STAR

STAR Offline Library Long Writeup

STAR C++ Class Library

User Guide and Reference Manual

Revision: 1.15

Date: 1998/08/20 12:36:03

Contents

Part I

User Guide

1 Philosophy and Motivation

Code-reusability is often claimed to be one of the most important benefits to be realized from Object-Oriented programming. This is not really a new concept, especially in High Energy Physics (HEP) where many fortran subroutine libraries, most notably CERNLIB[1] have been used for many years. There are however, several distinct differences between such subroutine libraries and class libraries which Object-Oriented languages allow. First standardized data structures, or containers, along with operations associated with such objects are combined in classes. Second, with subroutine libraries one does not have the opportunity to extend or alter the functionality of a routine unless the source code is available and recompilation is possible. On the other hand, Object-Oriented languages like C++ allows the user to produce a derived class via inheritance which one can add functionality through the addition of member functions or data storage through the addition of data members.

Most C++ compilers come with what is called the "Standard C++ Library" which defines generic container classes (i.e. linked lists, sorted lists, etc.) and simple algorithms associated with these containers. However the needs of the HEP community are much more specialized in terms of the containers we use—things like three- and four-vectors, random number generators, matrices, etc., and such objects are poorly dealt with in expensive commercial class libraries. It was with this motivation that Leif Lönnblad proposed *A Class Library for High Energy Physics* (CLHEP) in C++ at the *Computing in High Energy Physics* (CHEP) conference in 1992. This library provided basic HEP specific classes and although it went a long way in providing a standard for C++ in HEP, C++ was not a mature nor standardized language at the time. Recently (December 1997) an ANSI committee has proposed an international standardization of the C++ language[2] which makes some components of CLHEP redundant and some other parts unnecessarily inflexible (i.e. adding three-vectors with elements of different types).

For these reasons it was felt that the basic component classes of CLHEP (like vectors and matrices) could be rewritten incorporating contemporary new standardized features of the C++ language. These include the use of templates, exception handling, namespaces, use of the Standard Template Library (STL) STL etc. It was also felt that additional functionality specific to the STAR experiment could be incorporated to make this a real STAR Class Library (SCL); things like implementing the STAR track model, data base interfaces etc. Even a simple interface to HBOOK is included. The code in the SCL is written to conform to the ANSI standard and conventions of the Standard C++ Library. The directory and `Makefile` structure is modelled very closely after the pioneering work of the GEANT4 collaboration. Documentation is also seen as an important component of this development. This manual provides a detailed description of each class, its functionality, its dependencies, as well as a description of all user accessible member functions. Example programs and there expected output are also included. More detailed examples testing the functionality of each and every member function are provided in the `examples` directory in the library itself. Web based documentation, giving the user access to the header files and source code is foreseen in the near future.

These developments have been made in consultation with the CERN LHC++ group who maintain the official version of CLHEP and they have expressed interest in perhaps incorporating some of this library in a future release. This is also by no means a complete library and as more and more people get involved with development, it is expected to expand. So look through the manual and feel free to make any comments, good or bad to the collaboration and developers.

[1] http://wwwcn1.cern.ch/asd/index.html
[2] http://www.cygnus.com/misc/wp/nov97/

2 Platforms and Compilers

The StarClassLibrary (SCL) is currently tested on the following platforms and compilers.

1. HP-UX 10.20:
   - aCC A.01.06 or higher versions.
   - g++ 2.8.1.
2. Red Hat Linux 5.0:
   - egcs 1.0.2 (g++).
3. Solaris 2.4–2.6:
   - CC 4.2 and RogueWave's Standard C++ Library.
4. IRIX 6.4:
   - CC 7.1 and Object Space 2.0.2 for IRIX 6.3 modified and compiled with -n32 flag.

Tests with Visual C++ 5.0 fail because of the lack of support of templated member functions and a broken overloading mechanism. Further tests on Irix and AIX are pending.

3 Organization of the SCL

The following is of general interest for software developer only:

All documentation, code, and header files of the SCL is contained in a directory named StarClassLibrary. The SCL is divided in so called *categories*. Each category resides in its own subdirectory which is again divided up in four directories: ./include, ./src, ./examples and - after compilation - a directory which will contain the (shared) library for this category. The name of the latter depends on the referring platform (see section 4) and is stored in the environment variable `STAR_SYS`. Documentation in TEXand HTML, manuals, and README files for the SCL are stored in StarClassLibrary/doc.

To summarize:

StarClassLibrary is the top directory,

- **StarClassLibrary/doc** contains all SCL documentations,
- **StarClassLibrary/*categoryX*** contains all classes of *categoryX*,
  - ***categoryX*/include** contains the different header files (extension .h and .hh),
  - ***categoryX*/src** the referring source code (extension .cc),
  - ***categoryX*/examples** contains small self-describing programs to test and demonstrate most of the features of the classes in this category,
  - ***categoryX*/$STAR_SYS** contains the shared and/or archive library for the platform identified by `$STAR_SYS`.

StarClassLibrary/*categoryY* ...

...

There are three Makefiles in the top directory StarClassLibrary:

GNUmakefile is the general makefile used to compile and install the SCL and the referring examples,

GNUmakefile.common contains the platform independent rules, and

GNUmakefile.architecture defines the macros and compiler flags for each individual supported platform.

Each directory representing a category contains a GNUmakefile which is invoked by the top level StarClassLibrary/GNUmakefile. Each of these makefiles uses the definitions in GNUmakefile.common and GNUmakefile.architecture.

All makefiles are especially written for the GNU make utility (gmake) and rely on certain features only available therein. This is also the reason why we do not follow the standard naming scheme but named them GNUmakefile. Note that these makefiles do not contain any hard-wired file names; the dependencies are rather created on-the-fly.

4 Installing the SCL

The SCL is part of the official STAR software distribution and is therefore present in the actual STAR software releases.
If you want to install the SCL locally on your machine you should continue to read the following three sections which explains how to obtain, compile and install the code.

4.1 Getting the source code

The *StarClassLibrary* is now under **CVS** control at BNL. It can be accessed via afs:

1. Obtain an afs token: `klog -cell rhic`.
2. Make sure `$CVSROOT` is set properly:
   (i.e. `CVSROOT = /afs/rhic/star/packages/repository`)
3. Check-out package into your current working directory:
   `cvs checkout StarClassLibrary`

4.2 Compilation

In order to create the library and (optionally) the referring examples first **set the correct system flag**. The flag is stored in the environment variable `STAR_SYS`. If you are using the official STAR login scripts it should be already defined. Note, however, that the SCL is not yet supported on all STAR platforms. See section 2 for the current platforms that have been tested. To compile:

- set the correct system flag:
  In csh, tcsh:
  `setenv STAR_SYS` *system-flag*.
  In sh, ksh, bash, zsh:
  `export STAR_SYS=`*system-flag*.
  where *system-compiler* can be one of:
  - `i386_linux2`
  - `i386_redhat50`
  - `hp_ux102`
  - `sun4x_54`
  - `sun4x_55`
  - `sun4x_56`
- `cd StarClassLibrary`
- `gmake`
  compiles the shared or archived libraries in each category.
- `gmake examples`
  compiles all the example/test programs in each category.

This creates shared or archived libraries against which your programs can already link. They are located for each category under a directory with the same name as the `STAR_SYS` environment variable. This setup is best suited for developers working with and on the SCL. The disadvantage is that the header files and the libraries are split over several directories which requires slightly more complex makefiles.

Hint: In case your platform is not supported and you encounter problems with your native C++ compiler you can always use the GNU compiler (egcs or gcc depending on your installation) by defining `STAR_SYS=i386_linux2`. Note also that "unsupported platform" does not necessarily mean that the SCL will not compile but rather that it was never tested and that the makefiles contain no parameters specific for this environment. In general it should be sufficient to adjust the settings in GNUmakefile.architecture (see also section 5).

4.3 Installation

In order to install the library in *one* location, (i.e, not distributed over several categories) first define the environment variable `SCL_DEST` which holds the name of the target directory where you want to install the SCL. If not defined the installation target is always StarClassLibrary/scl.

- set the correct target:
  In csh, tcsh:
  `setenv SCL_DEST` *install-dir*.
  In sh, ksh, bash, zsh:
  `export SCL_DEST=`*install-dir*.

- `cd StarClassLibrary`
- `gmake install`

Note that if *install-dir* doesn't exist it is created. The makefile installs three directories: *install-dir*`/$STAR_SYS/lib` which contains the library[3], *install-dir*`/include` which holds all SCL include files and *install-dir*`/doc` for the documentation.

5 Macros

The StarClassLibrary is coded under the assumption that all ANSI features are available. If the used compiler is fully ANSI compliant the SCL will compile without any modifications.
Because the new C++ ANSI standard pushes the limits of current compiler technology, a number of compiler and Standard C++ Library features are often missing or implemented in a way that differs from the ANSI standard.
In order to use the SCL also on those systems various macros were defined which either disable certain features, e.g. exception handling, or use slightly modified (and less elegant) code. The following macros are used throughout the SCL. If the SCL is installed properly they should be defined according to your platform/compiler.

`ST_NO_MEMBER_TEMPLATES:` defined if the compiler does not support template member functions.

`ST_NO_EXCEPTIONS:` defined if the compiler does not support exception handling.

`ST_NO_NUMERIC_LIMITS:` defined if the STL class numeric_limits is not available (it is usually located in the `<limits>` header file).

`ST_NO_TEMPLATE_DEF_ARGS:` defined if the compiler does not support template default arguments

`ST_NO_NAMESPACES:` definded if the compiler does not support multiple namespaces.

`ST_OLD_CLHEP_SYSTEM_OF_UNITS:` user defined if one must use units as defined in CLHEP v1.2 (use of this macro is strongly discouraged).

`NO_HBOOK_INIT:` restricts the automatic initialization of HBOOK memory.

6 Documentation

The documentation of the SCL consists of the *User Guide and Reference Manual* located in StarClassLibrary/doc/tex, i.e. this document, and a HTML class browser in StarClassLibrary/doc/html. The latter is not automatically made during the installation but must be created separately as follows:

- `cd StarClassLibrary`

[3] or libraries if archive *and* shared libraries are supported on your platform

- `gmake class-browser`

or alternatively

- `cd StarClassLibrary/doc/html`
- `gmake`

The class index and all other HTML pages referenced therein are generated automatically from the current code.

Load the file `index.html` in your browser first. The index gives an alphabetical listing of all of the classes, algorithms, and function objects provided by the Star C++ Class Library. The pages for the individual classes begin with a synopsis, which indicates the header file(s) and the signature of the class object followed by the C++ code that describes the public interface. Please note, that the interface is displayed in its pure ANSI form. In a few rare cases and depending on your platform the actual code might differ slightly. This browser only shows the interfaces, i.e., the prototypes of the classes and function objects.

7 Known Problems

- The SCL should compile without any compiler warnings except on Sun platforms due to shortcomings of the native compiler.
- The Sun compiler does not provide the Boolean data type `bool`. Therefore `bool` is implemented as integer type (`int`). As a consequence overloading of (member) functions according to `bool/int` types is not possible. Note that the StPrompt class is affected (see section 10.7).
- In order to run programs built with the shared version of the SCL and standard library on Sun platforms, it is necessary to define an environment variable, `LD_LIBRARY_PATH`, to indicate the directory containing the shared library.

8 Support and Reporting Bugs

Currently the StarClassLibrary is supported by a small group. Hopefully as more people begin to use it and add to it, there will be a larger support base for it. If there is a problem or bug, report it to one or more of the following:

- starsas-l@bnl.gov
- starsofi-l@bnl.gov
- brian.lasiuk@yale.edu
- thomas.ullrich@yale.edu

Part II

Reference Manual

9 Global Constants and Definitions

There is a distinction between the header files which define the STAR data types, system of units, physical constants, simple macros, etc. from those files that actually contain class definitions. This section contains the headers that are necessary to operate in the STAR C++ programming environment.

9.1 Physical Constants

Summary Physical Constants contains the definitions of many important physical constants with the units.

Synopsis `#include "PhysicalConstants.h"`
requires `SystemOfUnits.h` (included by default)

Category global

Description Is now modified from v1.2 of CLHEP to include the new naming convention of the units. All units and constants are still defined to be of type:
`static const HepDouble`.

```
pi          = 3.14159265358979323846
twopi       = 2*pi
halfpi      = pi/2
pi2         = pi*pi

Avogadro    = 6.0221367e+23/mole
c_light     = 2.99792458e+8 * meter/second
c_squared   = c_light * c_light

h_Planck    = 6.6260755e-34 * joule*second
hbar_Planck = h_Planck/twopi
hbarc       = hbar_Planck * c_light
hbarc_squared    = hbarc * hbarc

electron_charge  = - eplus
e_squared        = eplus * eplus

electron_mass_c2 = 0.51099906 * MeV
proton_mass_c2   = 938.27231 * MeV
neutron_mass_c2  = 939.56563 * MeV
amu_c2      = 931.49432 * MeV
amu         = amu_c2/c_squared

mu0         = 4*pi*1.e-7 * henry/meter
epsilon0    = 1./(c_squared*mu0)
```

```
electromagnetic coupling
   = 1.43996e-12 MeV*millimeter/(eplus\^2)
elm_coupling
   = e_squared/(4*pi*epsilon0)
fine_structure_const
   = elm_coupling/hbarc
classic_electr_radius
   = elm_coupling/electron_mass_c2
electron_Compton_length
   = hbarc/electron_mass_c2
Bohr_radius
   =electron_Compton_length/
    fine_structure_const

alpha_rcl2 =
   fine_structure_const*
   classic_electr_radius*
   classic_electr_radius
twopi_mc2_rcl2  =
   twopi*electron_mass_c2*
   classic_electr_radius*
   classic_electr_radius
k_Boltzmann     = 8.617385e-11 * MeV/kelvin
STP_Temperature = 273.15*kelvin
STP_Pressure    = 1.*atmosphere
kGasThreshold   = 1.e-2*gram/centimeter3
```

9.2 StGlobals

Summary StGlobals defines the STAR data types as well as simple macros and templates.

