notice that the indexes start from zero and not from one as in FORTRAN. Furthermore, all the matrix elements can be set also by using the `SetElements` function passing a generic iterator. The elements can be accessed by the same methods as well as by using the function `ROOT::Math::SMatrix::apply`. The `apply(i)` has exactly the same behavior for general and symmetric matrices; in contrast to the iterator access methods which behave differently (it follows the data order).

```

SMatrix33 m;
m(0,0) = 1;           // set the element in first row and first column
*(m.begin()+1) = 2;   // set the second element (0,1)
double d[9]={1,2,3,4,5,6,7,8,9};
m.SetElements(d,d+9); // set the d[] values in m
double x = m(2,1);    // return the element in 3
x = m.apply(7);       // return the 8-th element (row=2,col=1)
x = *(m.begin()+7);   // return the 8-th element (row=2,col=1)
// symmetric matrices
//(note the difference in behavior between apply and the iterators)
x = *(m.begin()+4)    // return the element (row=2,col=1)
x = m.apply(7);       // returns again the (row=2,col=1) element

```

There are methods to place and/or retrieve **ROOT::Math::SVector** objects as rows or columns in (from) a matrix. In addition one can put (get) a sub-matrix as another **ROOT::Math::SMatrix** object in a matrix. If the size of the sub-vector or sub-matrix is larger than the matrix size a static assert (a compilation error) is produced. The non-const methods are:

```

SMatrix33 m;
SVector2 v2(1,2);
// place a vector in the first row from
// element (0,1) : m(0,1)=v2[0]
m.Place_in_row(v2,0,1);
// place the vector in the second column from
// (0,1) : m(0,1) = v2[0]
m.Place_in_col(v2,0,1);
SMatrix22 m2;
// place m2 in m starting from the
// element (1,1) : m(1,1) = m2(0,0)
m.Place_at(m2,1,1);
SVector3 v3(1,2,3);
// set v3 as the diagonal elements
// of m : m(i,i) = v3[i] for i=0,1,2
m.SetDiagonal(v3)

```

The const methods retrieving contents (getting slices of a matrix) are:

```

a = {1,2,3,4,5,6,7,8,9};
SMatrix33 m(a,a+9);
SVector3 irow = m.Row(0); // return as vector the first row
SVector3 jcol = m.Col(1); // return as vector the second column

// return a slice of the first row from
// (0,1): r2[0]= m(0,1); r2[1]=m(0,2)
SVector2 r2 = m.SubRow<SVector2>(0,1);
// return a slice of the second column from
// (0,1): c2[0] = m(0,1); c2[1] = m(1,1)
SVector2 c2 = m.SubCol<SVector2>(1,0);

// return a sub-matrix 2x2 with the upper left corner at(1,1)
SMatrix22 subM = m.Sub<SMatrix22>(1,1);

// return the diagonal element in a SVector
SVector3 diag = m.Diagonal();

// return the upper(lower) block of the matrix m
SVector6 vub = m.UpperBlock(); // vub = [ 1, 2, 3, 5, 6, 9 ]
SVector6 vlb = m.LowerBlock(); // vlb = [ 1, 4, 5, 7, 8, 9 ]

```

1.7.2.3 Linear Algebra Matrix Functions (Inversion, Determinant)

Only limited linear algebra functionality is available for **SMatrix**. It is possible for squared matrices $N \times N$, to find the inverse or to calculate the determinant. Different inversion algorithms are used if the matrix is smaller than 6×6 or if

it is symmetric. In the case of a small matrix, a faster direct inversion is used. For a large ($N > 6$) symmetric matrix the Bunch-Kaufman diagonal pivoting method is used while for a large ($N > 6$) general matrix an LU factorization is performed using the same algorithm as in the CERNLIB routine `dinv`.

```
// Invert a NxN matrix.
// The inverted matrix replaces the existing one if the
// result is successful
bool ret = m.Invert();           // return the inverse matrix of m.

// If the inversion fails ifail is different than zero ???
int ifail = 0;
ifail = m.Inverse(ifail);

// determinant of a square matrix - calculate the determinant
// modifying the matrix content and returns it if the calculation
// was successful
double det;
bool ret = m.Det(det);

// calculate determinant by using a temporary matrix; preserves
// matrix content
bool ret = n.Det2(det);
```