Synopsis `#include "StGlobals.hh"`

Category global

Description Note, that StGlobals.hh includes PhysicalConstants.hh and SystemOfUnits.hh.

```
typedef double         HepDouble
typedef int            HepInt
typedef float          HepFloat
typedef bool           HepBoolean

typedef HepDouble      StDouble
typedef HepFloat       StFloat
typedef HepInt         StInt
typedef HepBoolean     StBool
typedef long           StLong
typedef unsigned short StUshort
typedef unsigned long  StSizeType
```

Global macros

```
#define StNPOS (~(StSizeType)0)
```

Global templates

```
template<class T>
inline StInt sign(T a)
{ return a < 0 ? -1 : 1; }

template<class T>
inline StDouble sqr(T a)
{ return a*a; }
```

Macros for debugging and testing

```
#define PR(x) cout << (#x) << " = " << (x) << endl;
```

9.3 SystemOfUnits

Summary SystemOfUnits defines a set of consistent SI units. All are defined as `static const HepDouble`. The units are backwards compatible with CLHEP v1.2.

Synopsis
```
#define  ST_ADD_OLD_CLHEP_SYSTEM_OF_UNITS // if necessary
#include "SystemOfUnits.h"
```

Category global

Description SystemOfUnits differs from version 1.2 of CLHEP by using the complete name of the unit. In this manner there is no pollution of the global namespace with single character constants like 'm' for meter, 's' for second, etc. The previous units from CLHEP are still available but the user must define a flag `ST_ADD_OLD_CLHEP_SYSTEM_OF_UNITS` before including `SystemOfUnits.h`. One other important difference is that the constants as defined in the header file are contained in a namespace **namespace units**. If your compiler does not support namespaces, the macro `ST_NO_NAMESPACES` must be defined and the constants will reside in the global namespace!

`namespace units {`

BASE UNITS:

Length [L] *(the base unit is millimeter)*

```
millimeter  = 1.
millimeter2 = millimeter*millimeter
millimeter3 = millimeter*millimeter*millimeter

centimeter  = 10.*millimeter
centimeter2 = centimeter*centimeter
centimeter3 = centimeter*centimeter*centimeter

meter       = 1000.*millimeter
meter2      = meter*meter
meter3      = meter*meter*meter

kilometer   = 1000.*meter
kilometer2  = kilometer*kilometer
kilometer3  = kilometer*kilometer*kilometer

micrometer  = 1.e-6*meter
nanometer   = 1.e-9*meter
femtometer  = 1.e-15*meter
fermi       = 1*femtometer

barn        = 1.e-28*meter2
millibarn   = 1.e-3*barn
```

```
microbarn   = 1.e-6*barn
nanobarn    = 1.e-9*barn
```

Angle *(the base unit is radian)*

```
radian      = 1.
milliradian = 1.e-3*radian
degree      = (M_PI/180.0)*radian
steradian   = 1.
```

Time [T] *(the base unit is nanosecond, and Hertz)*

```
nanosecond  = 1.
second      = 1.e+9*nanosecond
millisecond = 1.e-3*second

hertz       = 1./second
kilohertz   = 1.e+3*hertz
Megahertz   = 1.e+6*hertz

Hz          = 1*hertz
kHz         = 1*kilohertz
MHz         = 1*Megahertz
```

Electric charge [Q]

```
eplus       = 1.
e_SI        = 1.60217733e-19
coulomb     = eplus/e_SI
```

DERIVED UNITS:

Energy [E] *(the base unit is MeV)*

```
Megaelectronvolt = 1.
electronvolt     = 1.e-6*Megaelectronvolt
kiloelectronvolt = 1.e-3*Megaelectronvolt
Gigaelectronvolt = 1.e+3*Megaelectronvolt
Teraelectronvolt = 1.e+6*Megaelectronvolt
MeV              = Megaelectronvolt
eV               = electronvolt
keV              = kiloelectronvolt
GeV              = Gigaelectronvolt
TeV              = Teraelectronvolt
joule            = eV/e_SI
```

Mass [E][T^2][L^{-2}]

```
kilogram    = joule*second*second/(meter*meter)
gram        = 1.e-3*kilogram
milligram   = 1.e-3*gram
```

Power [E][T^{-1}]

```
watt        = joule/second
```

Force [E][L^{-1}]

```
newton      = joule/meter
```

Pressure [E][L^{-3}]

```
hep_pascal  = newton/meter2
bar         = 100000*pascal
atmosphere  = 101325*pascal
```

Electric current [Q][T^{-1}]

```
ampere        = coulomb/second
```

Electric potential [E][Q^{-1}]

```
Megavolt      = MeV/eplus
kilovolt      = 1.e-3*Megavolt
volt          = 1.e-6*Megavolt
```

Electric resistance [E][T][Q^{-2}]

```
ohm           = volt/ampere
```

Electric capacitance [Q^{2}][E^{-1}]

```
farad         = coulomb/volt
millifarad    = 1.e-3*farad
microfarad    = 1.e-6*farad
nanofarad     = 1.e-9*farad
picofarad     = 1.e-12*farad
```

Magnetic Flux [T][E][Q^{-1}]

```
weber         = volt*second
```

Magnetic Field [T][E][Q^{-1}][L^{-2}]

```
tesla      = volt*second/meter2
gauss      = 1.e-4*tesla
kilogauss  = 1.e-1*tesla
```

Inductance [T^{2}][E][Q^{-2}]

```
henry      = weber/ampere
```

Temperature

```
kelvin     = 1.
```

Amount of substance

```
mole       = 1.
```

Activity [T^{-1}]

```
becquerel  = 1./second
curie      = 3.7e+10 * becquerel
```

Absorbed dose [L^2][T^{-2}]

```
gray       = joule/kilogram
```

Miscellaneous

```
perCent     = 0.01
perThousand = 0.001
perMillion  = 0.000001
```

As defined in CLHEP

```
#ifdef ST_ADD_OLD_CLHEP_SYSTEM_OF_UNITS
```

BASE UNITS:

Length [L] *(the base unit is millimeter)*

```
mm  = 1.
mm2 = mm*mm
mm3 = mm*mm*mm

cm  = 10.*mm
cm2 = cm*cm
cm3 = cm*cm*cm

m   = 1000.*mm
m2  = m*m
m3  = m*m*m

km  = 1000.*m
km2 = km*km
km3 = km*km*km

microm = 1.e-6*m
nanom  = 1.e-9*m
```

Angle *(the base unit is radian)*

```
rad     = 1.
mrad    = 1.e-3*rad
deg     = (M_PI/180.0)*rad
st      = 1. // (steradian)
```

Time [T] *(the base unit is nanosecond, and Hertz)*

```
ns      = 1.
s       = 1.e+9*ns
```

```
ms      = 1.e-3*s
```

Mass [E][T^2][L^{-2}]

```
kg      = joule*s*s/(m*m)
g       = 1.e-3*kg
mg      = 1.e-3*g
#endif
```

Examples

Program Code:

```
#define ST_ADD_OLD_CLHEP_SYSTEM_OF_UNITS
#include "SystemOfUnits.h"

using namespace units;
int main()
{
    cout << "This program illustrates the use of SystemOfUnits
             and PhysicalContants" << endl;
    cout << "------------------------------------------------
             -------------------" << endl;

    cout << "Base unit is millimeter" << endl;
    cout << "1 millimeter = " << (1*millimeter) << endl;
    cout << "1 meter =      " << (1*meter)      << endl;
    cout << "1 centimeter = " << (1*centimeter) << endl;
    cout << "1 fermi =      " << (1*fermi)      << endl;

    cout << "1 barn =       " << (1*barn)       << endl;

    cout << "1 degree =     " << (1*degree)     << endl;

    cout << "1 second =     " << (1*second)     << endl;
    cout << "1 nanosecond = " << (1*nanosecond) << endl;

    cout << "1 kHz =        " << (1*kHz)     << endl;

    cout << "1 newton =    " << (1*newton)     << endl;
    cout << "1 joule =     " << (1*joule)     << endl;

    cout << "1 GeV =        " << (1*GeV)     << endl;
    cout << "1 Gigaelectronvolt = " << (1*Gigaelectronvolt)     << endl;

#ifdef ST_ADD_OLD_CLHEP_SYSTEM_OF_UNITS
    cout << "The old CLHEP definitions are also supported:" << endl;
    cout << "--------------------------------------------" << endl;
    cout << "1 mm = " << (1*mm) << endl;
    cout << "1 m  = " << (1*m)  << endl;
    cout << "1 cm = " << (1*cm) << endl;

    cout << "1 s  = " << (1*s)  << endl;
    cout << "1 ns = " << (1*ns) << endl;
#else
    cout << "\nTo use the old CLHEP definitions the flag:" << endl;
    cout << "    ST_ADD_OLD_CLHEP_SYSTEM_OF_UNITS" << endl;
```

```
    cout << "must be defined" << endl;
    cout << "**This is an obselete file and should NOT be used**" << endl;
#endif

    return 0;
}
```

Program Output:

```
This program illustrates the use of SystemOfUnits and PhysicalContants
--------------------------------------------------------------------
Base unit is millimeter
1 millimeter = 1
1 meter =      1000
1 centimeter = 10
1 fermi =      1e-12
1 barn =       1e-22
1 degree =     0.0174533
1 second =     1e+09
1 nanosecond = 1
1 kHz =        1e-06
1 newton =    6.24151e+09
1 joule =     6.24151e+12
1 GeV =        1000
1 Gigaelectronvolt = 1000
The old CLHEP definitions are also supported:
---------------------------------------------
1 mm = 1
1 m  = 1000
1 cm = 10
1 s  = 1e+09
1 ns = 1
```

10 Class Reference

The classes which are currently implemented and available from the BNL CVS repository are described in alphabetic order.

10.1 StGetConfigValue

Summary StGetConfigValue is a set of templated functions which read resource values from a given configuration file.

Synopsis `#include "StGetConfigValue.hh"`

Category utilities

Description StGetConfigValue() implements a simple parser which reads resource values associated with a given name from a configuration file. There are three versions of StGetConfigValue(): one for reading a scalar resource, one for a multi-value resources (e.g arrays), and one for STL containers.

These functions are useful when many parameters have to be read from one or several ascii resource files. They can not replace a real database but are useful for development purposes and small programs.

StGetConfigValue() ignores every character after the shell-like comment character "#" or the C++-like comment characters "//". They can be used for inline comments as well as comments which spawn a whole line.

comment
// *comment*

In the configuration file the name of the variable has to be separated from its value by a colon ":".

`name: value`

Syntax

```
template<class T>
void StGetConfigValue(const char* fname,
                      const char* name, T& val);
```

Searches the file `fname` for `name` and stores the referring value in `val`. If the name is not found the value of `val` is not altered. Note that this version can also be used to read in `StThreeVector` objects since the `operator>>()` is properly defined in the context of this class.

```
template<class T>
void StGetConfigValue(const char* fname,
                      const char* name,
                      T& val, int N);
```

Searches the file `fname` for `name` and stores the referring `N` values in the *(i)* container object `val`. Note that `T` can be an ordinary C array (e.g. `float*` or a STL container (e.g. `vector`). If the name is not found the content of `val` is not altered.

Examples **Program Code:**

```
#include "StGlobals.hh"
#include "StGetConfigValue.hh"
#include "StThreeVector.hh"
#include <vector>
```

```
int main()
{
    const char* filename = "example.conf";

    double singleValue = 10;
    StGetConfigValue(filename, "singleValue", singleValue);
    cout << "singleValue = " << singleValue << endl;

    float *manyValues = new float[10];
    StGetConfigValue(filename, "manyValues", manyValues, 10);
    cout << "manyValues = ";
    for (int i=0; i<10; i++) cout << manyValues[i] << ' ';
    cout << endl;

    vector<double> vec(10);
    StGetConfigValue(filename, "vec", vec, 5);
    cout << "vec = ";
    for (int k=0; k<10; k++) cout << vec[k] << ' ';
    cout << endl;

    StThreeVector<double> vec3;
    StGetConfigValue(filename, "vec3", vec3);
    cout << "vec3 = " << vec3 << endl;

    string anyName;
    StGetConfigValue(filename, "anyName", anyName);
    cout << "anyName = " << anyName << endl;

    float xfoo = 3.14;
    StGetConfigValue(filename, "xfoo", xfoo);
    cout << "xfoo = " << xfoo << endl;

    return 0;
}
```

Configuration file:

```
#
# This is a comment at the beginning
#
singleValue:                    100.                    // comment
manyValues:                     0 1 2 3 4 5 6 7 8 9     #  comment
vec:                            10.1 10.2 30.3 0 42
//  This is an comment
#   and this is an comment
vec3:                           1.05 1.05 2.5
anyName:                        Aladin
```

Programs Output:

```
singleValue = 100
manyValues = 0 1 2 3 4 5 6 7 8 9
vec = 1.05 1.05 2.5 0 42 0 0 0 0 0
vec3 = (1.05, 1.05, 2.5)
anyName = Aladin
xfoo = 3.14
```

10.2 StHbook

Summary StHbook is an HBOOK[4] wrapper which allows creation of 1 and 2 dimensional histograms and ntuples.

Synopsis `#include "StHbook.hh"`
`NOTE:` The user must link with `packlib`.[5] This is set up in the `GNUMakefile.architecture` file according to the **CERN_ROOT** environmental variable set in the STAR login scripts.

Category histograms

Description This class defines a wrapper that allows the user to fill and create one and two dimensional histograms as well as ntuples. It defines its own function prototypes in C, so there is no need for the use of `cfortran.h`, however the number of operations is more limited. Should the user require more functionality, such prototypes must be added. All initialization of HBOOK and ZEBRA is performed within the classes. No additional compile flags are needed. The user must ensure that his/her code be linked with `packlib` only. If the pawc common is too small it is possible to increase the value of the `constant 'SizeOfPawCommon'` to whatever is required and recompile.

The initialization of HBOOK may be prevented by defining the macro `NO_HBOOK_INIT` in case it is initialized in other parts of the program.

Persistence None.

Related Classes None.

Public Constructors

Histogram File

```
StHbookFile(const char* name,
       int rl = 1024, int lun = 10);
```

Create an HBOOK output file with a name *name* a default record size *rl* of 1024, and a logical unit number *lun* of 10. Note that the record length can take values between 1024 and 8192 only.

1-dimensional Histograms

```
StHbookHisto(const char* name,
     int nbin, float x1, float x2);
```

Constructs a histogram with title *name* and *nbin* bins over a range $x1 < x < x2$.

```
StHbookHisto(int id, const char* name, int nbins,
                         float x1, float x2);
```

Constructs a histogram with id number *id*, title *name* and *nbin* bins over a range $x1 < x < x2$.

```
StHbookHisto(const StHbookHisto&);
```

Copy constructor.

[4] http://wwwcn1.cern.ch/asd/index.html
[5] The process is somewhat more complicated for SUN.

<u>2-dimensional Histograms</u>

```
StHbookHisto2(const char* name, int xbins,
      float x1, float x2, int ybins, float y1, float y2);
```

Constructs a two-dimensional histogram with a label *name*, *xbins* in the x direction bounded by $x1 < x < x2$ and *ybins* in the y direction bounded by $y1 < y < y2$.

```
StHbookHisto2(int id, const char* name,
              int xbins, float x1, float x2
              int ybins, float y1, float y2);
```

Constructs a two-dimensional histogram with an id number *id*, a label *name*, *xbins* in the x direction bounded by $x1 < x < x2$ and *ybins* in the y direction bounded by $y1 < y < y2$.

```
StHbookHisto2(const StHbookHisto2&);
```

Copy constructor;

<u>Row Wise N-tuples</u>

```
StHbookTuple(int id, const char* name, int ntag);
```

Constructs an ntuple with an id number *id*, a name *name* and *ntag* rows.

```
StHbookTuple(const char* name, int ntag);
```

Constructs an ntuple with a default id number, a name *name* and *ntag* rows.

Public Member Operators

<u>1-dimensional Histograms</u>

```
StHbookHisto& operator= (const StHbookHisto&);
```

Assignment operator.

<u>2-dimensional Histograms</u>

```
StHbookHisto2& operator= (const StHbookHisto2&);
```

Assignment operator.

<u>Row Wise N-tuples</u>

```
operator<<
```

Assignment of tags to the ntuple.

Public Member Functions

<u>Hbook Files</u>

```
int isGood() const;
```

Checks the status of the file. Checks the return code from *hropen* and the ZEBRA error flag. Returns 1 if okay.

```
void StHbookFile::list(const char* opt = 0);
```

Lists the contents of an HBOOK directory with *opt* being the possible options. As a default, only the histograms are listed. HBOOK equivalent is *hldir*.

```
void StHbookFile::saveAndClose();
```

Saves the Histograms and N-tuples in the HBOOK file and closes the file. Combines the calls of *hrout, hrend, hcdir*.

<u>1-dimensional histograms</u>

```
void fill(float x, float weight = 1);
```

Adds an entry *x* to a histogram with a default weight of 1. HBOOK equivalent is: *hfill*.

`fastFill(float x, float weight = 1);`
Adds an entry *x* to a histogram with a default weight of 1. HBOOK equivalent is: *hf1*.

`int id();`
Returns the id number of the histogram.

`int entries();`
Returns the number of entries in a histogram. HBOOK equivalent is *hnoent*.

`float max()`
Returns the maximum channel content of a histogram. HBOOK equivalent is *hmax*.

`float min()`
Returns the minimum channel content of a histogram. HBOOK equivalent is *hmin*.

`float sum();`
Returns the integrated contents of a histogram. HBOOK equivalent is *hsum*.

`float mean();`
Returns the mean of the distribution in the histogram. HBOOK equivalent is *hstati*.

`float sigma();`
Returns the standard deviation of the distribution in the histogram. HBOOK equivalent is *hstati*.

`void setOpt(const char*);`
Select an option for the histogram. HBOOK equivalent is *hidopt*.

`void print();`
Prints an ascii representation of the histogram to standard out. HBOOK equivalent is *hprint*.

`void reset();`
Zeros the bin contents of a histogram. HBOOK equivalent is *hreset*.

<u>`2-dimensional Histograms`</u>
`void  fill(float x, float y, float weight = 1);`
Adds an entry *x,y* to a histogram with a default weight of 1. HBOOK equivalent is: *hfill*.

`void  fastFill(float x, float y, float weight = 1);`
Adds an entry *x,y* to a histogram with a default weight of 1. HBOOK equivalent is: *hf2*.

`int id();`
Returns the id number of the histogram.

`int entries();`
Returns the number of entries in a histogram. HBOOK equivalent is *hnoent*.

`float max()`
Returns the maximum channel content of a histogram. HBOOK equivalent is *hmax*.

`float min()`
Returns the minimum channel content of a histogram. HBOOK equivalent is *hmin*.

`float sum();`
Returns the integrated contents of a histogram. HBOOK equivalent is *hsum*.

`float mean();`
Returns the mean of the distribution in the histogram. HBOOK equivalent is *hstati*.

`float sigma();`
Returns the standard deviation of the distribution in the histogram. HBOOK equivalent is *hstati*.

`void setOpt(const char*);`
Select an option for the histogram. HBOOK equivalent is *hidopt*.

`void print();`
Prints an ascii representation of the histogram to standard out. HBOOK equivalent is *hprint*.

`void reset();`
Zeros the bin contents of a histogram. HBOOK equivalent is *hreset*.

`Row Wise N-Tuples`

`void setTag(const char* tag);`
Labels the entries in a row with a name *tag*.

`void book();`
Allocates space for the n-tuple. HBOOK equivalent is *hbookn*.

`int id();`
Returns the id number of the ntuple.

`int length();`
Returns the number of tags in the ntuple.

`int entries();`
Returns the number of rows the ntuple contains. HBOOK equivalent is *hnoent*

`void fill(float *vec);`
Fills a row in an ntuple. HBOOK equivalent is *hfn*.

`StBool getEvent(int number, float *vec);`
Returns the entries from the row *number* in an array *vec*. Returns 1 if no error code is returned. HBOOK equivalent is *hgnf*.

Examples

```
#include <unistd.h>
#include <iostream.h>
#include <string>

#include "Random.h"
#include "JamesRandom.h"
#include "RandGauss.h"

#include "StHbook.hh"

int main()
{
    // Define name of output file
    string filename("hbook.ntp");
```

```
cout << "HBOOK file is: " << filename << endl;

// Define the HBOOK file
StHbookFile hbookFile(filename.c_str());

// The N-tuple
const int tupleSize1 = 2;
StHbookTuple theTuple("random",tupleSize1);

// label the rows
theTuple.setTag("index").setTag("random").book();
// or alternatively
//theTuple << "index" << "random" << book;

// the Histogram
StHbookHisto theHisto("gauss distribution", 100, -10, 10);

float tuple[tupleSize1];  // array to be filled

HepJamesRandom engine;
RandGauss gaussDistribution(engine);

for(int ii=0; ii<100; ii++) {
    int k=0;
    tuple[k++] = static_cast<float>(ii);
    double randomNumber = gaussDistribution.shoot();
    tuple[k++] = static_cast<float>(randomNumber);

    theTuple.fill(tuple);

    theHisto.fill(static_cast<float>(randomNumber),1);
}

cout << "\nAvailable information regarding the histogram" << endl;
cout << "---------------------------------------------" << endl;
cout << "Histogram ID:      " << theHisto.id()      << endl;
cout << "Histogram entries: " << theHisto.entries() << endl;
cout << "Histogram max:     " << theHisto.max()     << endl;
cout << "Histogram min:     " << theHisto.min()     << endl;
cout << "Histogram sum:     " << theHisto.sum()     << endl;
cout << "Histogram mean:    " << theHisto.mean()    << endl;
cout << "Histogram sigma:   " << theHisto.sigma()   << endl;

// print ASCII representation of the Histogram to std out
//theHisto.print();

theHisto.setOpt("show");

cout << "\nAvailable information regarding the N-tuple" << endl;
cout << "-------------------------------------------" << endl;
cout << "N-tuple ID:      " << theTuple.id()      << endl;
cout << "N-tuple entries: " << theTuple.entries() << endl;
cout << "N-tuple length:  " << theTuple.length()  << endl;

cout << "\nFile information" << endl;
cout << "hbookFile status: (1) is good " << hbookFile.isGood() << endl;
hbookFile.list();
```

```
    hbookFile.saveAndClose();

    return 0;
}
```

Programs Output:

```
HBOOK file is: hbook.ntp

Available information regarding the histogram
---------------------------------------------
Histogram ID:      2
Histogram entries: 100
Histogram max:     14
Histogram min:     0
Histogram sum:     100
Histogram mean:    -0.114711
Histogram sigma:   0.907482

Available information regarding the N-tuple
-------------------------------------------
N-tuple ID:      1
N-tuple entries: 100
N-tuple length:  2

File information
hbookFile status: (1) is good 1


 ===> Directory : //HISTOS
```

10.3 StHelix

Summary StHelix is a parameterization of a helix which is a mathematical representation of the trajectory of a charged particle in a uniform magnetic field.

Synopsis `#include "StHelix.hh"`

Category tracks

Description This class defines a parameterized helix in space which can be used to represent the trajectory of a charged particle in a homogeneous magnetic field. The parameterization is taken from Bock[6]. It is also the model that is used in the E896 experiment at BNL and the model used in the current STAR tracking code. In the STAR coordinate system it specifies a helix by 5 independent parameters:

- Curvature – the inverse of the radius of the circle in the x-y plane.
- Dip Angle – the inclination angle of the helix in the y-z plane.
- Phase – the azimuth in the xy plane measured from the ring center.
- Origin – starting point of the helix.
- Orientation – gives the sense of whether the helix rotates clockwise or counter clockwise.

A more detailed description of the underlying parametrization and several formulas used for the implementation can be found in appendix A.

A special case of the helix occurs when the curvature is set to zero. This represents a straight line and so the StHelix class can be used to represent tracks in the absence of a magnetic field as well. The origin of the helix is specified by an `StThreeVector` and may be set individually as may all other components of the parameterization. The interface provides access functions for all parameters that define the helix.

No equivalent class exists in CLHEP.

Persistence None

Related Classes Class **StPhysicalHelix** is derived from StHelix which defines a helix by specifying the 3 momentum, the charge, and the origin of a particle.

Public Constructors

```
StHelix(double c, double dip, double phase,
        StThreeVector<double> o, int h=-1);
```

Constructs a helix with curvature=c, dip angle=dip, phase=phase, origin=o, and charge=-h.

```
StHelix(const StHelix& hlx)
```

Copy Constructor. Constructs a helix with the content of `hlx`.

[6] R. K. Bock et al., *Data Analysis Techniques for High-Energy Physics Experiments*, : ed M. Regler, [Cambridge University Press: 1990].

Public Member Operators

None

Public Member Functions

It is possible to set and access the data members within the class via the names associated with a space-time vector:

```
double dipAngle() const;
```
Returns the dip angle of the helix.

```
double curvature() const;
```
Returns the curvature (1/R) of the helix in the xy plane.

```
double phase() const;
```
Returns the phase of the helix.

```
double xcenter() const;
```
Returns the x-coordinate of the center of the circle.

```
double ycenter() const;
```
Returns the y-coordinate of the center of the circle.

```
double h() const;
```
Returns the orientation of the helix. In StPhysicalHelix it has the meaning: -sign(q*B) where q is the charge of the particle and B is the sign of the magnetic field. It specifies whether the helix rotates clockwise or counter-clockwise.

```
StThreeVector<double>& origin() const;
```
Returns the origin or starting point of the helix.

```
void setParameters(double c, double dip, double phase,
                   StThreeVector<double> o, int h);
```
Sets parameters in a unique an order dependent way.

```
double x(double s) const;
```
Returns the x-coordinate of the helix at a pathlength *s*.

```
double y(double s) const;
```
Returns the y-coordinate of the helix at a pathlength *s*.

```
double z(double s) const;
```
Returns the z-coordinate of the helix at a pathlength *s*.

```
StThreeVector<double> at(double s) const;
```
Returns the position in three space of the helix at a pathlength *s*.

```
double period() const;
```
Returns the period length of the helix.

```
pair<double, double> pathLength(double r) const;
```
Returns the path length of the helix given a radial distance (specified in cylindrical coordinates). A pair is returned because the function is double valued within a single period. The smallest path length (sometimes negative) is returned as the first number of the pair.

```
double pathLength(const StThreeVector<double>& p) const;
```
Given a position *s* in space, returns the path length of the helix at the distance of closest approach to the helix.

`double pathLength(const StThreeVector<double>& r,`
`                  const StThreeVector<double>& n) const;`
Returns the path length at which the helix intersects with a given plane. The plane is defined by two vectors: *r* and *n*. *r* defines the position of one (arbitrary) point on the plane and *n* is the referring normal vector. This method works for straight tracks and helices. In case no solution exist, i.e., the helix does not intersect with the plane, the largest possible numeric value is returned. This number can be obtained by: `numeric_limits<double>::max()`.

`double distance(const StThreeVector<double>& p) const;`
Returns the minimal distance between a point *s* and self. Uses `pathLength()` to obtain *s* at the distance of closest approach and returns `abs(at(s)-p);`

`bool valid() const;`
checks for a valid parameterization. Currently implies that:

- the dip angle cannot be $\pi/2$.
- h = ± 1 only!
- the curvature ≥ 0.

`void moveOrigin(double s);`
Moves the origin along the helix to s which becomes the s=0 point. This redefines the phase in a self-consistent way.

Global Operators

`int operator== (const StHelix& v1, const StHelix& v2);`
Returns 1 if the parameters of the two helices are identical.

`int operator!= (const StHelix& v1, const StHelix& v2);`
Returns 1 if any of the parameters of the two helices are not identical.

`ostream& operator<<(ostream& os, const StHelix& h);`
Prints the parameters of the helix `h` to output stream `os`.

Examples

```
#include "StHelix.hh"
#include "SystemOfUnits.h"

int main()
{
    double radius    = 2;
    double dipAngle  = 10;
    double phase     = 10;
    double x0 = 0;
    double y0 = 0;
    double z0 = 0;
    int    H = -1;
    pair<double, double> s;

    StHelix  *helix = 0;
    double   r = 0.1;

    double   slow, sup, ds, ss;
    StThreeVector<double> origin, point, mmpoint;

    delete helix;
```

```
    helix = new StHelix(1/(radius*meter),
                        dipAngle*degree,
                        phase*degree,
                        origin*millimeter,
                        H);

    if (!helix->valid()) {
        cerr << "Error: parametrization is not valid" << endl;
    }
    else {
        cout << "The helix parameter are:" << endl;
        cout << *helix << endl;
        cout << "The period of the helix is: " << helix->period() << endl;
    }

    ds=100*centimeter;
    cout << "ds = " << ds << " -> " << helix->at(ds) << endl;

    r=1*m;
    s = helix->pathLength(r);
    cout << "The helix reaches r=1m at s1 = " << s.first
         << " and s2 = " << s.second << endl;

    mmpoint = StThreeVector<StDouble>(100, 100, 100);
    ss = helix->pathLength(mmpoint*millimeter);
    cout << "The helix reaches r at s = " << ss << endl;
    cout << "Crosscheck point = " << helix->at(ss)
         << ", delta = " << abs(mmpoint-helix->at(ss)) << endl;

    return 0;
}
```

Programs Output:

```
The helix parameter are:
(curvature = 0.0005, dip angle = 0.174533, phase = 0.174533,
h = -1, origin = (0, 0, 0))

The period of the helix is: 12760.2

ds = 1000 -> (-69.8095, -972.386, 173.648)
The helix reaches r=1m at s1 = -1026.31 and s2 = 1026.31
The helix reaches r at s = -59.1696
Crosscheck point = (-10.9531, 57.2299, -10.2747), delta = 162.174
```

10.4 StLorentzVector

Summary StLorentzVector is a templated general 4-vector class defining vectors in four-space.