1.7.3 Example: Matrix and Vector Functions and Operators

1.7.3.1 Matrix and Vector Operators

The `ROOT::Math::SVector` and `ROOT::Math::SMatrix` classes define the following operators described below. The `m1`, `m2`, `m3` are vectors or matrices of the same type (and size) and `a` is a scalar value:

```
m1 == m2 // returns whether m1 is equal to m2
           // (element by element comparison)
m1 != m2 // returns whether m1 is NOT equal to m2
           // (element by element comparison)
m1 < m2  // returns whether m1 is less than m2
           // (element wise comparison)
m1 > m2  // returns whether m1 is greater than m2
           // (element wise comparison)

// in the following m1 and m3 can be general and m2 symmetric,
// but not vice-versa

m1 += m2 // add m2 to m1
m1 -= m2 // subtract m2 to m1
m3 = m1 + m2 // addition
m1 - m2 // subtraction

// Multiplication and division via a scalar value a

m3 = a*m1; m3 = m1*a; m3 = m1/a;
```

Vector-Vector multiplication: The operator `*` defines an element by element multiplication between vectors. For the standard vector-vector algebraic multiplication returning a scalar, `vTv` (dot product), one must use the `ROOT::Math::Dot` function. In addition, the **Cross** (only for vector sizes of 3), `ROOT::Math::Cross`, and the Tensor product, `ROOT::Math::TensorProd`, are defined.

Matrix - Vector multiplication: The operator `*` defines the matrix-vector multiplication: $y_i = \sum_j M_{i,j} x_j$. The operation compiles only if the matrix and the vectors have the right sizes.

```
//M is a N1xN2 matrix, x is a N2 size vector, y is a N1 size vector
y = M * x
```

Matrix - Matrix multiplication: The operator `*` defines the matrix-matrix multiplication: $C_{i,j} = \sum_k A_{i,k} B_{k,j}$.

```
// A is a N1xN2 matrix, B is a N2xN3 matrix and C is a N1xN3 matrix
C = A * B
```

The operation compiles only if the matrices have the right size. In the case that **A** and **B** are symmetric matrices, **C** is a general one, since their product is not guaranteed to be symmetric.

1.7.4 Matrix and Vector Functions

The most used matrix functions are:

- `ROOT::Math::Transpose(M)` returns the transpose matrix **MT**
- `ROOT::Math::Similarity(v,M)` returns the scalar value resulting from the matrix-vector product **vTMv**
- `ROOT::Math::Similarity(U,M)` returns the matrix resulting from the product: **U M UT**. If **M** is symmetric, the returned resulting matrix is also symmetric
- `ROOT::Math::SimilarityT(U,M)` returns the matrix resulting from the product: **UT M U**. If **M** is symmetric, the returned resulting matrix is also symmetric

The major vector functions are:

- `ROOT::Math::Dot(v1,v2)` returns the scalar value resulting from the vector dot product
- `ROOT::Math::Cross(v1,v2)` returns the vector cross product for two vectors of size 3. Note that the **Cross** product is not defined for other vector sizes
- `ROOT::Math::Unit(v)` returns unit vector. One can use also the `v.Unit()` method.
- `ROOT::Math::TensorProd(v1,v2)` returns a general matrix **Mof** of size **N1xN2** resulting from the tensor product between the vector **v1** of size **N1** and **v2** of size **N2**:

For a list of all the available matrix and vector functions see the **SMatrix** online reference documentation.

1.7.4.1 Matrix and Vector I/O

One can print (or write in an output stream) Vectors and Matrices) using the **Print** method or the `<<` operator:

```
// m is a SMatrix or a SVector object
m.Print(std::cout);
std::cout << m << std::endl;
```

In the ROOT distribution, the Cling dictionary is generated for **SMatrix** and **SVector** for for **Double_t**, **Float_t** and **Double32_t** up to dimension 7. This allows the possibility to store them in a ROOT file.

1.8 Minuit2 Package

Minuit2 is a new object-oriented implementation, written in C++, of the popular **MINUIT** minimization package. Compared with the **TMinuit** class, which is a direct conversion from FORTRAN to C++, **Minuit2** is a complete redesign and re-implementation of the package. This new version provides all the functionality present in the old FORTRAN version, with almost equivalent numerical accuracy and computational performances. Furthermore, it contains new functionality, like the possibility to set single side parameter limits or the **FUMILI** algorithm (see “**FUMILI** Minimization Package” in “Fitting Histograms” chapter), which is an optimized method for least square and log likelihood minimizations. **Minuit2** has been originally developed by M. Winkler and F. James in the **SEAL** project. More information can be found on the [MINUIT Web Site](http://www.cern.ch/minuit/doc/doc.html) and in particular at the following documentation page at <http://www.cern.ch/minuit/doc/doc.html>.