Synopsis

```
#include "StLorentzVector.hh"
template<class T> StLorentzVector;
```

Category vectors+matrices

Description This class defines a general 4-vector which can be used to represent space-time points, 4-momenta, etc. It has most of the functionality of `StThreeVector` for the spatial component of the vector and adds further member functions which are specific to a 4-vector. Its interface is modelled very closely to that of `LorentzVector` in CLHEP with two significant differences: (i) it is templated vector and (ii) it is derived via (protected) inheritance from `StThreeVector`. It offers essentially the same functionality as the CLHEP version but like `StThreeVector` is more flexible in terms of precision and storage optimisation, i.e. in order to minimize for memory and storage volume a `StLorentzVector`'s with type argument `float` can be used but easily transformed into a double precision version for computation when higher accuracy is needed. In addition to the CLHEP version there are a few member functions added which are useful in the context of Heavy-Ion Physics such as `rapidity()`, and `mt()`.

The template argument is used to define the type associated with the x (p_x), y (p_y), z (p_z), and t (E) components. This argument must be one of the floating point number data types available in the C++ language, either `float`, `double`, or `long double`. The default type is `double`. These are also specified as: `StFloat`, `StDouble`, or `StLongDouble` in the STAR CLASS LIBRARY.

Please note that `StLorentzVector` is *not* virtual. This is a compromise in order to minimize the storage size, i.e. to avoid the additional ballast of the virtual table pointer.

Persistence None

Related Classes Class StLorentzVector is derived from **StThreeVector** defining a 4-dimensional Lorentz vector. Although StLorentzVector contains the member functions and operators contained in **StThreeVector**, they are included to be complete.

Public Constructors

`StLorentzVector<T>();`
Constructs a 4-vector with all components initialized to 0.

`StLorentzVector<T>(T x, T y, T z, T t);`
Constructs a 4-vector with given components x (p_x), y (p_y), z (p_z), and t (E).

`StLorentzVector<T>(StThreeVector<T> v, T t);`
Constructs a 4-vector with spatial components given by `StThreeVector` v and temporal component t.

`StLorentzVector<T>(T t, StThreeVector<T> v);`
Constructs a 4-vector with spatial components given by `StThreeVector` v and

temporal component t. Same as above but with the spatial component as 1^{st} argument. Note that this is not available in CLHEP.

```
template<class X>
StLorentzVector<T>(const StLorentzVector<X> &vec)
```
Copy Constructor. Constructs a 4-vector with the content of vec. Note that vec can be an object with different template arguments then self, i.e. one can instantiate a vector of type double with an vector of type float and vice versa.

Public Member Operators

```
template<class X>
StLorentzVector<T>
operator= (const StLorentzVector<X> &vec);
```
Assignment operator. Replaces the content of self with the content of vec. Note that vec can be an object with different template arguments then self, i.e. one can assign a vector of type double to a vector of type float and vice versa.

```
T operator() (size_t i) const;
```
Returns components by index. Note that the first index (the x-component) has index 0 while the time component has index 3. The result for indices > 3 is platform dependent. If the compiler supports exception handling an out_of_range exception is thrown.

```
T operator[] (size_t i) const;
```
Same as operator() above.

```
StLorentzVector<T> operator- ();
```
Unary minus. Returns copy of self with all components negated.

```
StLorentzVector<T> operator+ ();
```
Unary plus. Returns copy of self.

```
StLorentzVector<T>&
operator*= (double c);
```
Returns self multiplied by scalar c.

```
StLorentzVector<T>&
operator/= (double c);
```
Returns self divided by scalar c.

```
template<class X>
bool
operator== (const StLorentzVector<X>& vec);
```
Equality check. Returns true if self equals vec else false.

```
template<class X>
bool
operator!= (const StLorentzVector<X>& vec);
```
Inequality check. Returns true if self is not equal to vec else false.

```
template<class X>
StLorentzVector<T>&
operator+= (const StLorentzVector<X>& vec);
```
Adds vec to self and returns self.

```
template<class X>
StLorentzVector<T>&
operator-= (const StLorentzVector<X>& vec);
```
Subtracts `vec` from self and returns self.

Public Member Functions

It is possible to set and access the data members within the class via the names associated with a space-time vector:

```
void setX(T x);
```
Set the x-component in Cartesian coordinate system.

```
void setY(T y);
```
Set the y-component in Cartesian coordinate system.

```
void setZ(T z);
```
Set the z-component in Cartesian coordinate system.

```
void setT(T t);
```
Set the t-component.

```
T x() const;
```
Returns the x-component in Cartesian coordinate system.

```
T y() const;
```
Returns the y-component in Cartesian coordinate system.

```
T z() const;
```
Returns the z-component in Cartesian coordinate system.

```
T t() const;
```
Returns the temporal component.

It is also possible to access and set the data members via the momentum space names:

```
void setPx(T px);
```
Set the p_x component.

```
void setPy(T py);
```
Set the p_y component in momentum space.

```
void setPz(T pz);
```
Set the p_z component in momentum space.

```
void setE(T e);
```
Set the energy component in momentum space.

```
void setVect(StThreeVector<T> vec);
```
Set the spatial component by specifying the 3-vector.

```
T plus() const;
```
Returns the positive light-cone component given by $t + z$.

```
T minus() const;
```
Returns the negative light-cone component given by $t - z$.

```
T px() const;
```
Returns the p_x component in momentum space.

`T py() const;`
Returns the p_y component in momentum space.

`T pz() const;`
Returns the p_z component in momentum space.

`T e() const;`
Returns the energy component.

`const StThreeVector<T>& vect() const;`
Returns a constant reference to the 3-vector component.

`T m() const;`
Returns the 4-vector invariant mass defined by:

$$m = \sqrt{E^2 - \vec{p}^2}$$

Should the condition $E^2 < p^2$ be met, the function returns:

$$m = -\sqrt{-(E^2 - \vec{p}^2)}$$

`T m2() const;`
Returns the 4-vector invariant squared (above).

`T rapidity() const;`
Returns the rapidity defined by:

$$y = \frac{1}{2}\ln(\frac{E + p_z}{E - p_z})$$

`T mt() const;`
Returns the transverse mass defined by:

$$m_T = \sqrt{p_T^2 + m^2}$$

Note that this member function is not available from CLHEP.

`T mt2() const;`
Returns the square of the transverse mass (see above). Note that this member function is not available from CLHEP.

`T phi() const;`
Returns the azimuth angle of the 3-vector component.

`T theta() const;`
Returns the polar angle of the 3-vector component.

`T cosTheta() const;`
Returns the cosine of the polar angle of the 3-vector component.

`T perp2() const;`
Returns the transverse component squared of the 3-vector component. (R^2 in cylindrical coordinate system).

```
T perp() const;
```
Returns the transverse component (R in cylindrical coordinate system).

```
T pseudoRapidity() const;
```
Returns the pseudo-rapidity, i.e. $-\ln(\tan\theta/2)$ of the vector. Note that this value is only valid under the assumptions that the vector origins from the center of the referring reference frame. Be also aware that this member function is *not* present in the CLHEP LorentzVector class.

Global Functions

```
template<class T>
T abs(const StLorentzVector<T>& vec);
```
Returns the 4-vector invariant mass defined by:

$$|\vec{v}| = \sqrt{E^2 - \vec{p}^2}$$

Should the condition $\mathrm{E}^2 < \mathrm{p}^2$ be met, the function returns:

$$|\vec{v}| = -\sqrt{-(E^2 - \vec{p}^2)}$$

Same as `vec->m()`. Be also aware that this feature is *not* provided by CLHEP.

Global Operators

```
template<class T, class X>
StLorentzVector<T>
operator+ (const StLorentzVector<T>& v1,
           const StLorentzVector<X>& v2);
```
Returns the sum of `v1` and `v2`. The type of the returned vector is determined by the type argument of the first vector `v1`.

```
template<class T, class X>
StLorentzVector<T>
operator- (const StLorentzVector<T>& v1,
           const StLorentzVector<X>& v2);
```
Returns the `v1` minus `v2`. The type of the returned vector is determined by the type argument of the first vector `v1`.

```
template<class T, class X>
T operator* (const StLorentzVector<T>& v1,
             const StLorentzVector<X>& v2);
```
Returns the scalar product of `v1` and `v2`. The type of the returned value is determined by the type argument of the first vector `v1`.

```
template<class T>
StLorentzVector<T>
operator* (const StLorentzVector<T>& vec,
           double c);
```
Returns vector `vec` multiplied by scalar c.

```
template<class T>
StLorentzVector<T>
operator* (double c,
```

```
             const StLorentzVector<T>& vec);
```

Returns vector `vec` multiplied by scalar c.

```
template<class T, class X>
StLorentzVector<T>
operator/ (const StLorentzVector<T>& vec,
             X c);
```

Returns vector `vec` divided by scalar c.

```
template<class T>
ostream&
operator<< (ostream& os,
              const StLorentzVector<T>& vec);
```

Prints vector `vec` to output stream `os`.

```
template<class T>
istream&
operator>> (istream& is,
              StLorentzVector<T>& vec);
```

Reads vector `vec` from input stream `is`. Be also aware that this operator is *not* provided by the CLHEP ThreeVector class.

Examples

```
#include "StLorentzVector.hh"

int main()
{
  StLorentzVector<double> a;
  StLorentzVector<double> b(1,2,3,4);
  StLorentzVector<float>  c(b);
  StThreeVector<float>    q(M_PI,M_E,M_LN2);
  StThreeVector<float>    r;

  cout << "b = " << b << endl;
  cout << "c = " << c << endl;

  // add two 4-vectors
  a=b+c;
  cout << "b + c = " << a << endl;

  // modify components of vector
  c.setX(4);
  c.setPz(1);

  cout << "c = " << c << endl;

  // 3 Vectors
  r = b.vect();
  cout << "q = " << q << endl;
  cout << "q.perp() = " << (q.perp()) << endl;

  StFloat k = M_PI;
  StLorentzVector<StDouble>(b.vect(),k);
  cout << "k = " << k << endl;

  // scalar product
  double d = a*c;
```

```
  cout << "a * c = " << d << endl;

  // mass
  cout << "c.m() = " << c.m() << endl;

  // rapidity
  cout << "c.rapidity() = " << c.rapidity() << endl;

  return 0;
}
```

Programs Output:

```
b = ((1, 2, 3),4)
c = ((1, 2, 3),4)
b + c = ((2, 4, 6),8)
c = ((4, 2, 1),4)
q = (3.14159, 2.71828, 0.693147)
q.perp() = 4.15435
k = 3.14159
a * c = 10
c.m() = -2.23607
c.rapidity() = -0.255413
```

10.5 StMatrix

Summary StMatrix is a templated class defining matrices and associated operations.

Synopsis

```
// for bound checking on access operator
#define MATRIX_BOUND_CHECK
// compatibility with StThreeVector
#define WITH_ST_THREEVECTOR
// compatibility with StLorentzVector
#define WITH_ST_LORENTZVECTOR

#include "StMatrix.hh"
template<class T> StMatrix;
```

Category vectors+matrices

Description This class defines matrices of arbitrary dimension as well as associated operations that can be performed on a single matrix or algebraic operations that combine matrices. Its interface is modelled very closely to that of `Matrix` in CLHEP but it is templated and can handle operations on matrices of different types. It extends on the functionality of `Matrix` in CLHEP by allowing multiplication of a matrix with a `vector` from the Standard C++ library as well as `StThreeVector` and `StLorentzVector` from the Star Class Library (see Synopsis for required `define` statements). The storage of the matrix elements is done by pointer instead of using a Standard C++ or STL container to reduce overhead of the class. Unlike CLHEP, no distinction is made between the various matrix types such as a symmetric or diagonal matrix. This increases the memory required for the storage of some matrices, however it greatly reduces the amount of code required. It also allows the class to be completely non-virtual. For the storage of matrices in a data base, this point may have to be addressed in the near future. The use of `friends` has been greatly reduced in the mathematical operations and access functions according to the Star Coding Guidelines have been included. The class retains a macro definition from the CLHEP implementation to do bounds checking.

The template argument is used to define the type associated with the elements (i.e. M_{ij}) of the matrix. This argument must be one of the floating point number data types available in the C++ language, either `float`, `double`, or `long double`. The default type is `double`. These are also specified as: `StFloat`, `StDouble`, or `StLongDouble` in the STAR CLASS LIBRARY. It should be noted that the member function `swap()` which swaps two matrices only works with matrices of the same type.

Persistence None

Related Classes Class StMatrix is a base class that may be used to derive a **StRotation** class and a **StBoost** class in the near future.

Public Constructors `StMatrix<DataType>();`
Constructs a matrix with zero rows and zero columns.

`StMatrix<DataType>(size_t p, size_t q, size_t init=0);`
Constructs a matrix with *p* rows and *q* columns. By default all the elements within the matrix are initialized to 0. If *init=1*, and the matrix is N×N, the diagonal elements are initialized to 1 (i.e. the identity matrix). If *init* is specified as 1 and the matrix is not square, the elements are all initialized to 0.

```
template<class X>
StMatrix<DataType>(const StMatrix<X>& m1)
```
Copy Constructor. Constructs a matrix with the contents of `m1`. Note that `m1` can be an object with different template arguments then self, i.e. one can instantiate a matrix of type `double` with an matrix of type `float` and vice versa.

Public Member Operators

```
template<class X>
StMatrix<DataType>
operator= (const StMatrix<X>& m1);
```
Assignment operator. Replaces the contents of self with the contents of `m1`. Note that `m1` can be an object with different template arguments then self, i.e. one can assign a matrix of type `double` to a matrix of type `float` and vice versa.

`T operator() (size_t i, size_t j) const;`
Returns components by index. Note that 1≤i≤p where p is the number of rows and 1≤j≤q where q is the number of columns in the matrix. The result for indices outside these boundaries is platform dependent. If the compiler supports exception handling and `ST_USES_EXCEPTIONS` is defined, an `out_of_range` exception is thrown, otherwise a messages is printed to cerr alerting the user to an out_of_range condition.

`T operator() (size_t i, size_t j);`
Assigns components by index. Note that 1≤i≤p where p is the number of rows and 1≤j≤q where q is the number of columns in the matrix. The result for indices outside these boundaries is platform dependent. If the compiler supports exception handling and `ST_USES_EXCEPTIONS` is defined, an `out_of_range` exception is thrown, otherwise a messages is printed to cerr alerting the user to an out_of_range condition.

`T operator[] (size_t c) const;`
Allows C style access to elements of a matrix.
Indices run from 0≤c≤(p-1) where p is the number of rows
and 0≤c≤(q-1) where q is the number of columns.

`T operator[] (size_t c);`
Allows C style assignment to elements of a matrix.
Indices run from 0≤c≤(p-1) where p is the number of rows
and 0≤c≤(q-1) where q is the number of columns.

```
StMatrix<DataType>&
operator*= (double t);
```
Multiplies each element of self by the scalar `t` and returns self.

```
StMatrix<DataType>&
operator/= (double t);
```

Divides each element of self by the scalar `t` and returns self.

```
template<class X>
StMatrix<DataType>&
operator+= (const StMatrix<X>& m2);
```
Adds matrix `m2` to self and returns self.

```
template<class X>
StMatrix<DataType>&
operator-= (const StMatrix<X>& m2);
```
Subtracts matrix `m2` from self and returns self.

```
StMatrix<DataType> operator+ ();
```
Unary plus. Returns a copy of self.

```
StMatrix<DataType> operator- ();
```
Unary minus. Returns a copy of self with all components negated.

```
template<class X>
bool operator== (const StMatrix<X>& m1);
```
Equality check. Returns `true` if self equals `m1` else `false`.

```
template<class X>
bool operator!= (const StMatrix<X>& m1);
```
Inequality check. Returns `true` if self is not equal to `m1` else `false`.

Public Member Functions

StMatrix provides 6 access functions for data members:

```
unsigned int numRow();
```
Returns the number of rows in self. Note this is not a CLHEP member function.

```
unsigned int num_row();
```
Returns the number of rows in self. Note this is provided to be backward compatible with CLHEP.

```
unsigned int numCol();
```
Returns the number of columns in self. Note this is not a CLHEP member function.

```
unsigned int num_col();
```
Returns the number of columns in self. Note this is provided to be backward compatible with CLHEP.

```
unsigned int numSize();
```
Returns the number of elements in self. Note this is not a CLHEP member function.

```
unsigned int num_size();
```
Returns the number of elements in self. Note this is provided to be backward compatible with CLHEP.

Access to the elements of the matrix are only provided via the operators **()** and **[]** as specified above.

```
StMatrix<DataType>
dot(StMatrix<DataType >& m2);
```
Returns a matrix that is the product of self and m2. Analogous to the ***=** operator.

```
StMatrix<DataType>
apply(DataType (*f)(DataType, size_t, size_t)) const;
```
Applies a user defined function `(*f)` to each element with the matrix.

```
const StMatrix<DataType> T() const;
```
Returns the transpose of self.

```
const StMatrix<DataType> transpose() const;
```
Returns the transpose of self by calling T(). Note this is not a CLHEP member function.

```
StMatrix<DataType>
sub(size_t i, size_t ii, size_t j, size_t jj) const;
```
Returns a sub matrix of self of dimension (ii-i+1)×(jj-j+1) by specifying the minimum (i) and maximum (ii) row and the the minimum (j) and maximum (jj) column.

```
void sub(size_t row, size_t col,
         const StMatrix<DataType> &m1);
```
Replaces the sub matrix specified by row×column with the matrix m1. Note that the matrix and the sub matrix must be of the same type.

```
StMatrix<DataType> inverse(size_t& ierr) const;
```
Returns a matrix that is the inverse of self. `ierr` is a status variable that is zero when successful and is one (1) when the matrix is singular.

```
virtual void invert(size_t& ierr);
```
Replaces self with its inverse. `ierr` is a status variable that is zero when successful and is one (1) when the matrix is singular.

```
DataType determinant() const;
```
Returns the determinant of the self.

```
static void swap(size_t&, size_t&);
```
Utility member function for `swap(StMatrix<DataType>&, StMatrix<DataType>&).` Interchanges the number of rows and columns.

```
static void swap(DataType *&, DataType *&);
```
Utility member function for `swap(StMatrix<DataType>&, StMatrix<DataType>&).` Interchanges the elements.

```
friend void
swap(StMatrix<DataType>& m1, StMatrix<DataType>& m2);
```
Interchanges matrix m1 with m2. Note that only matrices of the same DataType may be exchanged.

Global Operators

```
template<class DataType, class X>
StMatrix<DataType> operator*
(const StMatrix<DataType>& m1, const StMatrix<X>& m2);
```
Returns the product of m1 and m2. The matrices may be of different types, but the return type is that of the first argument. Bounds checking is done by default.

```
template <class DataType, class X>
vector<DataType>
operator*(const StMatrix<X>& m1,
          const vector<DataType>& v)
```
Multiplication of a matrix (m1) with a vector (v) from the Standard C++ library. Returns a vector of the same type as the vector that is used in the operation. Bounds checking is done by default.

```
template <class DataType, class X>
vector<DataType>
operator*(const vector<DataType>& v,
          const StMatrix<X>& m1)
```
Same as above with different ordering.

```
template <class DataType, class X>
StThreeVector<DataType> operator*
(const StMatrix<X>& m1,
 const StThreeVector<DataType>& v3)
```
Multiplication of a matrix with a StThreeVector from the Star Class library. Returns a StThreeVector of the same type as the StThreeVector that is used in the operation. `WITH_ST_THREEVECTOR` must be defined before the inclusion of **StMatrix.hh**. Bounds checking is done by default.

```
template <class DataType, class X>
StThreeVector<DataType>operator*
(const StThreeVector<DataType>& v3,
 const StMatrix<X>& m1)
```
Same as above with different ordering.

```
template <class DataType, class X>
StLorentzVector<DataType>operator*
(const StMatrix<X>& m1,
 const StLorentzVector<DataType>& v4)
```
Multiplication of a matrix with a StLorentzVector from the Star Class library. Returns a StLorentzVector of the same type as the StLorentzVector that is used in the operation. `WITH_ST_LORENTZVECTOR` must be defined before the inclusion of **StMatrix.hh**. Bounds checking is done by default.

```
template <class DataType, class X>
StLorentzVector<DataType>operator*
(const StLorentzVector<DataType>& v3
 const StMatrix<X>& m1)
```
Same as above with different ordering.

```
template<class DataType, class X>
StMatrix<DataType>operator+
(const StMatrix<DataType>& m1,const StMatrix<X>& m2)
```
Returns the sum of m1 and m2. Bounds checking is done by default.

```
template<class DataType, class X>
StMatrix<DataType>operator-
```

`(const StMatrix<DataType>& m1,const StMatrix<X>& m2)`
Returns the difference of m1 and m2 matrices. Bounds checking is done by default.

```
template<class DataType>
ostream&
operator<<(ostream& s, const StMatrix<DataType>& q)
```
Prints a formatted matrix to stdout.

```
template<class DataType>
DataType norm_infinity(const StMatrix<DataType>& m1)
```
Returns the sum of the column whose element have the largest sum in the matrix m1.

```
template<class DataType>
DataType norm1(const StMatrix<DataType>& m1)
```
Returns the sum of the row whose element have the largest sum in the matrix m1.

Examples

```
#include "StGlobals.hh"

#define MATRIX_BOUND_CHECK
#define WITH_ST_THREEVECTOR
#include "StMatrix.hh"

template <class X>
X hold1(X a, unsigned int r, unsigned int q)
{
    return a*a;
}

int main()
{
    StMatrix<StDouble> A(2,2,1);

    A(1,1) = 2;
    A(1,2) = 1;
    A(2,1) = 0;
    A(2,2) = 5;

    cout << "A=" << A << endl;

    StMatrix<StFloat> B(2,2);

    B(1,1) = 0;
    B(1,2) = 1;
    B(2,1) = 1;
    B(2,2) = 0;

    cout << "-B=" << -B;

    StMatrix<StFloat> C(A);

    cout << "C=" << C;
    cout << "A=" << C;

    C+=C;
    cout << "C+=C =>" << C << endl;
```

```
    cout << "C " << C;

    C = B+A;
    cout << "C=B+A:" << C << endl;

    unsigned int ierr;
    cout << "B.inverse(ierr)" << B.inverse(ierr) << endl;

    StMatrix<StDouble> IDENTITY(2,2,1);

    C = B*B.inverse(ierr);

    if(C == IDENTITY)
        cout << "C=B*B.inverse(ierr) == IDENTITY" << endl;
    else
        cout << "oops.." << endl;

    cout << "1st row of C" << endl;
    cout << C.sub(1,1,1,2) << endl;

    StMatrix<StDouble> K(3,3,1);
    K(1,3) = 2;
    K(2,1) = -2;
    K(1,3) = -1;

    cout << "K=" << K << endl;

    StMatrix<> TT;
    TT = K.apply(hold1);
    cout << "square each element of K:=" << TT << endl;

    StThreeVector<> a(1,1,1);
    cout << "StThreeVector a= " << a << endl;
    cout << "a*K=" << a*K << endl;
    cout << "K*a=" << K*a << endl;

    try {
        cout << "K*A=" << endl;
        cout << (K*A) << endl;

    }
    catch(exception &e){
        cout << e.what() << endl;
    }

    return 0;
}
```

Programs Output:

```
A=
            2             1
            0             5

-B=
            0            -1
           -1             0
C=
            2             1
```

```
            0             5
A=
            2             1
            0             5
C+=C =>
            4             2
            0            10

C
            4             2
            0            10
C=B+A:
            2             2
            1             5

B.inverse(ierr)
            0             1
            1             0

C=B*B.inverse(ierr) == IDENTITY
1st row of C

            1             0

K=
            1             0            -1
           -2             1             0
            0             0             1

square each element of K:=
            1             0             1
            4             1             0
            0             0             1

StThreeVector a= (1, 1, 1)
a*K=(-1, 1, 0)
K*a=(0, -1, 1)
K*A=
StMatrix<double>::dot() Incompatible matrix sizes
```

10.6 StPhysicalHelix

Summary StPhysicalHelix describes the trajectory of a charged particle in a uniform magnetic field. While its base class StHelix represents only the mathematical model StPhysicalHelix provides additional access to physical quantities associated with a particles trajectory, e.g. charge and momentum.

Synopsis `#include "StPhysicalHelix.hh"`

Category tracks

Description Other than StHelix instances of StPhysicalHelix can be created from known physical quantities as charge and momentum of a trajectory in a magnetic field. The same values can also be returned from StPhysicalHelix even if the object was instantiated with mathematical parameters (curvature, dip angle, phase, origin and orientation).

No equivalent class exists in CLHEP.

In the following only those methods are described which are specific to StPhysicalHelix are described in the following. For a full description of StHelix see section 10.3.

Persistence None

Related Classes Class StPhysicalHelix is derived from **StHelix** which defines the underlying mathematical model.

Public Constructors

```
StPhysicalHelix(StThreeVector<double> p,
                StThreeVector<double> o,
                double B, double q);
```

Constructs a helix with for a given momentum `p`, origin `o`, magnetic field `B`, and charge `q`.

```
StPhysicalHelix(const StPhysicalHelix& hlx);
```

Copy Constructor. Constructs a helix with the content of `hlx`.

Public Member Functions

```
StThreeVector<double> momentum(double B) const;
```

Returns the momentum at the origin of the helix for a given magnetic field `B`. The sign of `B` is relevant. Note that the actual origin can be obtained from the `origin()` member function.

```
StThreeVector<double> momentumAt(double s, double B);
```

Returns the momentum at pathlength `s` for a given magnetic field `B`. Note that the sign of `B` is relevant. Note that the actual position at pathlength `s` can be obtained from the `at()` member function.

```
int charge(double B) const;
```

Returns the charge of the trajectory for a given magnetic field `B`. Note that the sign of `B` is relevant.

Examples

```
#include "StPhysicalHelix.hh"

int main(int, char*)
```

```
{
    StThreeVector<double> p(1*GeV, 1.2*GeV, 0.03*GeV);
    StThreeVector<double> o;  // defaults to (0,0,0)
    const double B = 0.5*tesla;
    const double q = -1;

    cout << "Creating helix with:" << endl;
    cout << "momentum = " << p << endl;
    cout << "origin   = " << o << endl;
    cout << "charge   = " << q << endl;
    cout << "B field  = " << B << endl;

    StPhysicalHelix track(p, o, B, q);

    cout << "The helix parameters are:" << endl;
    cout << track << endl;

    cout << "Scanning from s=0 to s=2 m:" << endl;

    for (double s=0; s<=2*meter; s+=40*centimeter) {
        p = track.momentumAt(s, B);
        cout << "s=" << s << " mm" << "\t-> p = " << p
             << " MeV" << "\t |p| = " << abs(p) << endl;
    }

    return 0;
}
```

Programs Output:

```
Creating helix with:
momentum = (1000, 1200, 30)
origin   = (0, 0, 0)
charge   = -1
B field  = 0.0005
The helix parameters are:
(curvature = 9.59612e-05, dip angle = 0.0192032,
 phase = -0.694738, h = 1, origin = (0, 0, 0))
Scanning from s=0 to s=2 m:
s=0 mm    -> p = (1000, 1200, 30) MeV         |p| = 1562.34
s=400 mm  -> p = (953.222, 1237.48, 30) MeV   |p| = 1562.34
s=800 mm  -> p = (905.04, 1273.15, 30) MeV    |p| = 1562.34
s=1200 mm -> p = (855.526, 1306.93, 30) MeV   |p| = 1562.34
s=1600 mm -> p = (804.752, 1338.8, 30) MeV    |p| = 1562.34
s=2000 mm -> p = (752.792, 1368.69, 30) MeV   |p| = 1562.34
```

10.7 StPrompt

Summary StPrompt is a templated function to prompt the user for input of various type.

Synopsis `#include "StPrompt.hh"`

Category utilities

Description Template function to prompt the user for input of type T. Reads the new input value, except if <CR> is pressed directly after the prompt. In this case the previous value (indicated in brackets) stays untouched.

This function is essentially useful for debugging purposes and in small programs (prototyping) where user input is required.