The API has been then changed in this new version to follow the ROOT coding convention (function names starting with capital letters) and the classes have been moved inside the namespace `ROOT::Minuit2`. In addition, the ROOT distribution contains classes needed to integrate **Minuit2** in the ROOT framework, like **TFitterMinuit** and **TFitterFumili**. **Minuit2** can be used in ROOT as another fitter plug-in. For example for using it in histogram fitting, one only needs to do:

```
TVirtualFitter::SetDefaultFitter("Minuit2"); //or Fumili2 for the
// FUMILI algorithm histogram->Fit();
```

For minimization problem, providing an FCN function to minimize, one can do:

```
TVirtualFitter::SetDefaultFitter("Minuit2");
TVirtualFitter * minuit2 = TVirtualFitter::Fitter(0,2);
```

Then set the parameters, the FCN and minimize using the **TVirtualFitter** methods: **SetParameter**, **SetFCN** and **ExecuteCommand**. The FCN function can also be given to Minuit2 as an instance of a class implementing the **ROOT::Minuit2::FCNBase** interface. In this case one must use directly the **TFitterMinuit** class via the method **SetMinuitFCN**.

Examples on how to use the Minuit2 and Fumili2 plug-ins are provided in the tutorials' directory `$ROOTSYS/tutorials/fit:minuit2FitBench.C`, `minuit2FitBench2D.C` and `minuit2GausFit.C`. More information on the classes and functions present in Minuit2 is available at [online reference documentation](http://seal.cern.ch/documents/minuit/mnusersguide.pdf). In addition, the C++ MINUIT User Guide provides all the information needed for using directly the package without **TVirtualFitter** interface (see <http://seal.cern.ch/documents/minuit/mnusersguide.pdf>). Useful information on MINUIT and minimization in general is provided in the following documents:

F. James, *Minuit Tutorial on Function Minimization* (<http://seal.cern.ch/documents/minuit/mntutorial.pdf>); F. James, *The Interpretation of Errors in Minuit* (<http://seal.cern.ch/documents/minuit/mnerror.pdf>);

1.9 ROOT Statistics Classes

1.9.1 Classes for Computing Limits and Confidence Levels

TFeldmanCousins class calculates the CL upper/lower limit for a Poisson process using the Feldman-Cousins method (as described in PRD V57 #7, p3873-3889). No treatment is provided in this method for the uncertainties in the signal or the background.

TRolke computes confidence intervals for the rate of a Poisson process in the presence of background and efficiency, using the profile likelihood technique for treating the uncertainties in the efficiency and background estimate. The signal is always assumed to be Poisson; background may be Poisson, Gaussian, or user-supplied; efficiency may be Binomial, Gaussian, or user-supplied. See publication at Nucl. Instrum. Meth. A551:493-503,2005.

TLimit class computes 95% C.L. limits using the Likelihood ratio semi-Bayesian method (method; see e.g. T. Junk, NIM A434, p. 435-443, 1999). It takes signal background and data histograms wrapped in a **TLimitDataSource** as input, and runs a set of Monte Carlo experiments in order to compute the limits. If needed, inputs are fluctuated according to systematic.

1.9.2 Specialized Classes for Fitting

TFractionFitter fits Monte Carlo (MC) fractions to data histogram (a la HMCMLL, R. Barlow and C. Beeston, Comp. Phys. Comm. 77 (1993) 219-228). It takes into account both data and Monte Carlo statistical uncertainties through a likelihood fit using Poisson statistics. However, the template (MC) predictions are also varied within statistics, leading to additional contributions to the overall likelihood. This leads to many more fit parameters (one per bin per template), but the minimization with respect to these additional parameters is done analytically rather than introducing them as formal fit parameters. Some special care needs to be taken in the case of bins with zero content.

TMultiDimFit implements multi-dimensional function parameterization for multi-dimensional data by fitting them to multi-dimensional data using polynomial or Chebyshev or Legendre polynomial

TSpectrum contains advanced spectra processing functions for 1- and 2-dimensional background estimation, smoothing, deconvolution, peak search and fitting, and orthogonal transformations.

RooFit is a complete toolkit for fitting and data analysis modeling (see the RooFit User Guide at <ftp://root.cern.ch/root/doc/RooFit29.pdf>)

TSploit to disentangle signal from background via an extended maximum likelihood fit and with a tool to access the quality and validity of the fit producing distributions for the control variables. (see M. Pivk and F.R. Le Diberder, Nucl. Inst. Meth.A 555, 356-369, 2005).

1.9.3 Multi-variate Analysis Classes

TMultiLayerPerceptron is a Neural Network class (see for more details the chapter “Neural Networks”).

TPrincipal provides the Principal Component Analysis.

TRobustEstimator is a robust method for minimum covariance determinant estimator (MCD).

TMVA is a package for multivariate data analysis (see <http://tmva.sourceforge.net/docu/TMVAUsersGuide.pdf> the User’s Guide).