Syntax `void StPrompt(const char* txt,T& var);`
Prompt for a value of type `T`. The prompt string includes the given text `txt` and the current value of `var` in square brackets (default value). The new input value is assigned to `var` except if <CR> is pressed directly after the prompt string in which case the content of `var` is not altered. For non-build-in data types make sure the input and output operators << and >> are defined. (Note, that both operators are defined for StThreeVector!)

In addition to the more general version a few specializations are provided:

`void StPrompt(const char* txt, bool& var);`
In this case the functions accepts *true, t, yes, y, on, 1* as `true`. Everything else yields `false` (except <CR>).
Note: On Sun platforms the native CC compiler does not provide `bool` as fundamental type and is therefore implemented as type `int`. As a consequence
`void StPrompt(const char* txt, bool& var);`
cannot be distinguished from
`void StPrompt(const char* txt, int& var);`.
On Sun platforms (`STAR_SYS=sun4x*`) this specializations is disabled and replaced by: `void StBoolPrompt(const char* txt, bool& var);`

`void StPrompt(const char *txt, char* var, int len);`
Prompts for a C-like character string. In this case the length of the array has to be passed in order to prevent memory violations [7].

`void StPrompt();`
Prints ”– Press return to continue – ” on the screen and waits until <CR> is pressed.

Examples

```
#include "StPrompt.hh"
#include "StThreeVector.hh"
#include <string>

int main()
{
        StThreeVector<float>  vec2(1,2,3);
```

[7] The STL `string` class should always be preferred over C-like character strings.

```
        StPrompt("Enter new 3-vector", vec2);
        cout << "new value: " << vec2 << endl;

        string filename("var.cc");
        char   othername[16] = "short.c";
        double var = 1/3.;

        bool answer = true;
        while (answer) {
            StPrompt("Enter file name", filename);
            cout << "new value: " << filename << endl;
            StPrompt("Enter other name", othername, 16);
            cout << "new value: " << othername << endl;

            StPrompt("any number", var);
            cout << "new value: " << var << endl;

            StPrompt("more questions", answer);
        }

        StPrompt();
        cout << "bye" << endl;
        return 0;
}
```

Programs Output:

```
Enter new 3-vector [(1, 2, 3)]: 9 9 9<CR>
new value: (9, 9, 9)
Enter file name [var.cc]: <CR>
new value: var.cc
Enter other name [short.c]: long.c<CR>
new value: long.c
any number [0.333333]: 8.4<CR>
new value: 8.4
more questions [true]: n<CR>
-- Press return to continue --<CR>
bye
```

10.8 StThreeVector

Summary StThreeVector is a templated general 3-vector class defining vectors in three dimension.

Synopsis
```
#include "StThreeVector.hh"
template<class T> StThreeVector;
```

Category vectors+matrices

Description This class defines a general 3-vector which can be used to represent space points and 3-momenta. It has a large set of member functions, member operators and associated global operators which allow to multiply, subtract and add vectors with other vectors or scalar variables, calculate cross products and angles between vectors, and much more. Once defined its coordinates can be obtained in Cartesian, cylindrical and spherical representation. Its interface is essentially the same as `ThreeVector` in CLHEP with two significant differences: (i) it is templated vector and (ii) it is a concrete class, i.e. it is not derived from any other class. It offers essentially the same functionality as the CLHEP version but is more flexible in terms of precision and storage optimisation; i.e. in order to minimize for memory and storage volume a `StThreeVector`'s with type argument `float` can be used but easily transformed into a double precision version for computation when higher accuracy is needed. In addition to the CLHEP version there are a few member functions added which are useful in the context of Heavy-Ion Physics such as `pseudoRapidity()`.

The template argument is used to define the type associated with the x, y, z components. This argument must be one of the floating point number data types available in the C++ language, either `float`, `double`, or `long double`. The default type is `double`.

Please note that `StThreeVector` is *not* virtual. This is a compromise in order to minimize the storage size, i.e. to avoid the additional ballast of the virtual table pointer.

Persistence None

Related Classes Class **StLorentzVector** is derived from StThreeVector defining a 4-dimensional Lorentz vector.

Public Constructors

`StThreeVector<T>();`
Constructs a 3-vector with all components initialized to 0.

`StThreeVector<T>(T x, T y, T z);`
Constructs a 3-vector with given components x, y and z.

```
template<class X>
StThreeVector<T>(const StThreeVector<X> &vec)
```
Copy constructor. Constructs a 3-vector with the content of `vec`. Note that `vec` can be an object with different template arguments then self, i.e. one can instantiate a vector of type `double` with an vector of type `float` and vice versa.

Public Member Operators

```
template<class X>
StThreeVector<T>
operator= (const StThreeVector<X> &vec);
```
Assignment operator. Replaces the content of self with the content of vec. Note that vec can be an object with different template arguments then self, i.e. one can assign a vector of type double to a vector of type float and vice versa.

```
T operator() (size_t i) const;
```
Returns components by index. Note that the first index (the x-component) has index 0. The result for indices > 2 is platform dependent. If the compiler supports exception handling an out_of_range exception is thrown.

```
T operator[] (size_t i) const;
```
Same as operator() above.

```
StThreeVector<T> operator- ();
```
Unary minus. Returns copy of self with all components negated.

```
StThreeVector<T> operator+ ();
```
Unary plus. Returns copy of self.

```
StThreeVector<T>&
operator*= (double c);
```
Returns self multiplied by scalar c.

```
StThreeVector<T>&
operator/= (double c);
```
Returns self divided by scalar c.

```
template<class X>
bool
operator== (const StThreeVector<X>& vec);
```
Equality check. Returns true if self equals vec else false.

```
template<class X>
bool
operator!= (const StThreeVector<X>& vec);
```
Inequality check. Returns true if self is not equal to vec else false.

Public Member Functions

```
void setX(T x);
```
Set the x-component in Cartesian coordinate system.

```
void setY(T y);
```
Set the y-component in Cartesian coordinate system.

```
void setZ(T z);
```
Set the z-component in Cartesian coordinate system.

```
void setPhi(T ph);
```
Set the azimuthal angle in spherical coordinate system.

```
void setTheta(T ph);
```
Set the polar angle in spherical coordinate system.

`void setMag(T r);`
Set the magnitude of the vector keeping the polar and azimutal angles constant.

`void setMagnitude(T r);`
Set the magnitude of the vector keeping the polar and azimutal angles constant.

`T x() const;`
Returns the x-component in Cartesian coordinate system.

`T y() const;`
Returns the y-component in Cartesian coordinate system.

`T z() const;`
Returns the z-component in Cartesian coordinate system.

`T phi() const;`
Returns the azimuth angle.

`T theta() const;`
Returns the polar angle.

`T cosTheta() const;`
Returns the cosine of the polar angle.

`T mag2() const;`
Returns the magnitude squared of self (r^2 in a spherical coordinate system).

`T mag() const;`
Returns the magnitude of self (r in a spherical coordinate system). Note that the same value can be obtained using the overloaded `abs()` function (see below).

`T perp2() const;`
Returns the transverse component squared (R^2 in cylindrical coordinate system).

`T perp() const;`
Returns the transverse component (R in cylindrical coordinate system).

`T pseudoRapidity() const;`
Returns the pseudo-rapidity, i.e. $-\ln(\tan\theta/2)$ of the vector. Note that this value is only valid under the assumptions that the vector origins from the center of the referring reference frame. Be also aware that this member function is *not* present in the CLHEP ThreeVector class.

`StThreeVector<T> unit() const;`
Returns a unit vector parallel to self.

`T massHypothesis(T mass) const;`
Calculates what the energy component of a 4-vector should be given a mass. Returns:

$$\sqrt{(*this)^2 + (mass)^2}$$

Note this function is not provided in CLHEP.

`StThreeVector<T> orthogonal() const;`
Returns a vector orthogonal to self. The use of this member function is discouraged as the user should use the dot and cross products to produce such an object.

```
void rotateX(T angle);
```
Rotates a vector about the $\hat{x}$ axis according to a right-handed coordinate system by an angle specified by `angle`.

```
void rotateY(T angle);
```
Rotates a vector about the $\hat{y}$ axis according to a right-handed coordinate system by an angle specified by `angle`.

```
void rotateZ(T angle);
```
Rotates a vector about the $\hat{z}$ axis according to a right-handed coordinate system by an angle specified by `angle`.

`Note:` No provision to rotate a vector about an arbitrary axis is provided in `StThreeVector` as in CLHEP because it is much more neatly done by a matrix or rotation class and this would result in an additional dependency in `StThreeVector`. In order to do such a rotation, see `StRotation`.

```
template<class X>
T angle(const StThreeVector<X>& vec) const;
```
Returns the angle between self and `vec`.

```
template<class X>
T dot(const StThreeVector<X>& vec) const;
```
Returns the scalar product of self and `vec`.

```
template<class X>
StThreeVector<T>
cross(const StThreeVector<X>& vec) const;
```
Returns the cross product of self and `vec`.

Global Functions

```
template<class T>
T abs(const StThreeVector<T>& vec);
```
Returns the magnitude of `vec`. Same as `vec->mag()`. Be also aware that this feature is *not* provided by the CLHEP.

```
template<class T, class X>
StThreeVector<T>
cross_product(const StThreeVector<T>& v1,
              const StThreeVector<X>& v2);
```
Returns the cross product of `v1` and `v2`. The type of the returned vector is determined by the type argument of the first vector `v1`. Note that the name was chosen for compatibility with the STL.

Global Operators

```
template<class T, class X>
StThreeVector<T>
operator+ (const StThreeVector<T>& v1,
           const StThreeVector<X>& v2);
```
Returns the sum of `v1` and `v2`. The type of the returned vector is determined by the type argument of the first vector `v1`.

```
template<class T, class X>
StThreeVector<T>
```

```
operator- (const StThreeVector<T>& v1,
           const StThreeVector<X>& v2);
```

Returns the `v1` minus `v2`. The type of the returned vector is determined by the type argument of the first vector `v1`.

```
template<class T, class X>
T operator* (const StThreeVector<T>& v1,
             const StThreeVector<X>& v2);
```

Returns the scalar product of `v1` and `v2`. The type of the returned value is determined by the type argument of the first vector `v1`.

```
template<class T>
StThreeVector<T>
operator* (const StThreeVector<T>& vec,
           double c);
```

Returns vector `vec` multiplied by scalar c.

```
template<class T>
StThreeVector<T>
operator* (double c,
           const StThreeVector<T>& vec);
```

Returns vector `vec` multiplied by scalar c.

```
template<class T, class X>
StThreeVector<T>
operator/ (const StThreeVector<T>& vec,
           X c);
```

Returns vector `vec` divided by scalar c.

```
template<class T>
ostream&
operator<< (ostream& os,
           const StThreeVector<T>& vec);
```

Prints vector `vec` to output stream `os`.

```
template<class T>
istream&
operator>> (istream& is,
           StThreeVector<T>& vec);
```

Reads vector `vec` from input stream `is`. Be also aware that this operator is *not* provided by the CLHEP ThreeVector class.

Examples

```
#include "StThreeVector.hh"

int main()
{
    StThreeVector<double>  a;
    StThreeVector<double>  b(1,2,3);
    StThreeVector<float>   c(b);

    cout << "b = " << b << endl;
    cout << "c = " << c << endl;
```

```
    // add two vectors
    a = b+c;
    cout << "a = b+c = " << a << endl;

    // check for inequality
    if (a != b*2) {
        cerr << "Oops ..." << endl;
        return 1;
    }

    // modify components of vector c
    c.setX(4);
    c.setY(7);
    cout << "c = " << c << endl;

    // inner product
    double d = a*c;
    cout << "a*c = " << d << endl;

    // angle
    cout << "angle(a,c) = "
         << a.angle(c) << endl;

    // cross product
    cout << "a X c = "
         << a.cross(c) << endl;

    // pseudo rapidity
    cout << "pseudo rapidity(c) = "
         << c.pseudoRapidity() << endl;

    return 0;
}
```

Programs Output:

```
b = (1, 2, 3)
c = (1, 2, 3)
a = b+c = (2, 4, 6)
c = (4, 7, 3)
a*c = 54
angle(a,c) = 0.575631
a X c = (-30, 18, -2)
pseudo rapidity(c) = 0.364012
```

10.9 Random

Summary Random provides a mechanism to generate pseudo-random numbers in a variety of distributions. Provided are several engines that provide seeds to classes that are able to generate random numbers according to several pre-determined distribution.

Synopsis

```
#include "Random.hh"
// At least one engine
#include "JamesRandom.h"   // The engine used by default
#include "RanecuEngine.h"
#include "RanluxEngine.h"
#include "DRand48Engine.h"
#include "RandEngine.h"
// And a type of distribution
#include "RandFlat.h"
#include "RandPoisson.h"
#include "RandExponential.h"
#include "RandGauss.h"
#include "RandBreitWigner.h"
```

Category random

Description Random consists of a series of classes that provide mechanisms to generate pseudo-random numbers according to one of five pre-determined distributions:

- Flat
- Poissonian
- Exponential
- Gaussian
- Breit-Wigner

In order to generate a random number according to these distributions a pseudo-random number engine must be provided. These engines utilize either a predefined (static) seed table or a seed given by the user and one of several prescriptions to generate a pseudo-random number in a flat distribution. These numbers are then used to generate a random number based on a characteristic distribution.

It should be noted that these classes are nearly exact copies of those used in CLHEP (hence the lack of the “**St**” prefix) adapted for use in the STAR Class Library. Member functions that are not contained within CLHEP are specified as such.

- Arrays passed to member functions should be replaced with STL containers.
- A reduction of member functions by using default values for arguments.
- ...

There exists two classes which define the two different components of the category:

- `RandomEngine.h` contains an abstract class *HepRandomEngine* which defines the interface for all random engines.
- `Random.h` defines *HepRandom* which is a base class from which all distribution classes inherit. It defines both static and non-static interfaces as well as the default engine generator (i.e. HepJamesRandom).

Persistence None

Related Classes **The Engines:**

Class *HepRandomEngine* (contained in `RandomEngine.h`) is a purely abstract class which defines the user interface for all engines:

```
double flat();
```
Returns a pseudo-random number from a flat distribution in the interval (0,1).

```
void flatArray(const int size, double* vec);
```
Fills an array `vec` of size `size` with random numbers derived from a flat distribution.

```
void flatArray(vector<double>& vec);
```
Fills a vector `vec` with random numbers derived from a flat distribution. This is not supported by CLHEP.

```
void setSeed(long seed, int init);
```
(Re)-initialize the status of the algorithm with a user specified seed.

```
void setSeeds(const long* seeds, int init);
```
(Re)initializes the generator with a zero terminated list of seeds.

```
void saveStatus() const;
```
Saves the current status of the instantiated engine in a file (`.conf`). This provides a mechanism to recover a series of random numbers.

```
void restoreStatus();
```
Reads from a file (specific to the instantiated engine in use) and restore the last saved engine configuration. For use with `saveStatus()`.

```
showStatus() const;
```
Dumps the current engine status to `stdout`.

```
long getSeed() const;
```
Returns the current seed from the current generator.

```
const long* getSeeds() const;
```
Returns the current array of seeds from the current generator.

```
void getTableSeeds(long* seeds, int index) const;
```
Returns the seed values stored in the global `seedTable` that is located at the `index` position.

The Specific Engines:

HepJamesRandom implements the algorithm of Marsaglia-Zaman RANMAR which is described in *F. James, Comp. Phys. Comm.* **60** *(1990) 329*. It is a component of the MATHLIB HEP library for pseudo-random number generation. This is the default random engine invoked by each distribution unless the user specifies a different one.

DRand48Engine uses `drand48()` and `srand48()` functions from the C standard library to implement the basic `flat()` basic distribution and for setting seeds. Note that this file is part of the Geant4 simulation toolkit.

RandEngine uses the `rand()` and `srand()` functions from the C standard library to implement the `flat()` basic distribution and for setting seeds. Note that this file is part of Geant4 simulation toolkit.

RanecuEngine is an algorithm that is part of the MATHLIB HEP library. Seeds are taken from a seed table and given an index. The `getSeed()` member function returns the current index of the seed table, while the `getSeeds()` member function returns a pointer to the local table of seeds at the current index. Note that this file is part of Geant4 simulation toolkit.

RanluxEngine is an algorithm originally implemented in FORTRAN by Fred James as part of the MATHLIB HEP library. The initialisation is carried out using a Multiplicative Congruential generator using formula constants of L'Ecuyer as described in *F. James, Comp. Phys. Comm. 60 (1990) 329*. Note that this file is part of Geant4 simulation toolkit.

The Distributions:

Class *HepRandom* contained in `Random.h` is a base class from which all distribution classes inherit. An object of this class is instantiated by default within the HEP Random module. An instantiated *HepJamesRandom* engine is used as a default algorithm for pseudo-random number generation. HepRandom defines a static private data member `theGenerator` and a set of static inlined methods to manipulate it. By means of `theGenerator` the user can:

- change the underlying engine algorithm.
- get and set the seeds.
- use any kind of defined random distribution.

Note: Distribution classes inherit from *HepRandom* and define **both** static and non-static interfaces.

Public Constructors

`HepRandom();`
Contructor without a seed uses the default engine (i.e. *HepJamesRandom*).

`HepRandom(long seed);`
Contructor with a seed specified by the user which uses the default engine (i.e. *HepJamesRandom*).

`HepRandom(HepRandomEngine& algorithm);`
Constructor with a user specified generating engine. When algorithm is passed by reference, it will **not** be deleted by the *HepRandom* destructor.

`HepRandom(HepRandomEngine* algorithm);`
Constructor with a user specified generating engine. When algorithm is passed by pointer, it will be deleted by the HepRandom destructor.

Public Member Functions

`double flat();`
Returns the flat value in the interval (0,1).

`double flat (HepRandomEngine* theNewEngine);`
Returns a flat value in the interval (0,1) where `theNewEngine` is used as the Random Engine.

`void flatArray(const int size, double* vect);`
Fills the array `vect` of size `size` with flat random values. Please note that when STL containers are implemented the size will no longer be a required argument.

```
void flatArray(HepRandomEngine* theNewEngine,
                        const int size, double* vect);
```
Fills the array `vect` of size `size` with flat random values, given a user specified Random Engine. Please note that when STL containers are implemented the size will no longer be a required argument.

Static Member Functions

`void setTheSeed(long seed, int lux=3);`
Specifies the seed for the engine.

`long getTheSeed();`
Returns the static definition for the seed.

`void setTheSeeds(const long* seeds, int aux=-1);`
Specifies a table of seeds for the engine to use.

`const long* getTheSeeds();`
Returns the table of seeds currently in use.

`void getTheTableSeeds (long* seeds, int index);`
Returns the table of seeds starting from a specific `index` position.

`HepRandom* getTheGenerator();`
Return a pointer to the current static generator.

`void setTheEngine (HepRandomEngine* theNewEngine);`
Specifies the engine to be used in the pseudo-random number generation.

`HepRandomEngine* getTheEngine();`
Returns a pointer to the current engine in use.

`void saveEngineStatus();`
Saves the current engine status in a `.conf` file.

`void restoreEngineStatus();`
Restores the status of an engine to that specified in a `.conf` file. In the absence of such a file, nothing is done.

`void showEngineStatus();`
Prints the status of the random engine to `stdout`.

Public Member Operators

`double operator()();`
Generates a single random number.

The Specific Distribution classes include functionality that is specific to their own properties:

RandFlat defines methods for generating flat random numbers which can be either `integer` or `double` precision. It also provides static methods to fill arrays of a specified size.

Public Member Constructors

`RandFlat(HepRandomEngine& anEngine);`
Constructor instantiates a `RandFlat` distribution object defining a local engine (by reference) for it. The corresponding engine object will **not** be deleted by the `RandFlat` destructor.

`RandFlat(HepRandomEngine* anEngine);`
Constructor instantiates a `RandFlat` distribution object defining a local engine (by pointer) for it. The corresponding engine object will be deleted by the `RandFlat` destructor.

Public Static Member Functions

Static methods to generate random values using the static generator are provided:

`double shoot();`
Returns a double precision number from a flat distribution in the interval (0,1).

`shoot(double width);`
Returns a double precision number from a flat distribution in the interval (0,width).

`double shoot(double a, double b);`
Returns a double precision number from a flat distribution in the interval (a,b).

`long shootInt(long n);`
Returns an integer number from a flat distribution in the interval (0,n).

`long shootInt(long m, long n);`
Returns an integer number from a flat distribution in the interval (m,n).

`int shootBit();`
Returns either 0 or 1 according to a flat distribution.

`void shootArray(const int size, double* vect);`
Fills an array `vect` of size `size` with double precision numbers in the interval (0,1).

`void shootArray(vector<double>& vec);`
Fills a vector `vec` with double precision numbers in the interval (0,1). This is not supported in CLHEP.

```
void shootArray(const int size, double* vect,
                                double lx, double dx);
```
Fills an array `vect` of size `size` with double precision numbers in the interval (lx,dx).

```
void shootArray(vector<double>& vec, double lx, double dx);
```
Fills a vector `vec` with double precision numbers in the interval (lx,dx). This is not supported in CLHEP.

```
double shoot(HepRandomEngine* anEngine);
```
Returns a double precision number in the interval (0,1) where the engine is specified by the user.

```
double shoot(HepRandomEngine* anEngine,
                              double width);
```
Returns a double precision number in the interval (0,width) where the engine is specified by the user.

```
double shoot(HepRandomEngine* anEngine,
                            double a, double b);
```
Returns a double precision number in the interval (a,b) where the engine is specified by the user.

```
long shootInt(HepRandomEngine* anEngine, long n);
```
Returns an integer number in the interval (0,n) where the engine is specified by the user.

```
long shootInt(HepRandomEngine* anEngine,
                              long m, long n);
```
Returns an integer number in the interval (m,n) where the engine is specified by the user.

```
ont shootBit(HepRandomEngine* anEngine);
```
Returns 0 or 1 where the engine is specified.

```
void shootArray(HepRandomEngine* anEngine,
                  const int size, double* vect);
```
Fills an array `vect` of size `size` with double precision numbers in the interval (0,1) where the engine is specified.

```
void shootArray(HepRandomEngine* anEngine,
                         vector<double>& vec);
```
Fills a vector `vec` with double precision numbers in the interval (0,1) where the engine is specified. This is not supported in CLHEP.

```
void shootArray(HepRandomEngine* anEngine,
              const int size, double* vect,
                      double lx, double dx);
```
Fills an array `vect` of size `size` with double precision numbers in the interval (lx,dx) where the engine is specified.

```
void shootArray(HepRandomEngine* anEngine,
     vector<double>& vec, double lx, double dx);
```
Fills a vector `vect` with double precision numbers in the interval (lx,dx) where the engine is specified. This is not supported in CLHEP.

The following methods use the localEngine to shoot random values, by-passing the static generator.

`double fire();`
Returns a double precision number in the interval (0,1).

`double fire(double width);`
Returns a double precision number in the interval (0,width).

`double fire(double a, double b);`
Returns a double precision number in the interval (a,b).

`long fireInt(long n);`
Returns an integer number in the interval (0,n).

`long fireInt(long m, long n);`
Returns an integer number in the interval (m,n).

`int fireBit();`
Returns either 0 or 1.

`void fireArray(const int size, double* vect);`
Fills an array `vect` of size `size` with double precision numbers in the interval (0,1).

`void fireArray(vector<double>& vec);`
Fills a vector `vec` with double precision numbers in the interval (0,1). This is not supported in CLHEP.

```
void fireArray(const int size, double* vect,
                          double lx, double dx);
```
Fills an array `vect` of size `size` with double precision numbers in the interval (lx,dx)

`void fireArray(vector<double>& vec, double lx, double dx);`
Fills a vector `vec` with double precision numbers in the interval (lx,dx) This is not supported in CLHEP.

RandGauss defines methods for generating random numbers which are distributed in a Gaussian manner:

Public Member Constructors

`RandGauss(HepRandomEngine& anEngine);`
Constructor to instantiate a gaussian random number generator where `anEngine` is the generating engine. The corresponding engine object will **not** be deleted by the `RandGauss` destructor.

`RandGauss(HepRandomEngine* anEngine);`
Constructor to instantiate a gaussian random number generator where `anEngine` is the generating engine. The corresponding engine object will be deleted by the `RandGauss` destructor.

Public Static Member Functions

`double shoot();`
Returns a double precision number from a Gaussian distribution where the mean is 0 and standard deviation is 1.

`double shoot(double mean, double stdDev);`
Returns a double precision number from a Gaussian distribution where the mean is `mean` and standard deviation is given by `stdDev`.

```
void shootArray(const int size, double* vect,
         double mean=0.0, double stdDev=1.0);
```
Fills an array `vect` of size `size` with double precision number from a distribution where the mean is `mean` (0 by default) and standard deviation is `stdDev` (1 by default).

```
void shootArray(vector<double>& vec,
         double mean=0.0, double stdDev=1.0);
```
Fills a vector `vec` with double precision number from a distribution where the mean is `mean` (0 by default) and standard deviation is `stdDev` (1 by default). This is not supported in CLHEP.

Static methods to shoot random values using a given engine bypassing the static generator are also provided:

```
double shoot(HepRandomEngine* anEngine);
```
Returns a double precision number from a Gaussian distribution where the mean is 0 and standard deviation is 1 and the engine is specified by the user.

```
double shoot(HepRandomEngine* anEngine,
                 double mean, double stdDev);
```
Returns a double precision number from a Gaussian distribution where the mean is `mean` and standard deviation is `stdDev` and the engine is specified.

```
void shootArray(HepRandomEngine* anEngine,
       const int size,double* vect,
       double mean=0.0, double stdDev=1.0);
```
Fills an array `vect` of size `size` with double precision numbers from a Gaussian distribution where the mean is `mean` and standard deviation is `stdDev` and the engine is specified.

```
void shootArray(HepRandomEngine* anEngine,
       vector<double>& vec,
       double mean=0.0, double stdDev=1.0);
```
Fills a vector `vec` with double precision numbers from a Gaussian distribution where the mean is `mean` and standard deviation is `stdDev` and the engine is specified. This is not supported by CLHEP.

Methods using the local engine to generate random values, bypassing the static generator are also provided:

```
double fire();
```
Returns a double precision number from a Gaussian distribution where the mean is 0 and standard deviation is 1.

```
double fire(double mean, double stdDev);
```
Returns a double precision number from a Gaussian distribution where the mean is `mean` and standard deviation is `stdDev`.

```
void fireArray(const int size, double* vect,
        double mean=0.0, double stdDev=1.0);
```
Fills an array `vect` of size `size` with double precision number from a distribution

where the mean is `mean` (0 by default) and standard deviation is `stdDev` (1 by default).

```
void fireArray(vector<double>& vec,
        double mean=0.0, double stdDev=1.0);
```

Fills a vector `vec` with double precision number from a distribution where the mean is `mean` (0 by default) and standard deviation is `stdDev` (1 by default). This is not supported in CLHEP.

RandExponential defines methods for generating random numbers distributed according to an exponential distribution.

Public Member Constructors

```
RandExponential(HepRandomEngine& anEngine);
```

Constructor to instantiate a *RandExponential* distribution object and specifying a local engine for it. The engine passed by reference will **not** be deleted by the RandExponential destructor.

```
RandExponential(HepRandomEngine* anEngine );
```

Constructor to instantiate a *RandExponential* distribution object and specifying a local engine for it. The engine passed by pointer will be deleted by the RandExponential destructor.

Public Static Member Functions

```
double shoot();
```

Returns a double precision number from an exponential distribution where the mean is 1.

```
double shoot(double mean);
```

Returns a double precision number from an exponential distribution where the mean is specified by `mean`.

```
void shootArray(const int size, double* vect,
                                double mean=1.0);
```

Fills an array `vect` of size `size` from an exponential distribution where the mean is specified by `mean` (default is 1).

```
void shootArray(vector<double>& vec, double mean=1.0);
```

Fills a vector `vec` from an exponential distribution where the mean is specified by `mean` (default is 1). This is not supported in CLHEP.

Static methods to generate random values using a given engine bypassing the static generator are also provided.

```
double shoot(HepRandomEngine* anEngine);
```

Returns a double precision number from an exponential distribution where the mean is 1 and the engine is specified by `anEngine`.

```
double shoot(HepRandomEngine* anEngine,
                                double mean);
```

Returns a double precision number from an exponential distribution where the mean is specified by `mean` and the engine is specified by `anEngine`.

```
void shootArray(HepRandomEngine* anEngine,
const int size, double* vect, double mean=1.0);
```

Fills an array `vect` of size `size` from an exponential distribution where the mean is specified by `mean` (default is 1) and the engine is specified by `anEngine`.

```
void shootArray(HepRandomEngine* anEngine,
     vector<double>& vec, double mean=1.0);
```

Fills a vector `vec` from an exponential distribution where the mean is specified by `mean` (default is 1) and the engine is specified by `anEngine`. This is not supported in CLHEP.

Methods using the local Engine to generate random values, bypassing the static generator are also provided:

```
double fire();
```

Returns a double precision number from an exponential distribution where the mean is 1.

```
double fire(double mean);
```

Returns a double precision number from an exponential distribution where the mean is specified by `mean`.

```
void fireArray(const int size, double* vect,
                                  double mean=1.0);
```

Fills an array `vect` of size `size` from an exponential distribution where the mean is specified by `mean` (default is 1).

```
void fireArray(vector<double>& vec, double mean=1.0);
```

Fills a vector `vec` from an exponential distribution where the mean is specified by `mean` (default is 1). This is not supported in CLHEP.

RandPoisson defines methods for generation numbers according to a Poisson distribution. The algorithm was taken from *W. H. Press et al., Numerical Recipes in C, Second Edition*.

Public Member Functions

```
RandPoisson(HepRandomEngine& anEngine);
```

Constructor to instantiate a *RandPoisson* distribution and specifying a local engine for it. The engine `anEngine` passed by reference will **not** be deleted by the *RandPoisson* destructor.

```
RandPoisson( HepRandomEngine* anEngine);
```

Constructor to instantiate a *RandPoisson* distribution and specifying a local engine for it. The engine passed by pointer will be deleted by the *RandPoisson* destructor.

Static methods to shoot random values using the static generator are provided:

```
long shoot(double mean=1.0);
```

Returns an integer from a Poissonian distribution with a mean specified by `mean` (default is 1).

```
void shootArray(const int size, long* vect,
                                double mean=1.0);
```

Fills an integer array `vect` of size `size` from a Poissonian distribution with a mean specified by `mean` (default is 1).

```
void shootArray(vector<long>& vec, double mean=1.0);
```
Fills an integer vector `vec` from a Poissonian distribution with a mean specified by `mean` (default is 1). This is not supported by CLHEP.

Static methods to shoot random values using a given engine bypassing the static generator are also provided:

```
long shoot(HepRandomEngine* anEngine,
                                   double mean=1.0);
```
Returns an integer from a Poissonian distribution with a mean specified by `mean` (default is 1) and the engine specified by `anEngine`.

```
void shootArray(HepRandomEngine* anEngine,
      const int size, long* vect, double mean=1.0 );
```
Fills an integer array `vect` of size `size` from a Poissonian distribution with a mean specified by `mean` (default is 1) and the engine specified by `anEngine`.

```
void shootArray(HepRandomEngine* anEngine,
         vector<long>& vec, double mean=1.0 );
```
Fills an integer vector `vec` from a Poissonian distribution with a mean specified by `mean` (default is 1) and the engine specified by `anEngine`. This is not supported in CLHEP.

Methods using the localEngine to shoot random values, bypassing the static generator are also provided:

```
long fire(double mean=1.0);
```
Returns an integer from a Poissonian distribution with a mean specified by `mean` (default is 1).

```
void fireArray(const int size, long* vect,
                                    double mean=1.0);
```
Fills an integer array `vect` of size `size` from a Poissonian distribution with a mean specified by `mean` (default is 1).

```
void fireArray(vector<long>& vec, double mean=1.0);
```
Fills an integer vector `vec` from a Poissonian distribution with a mean specified by `mean` (default is 1). This is not supported by CLHEP.

RandBreitWigner defines methods for generating random numbers according to the Breit-Wigner distribution algorithms. Either the mean or the square of the mean may be specified:

Public Constructors

```
RandBreitWigner(HepRandomEngine& anEngine);
```
Constructor to instantiate a *RandBreitWigner* distribution object and defines a local engine (`anEngine`) for it. The engine passed by reference will **not** be deleted by the *RandBreitWigner* destructor.

```
RandBreitWigner ( HepRandomEngine* anEngine );
```
Constructor to instantiate a *RandBreitWigner* distribution object and defines a local engine (`anEngine`) for it. The engine passed by pointer will be deleted by the *RandBreitWigner* destructor.

Public Static Member Functions

Static methods to generate random values using the static generator are provided:

```
double shoot(double mean=1.0, double gamma=0.2);
```
Returns a double precision number from a Breit-Wigner distribution where the mean is `mean` (default is 1) and Γ is `gamma` (default is .2).

```
double shoot(double mean, double gamma, double cut);
```
Returns a double precision number from a Breit-Wigner distribution where the mean is `mean` (default is 1) and Γ is `gamma` (default is .2) and cut is `cut` (default is 1).

```
double shootM2(double mean=1.0, double gamma=0.2);
```
Returns a double precision number from a Breit-Wigner distribution where the mean is positive `mean` (default is 1) and Γ is `gamma` (default is .2).

```
double shootM2(double mean, double gamma,
                              double cut);
```
Returns a double precision number from a Breit-Wigner distribution where the mean is positive `mean` (default is 1) and Γ is `gamma` (default is .2) and cut is `cut` (default is 1).

```
void shootArray(const int size, double* vect,
 double mean=1.0, double gamma=0.2, double cut=1.0);
```
Fills an array `vect` of size `size` with double precision number from a Breit-Wigner distribution where the mean is `mean` (default is 1) and Γ is `gamma` (default is .2) and cut is `cut` (default is 1).

```
void shootArray(vector<double>& vec,
 double mean=1.0, double gamma=0.2, double cut=1.0);
```
Fills a vector `vec` with double precision number from a Breit-Wigner distribution where the mean is `mean` (default is 1) and Γ is `gamma` (default is .2) and cut is `cut` (default is 1). This is not supported in CLHEP.

Static methods to generate random values using a given engine bypassing the static generator are also provided:

```
double shoot(HepRandomEngine* anEngine,
            double mean=1.0, double gamma=0.2);
```
Returns a double precision number from a Breit-Wigner distribution where the mean is `mean` (default is 1) and Γ is `gamma` (default is .2) and the engine is specified by `anEngine`.

```
double shoot(HepRandomEngine* anEngine,
     double mean, double gamma, double cut);
```
Returns a double precision number from a Breit-Wigner distribution where the mean is `mean` (default is 1) and Γ is `gamma` (default is .2) and the engine is specified by `anEngine`.

```
double shootM2(HepRandomEngine* anEngine,
       double mean=1.0, double gamma=0.2);
```
Returns a double precision number from a Breit-Wigner distribution where the mean is positive `mean` (default is 1), Γ is `gamma` (default is .2), and the engine is specified by `anEngine`.

```
double shootM2(HepRandomEngine* anEngine,
  double mean,  double gamma, double cut);
```
Returns a double precision number from a Breit-Wigner distribution where the mean is positive `mean` (default is 1), Γ is `gamma` (default is .2), and the engine is specified by `anEngine`.

```
void shootArray(HepRandomEngine* anEngine,
   const int size, double* vect, double mean=1.0,
               double gamma=0.2, double cut=1.0);
```
Fills an array `vect` of size `size` with double precision number from a Breit-Wigner distribution where the mean is `mean` (default is 1), Γ is `gamma` (default is .2) and cut is `cut` (default is 1), and the engine is specified by `anEngine`.

```
void shootArray(HepRandomEngine* anEngine,
            vector<double>& vec, double mean=1.0,
               double gamma=0.2, double cut=1.0);
```
Fills a vector `vec` with double precision number from a Breit-Wigner distribution where the mean is `mean` (default is 1), Γ is `gamma` (default is .2) and cut is `cut` (default is 1), and the engine is specified by `anEngine`. This is not supported in CLHEP.

Methods using the local engine to generate random values, by-passing the static generator are also provided.

```
double fire(double mean=1.0, double gamma=0.2);
```
Returns a double precision number from a Breit-Wigner distribution where the mean is `mean` (default is 1) and Γ is `gamma` (default is .2).

```
double fire(double mean, double gamma, double cut);
```
Returns a double precision number from a Breit-Wigner distribution where the mean is `mean` (default is 1) and Γ is `gamma` (default is .2).

```
double fireM2(double mean=1.0, double gamma=0.2);
```
Returns a double precision number from a Breit-Wigner distribution where the mean is positive `mean` (default is 1) and Γ is `gamma` (default is .2).

```
double fireM2(double mean, double gamma,
                                 double cut);
```
Returns a double precision number from a Breit-Wigner distribution where the mean is positive `mean` (default is 1) and Γ is `gamma` (default is .2).

```
void fireArray(const int size, double* vect,
 double mean=1.0, double gamma=0.2, double cut=1.0);
```
Fills an array `vect` of size `size` with double precision number from a Breit-Wigner distribution where the mean is `mean` (default is 1) and Γ is `gamma` (default is .2) and cut is `cut` (default is 1).

```
void fireArray(vector<double>& vec,
 double mean=1.0, double gamma=0.2, double cut=1.0);
```
Fills a vector `vec` with double precision number from a Breit-Wigner distribution where the mean is `mean` (default is 1) and Γ is `gamma` (default is .2) and cut is `cut` (default is 1). This is not supported in CLHEP.

Examples

```
#include <iostream.h>
#include "StGlobals.hh"
#include "Random.h"

// the random engines
#include "JamesRandom.h"
#include "RanluxEngine.h"

// the different distributions
#include "RandFlat.h"
#include "RandPoisson.h"
#include "RandExponential.h"
#include "RandGauss.h"
#include "RandBreitWigner.h"

int main()
{
    int i, jj;

    const StInt size = 5;
    const StInt numberOfNumbers = 5;

    // Generator must be given an engine:
    //   - HepJamesRandom used by default

    HepJamesRandom  engine1;
    RanluxEngine    engine2;

    //HepRandom quasiRandom(engine1);   // pass engine by reference
    //HepRandom quasiRandom(&engine1);  // pass engine by pointer

    //engine.showStatus();              // show status of engine

    long seed = 7;
    HepRandom quasiRandom;              // or quasiRandom(seed);

    StDouble quasiRandomNumber =
        quasiRandom.flat();
    PR(quasiRandomNumber);

    int    *vecI = new int[size];       // StInt    vecI[size]
    double *vec  = new double[size];    // StDouble vec[size];

    quasiRandom.flatArray(size,vec);

    cout << "Pseudo-Random numbers from a flat distribution." << endl;
    for(int ii=0; ii<size; ii++)
        cout << "i " << *(vec+ii) << endl;

    PR(quasiRandom.getTheSeed());
    HepJamesRandom jr;

    cout << "All these numbers are the same" << endl;
    for(ii=0; ii<size;ii++) {
        jr.saveStatus();
        PR(jr.flat());
        jr.restoreStatus();        // restoring status keeps engine same
```

```
    }

    cout << "Different distributions:" << endl;

    RandFlat        flatDistribution(engine1);
    RandGauss       gaussDistribution(engine2);
    RandExponential exponentialDistribution(engine1);
    RandPoisson     poissonDistribution(engine2);
    RandBreitWigner breitWignerDistribution(engine2);

    double mean  = 2;
    double width = 10;

    cout << "Numbers from a Flat Distribution:" << endl;
    for(jj=0; jj<numberOfNumbers; jj++) {
        StDouble flatNumber  =
            flatDistribution.fire(width,width+10);
        PR(flatNumber);
    }

    cout << "\nNumbers from a Gaussian Distribution:" << endl;
    for(jj=0; jj<numberOfNumbers; jj++) {
        StDouble gaussNumber =
            gaussDistribution.shoot();
        PR(gaussNumber);
    }

    cout << "\nNumbers from an Exponential Distribution:" << endl;
    for(jj=0; jj<numberOfNumbers; jj++) {
        StDouble exponentialNumber =
            exponentialDistribution.shoot(&engine2, mean);
        PR(exponentialNumber);
    }

    cout << "\nNumbers from a Poissonian Distribution:" << endl;
    for(jj=0; jj<numberOfNumbers; jj++) {
        StDouble poissonNumber =
            poissonDistribution.shoot();
        PR(poissonNumber);
    }

    cout << "\nAn Array of Numbers from a Breit-Wigner Distribution:" << endl;
    breitWignerDistribution.shootArray(size, vec);

    for(i=0; i<size; i++)
        cout << "(" << i << ") " << *(vec+i) << endl;

    return 0;
}
```

Programs Output:

```
quasiRandomNumber = 0.995292

Pseudo-Random numbers from a flat distribution.
i 0.363878
i 0.657671
i 0.0843759
i 0.129367
```

```
i 0.447258
quasiRandom.getTheSeed() = 19780503
All these numbers are the same:
jr.flat() = 0.995292
jr.flat() = 0.995292
jr.flat() = 0.995292
jr.flat() = 0.995292
jr.flat() = 0.995292

Different distributions:
Numbers from a Flat Distribution:
flatNumber = 19.9529
flatNumber = 13.6388
flatNumber = 16.5767
flatNumber = 10.8438
flatNumber = 11.2937

Numbers from a Gaussian Distribution:
gaussNumber = -0.641262
gaussNumber = 0.243269
gaussNumber = -0.151572
gaussNumber = -1.06513
gaussNumber = -0.498471

Numbers from an Exponential Distribution:
exponentialNumber = 1.6481
exponentialNumber = 0.0081336
exponentialNumber = 0.814686
exponentialNumber = 1.13064
exponentialNumber = 6.33225

Numbers from a Poissonian Distribution:
poissonNumber = 3
poissonNumber = 3
poissonNumber = 0
poissonNumber = 0
poissonNumber = 0

An Array of Numbers from a Breit-Wigner Distribution:
(0) 0.60626
(1) 1.17676
(2) 1.20535
(3) 0.986062
(4) 0.982262
```

A Helix Parametrization

The trajectory of a charged particle in a static uniform magnetic field with $\vec{B} = (0, 0, B_z)$ is a helix. In principle five parameters are needed to define such a helix. From the various possible parametrizations we describe here the version which is well suited for the geometry of a collider experiment and therefore used for the implementation of the StHelix class.

This parametrization describes the helix in Cartesian coordinates, where x, y and z are expressed as functions of the track length s.

$$x(s) = x_0 + \frac{1}{\kappa}[\cos(\Phi_0 + h\, s\, \kappa\, \cos\lambda) - \cos\Phi_0] \quad (1)$$

$$y(s) = y_0 + \frac{1}{\kappa}[\sin(\Phi_0 + h\, s\, \kappa\, \cos\lambda) - \sin\Phi_0] \quad (2)$$

$$z(s) = z_0 + s\, \sin\lambda \quad (3)$$

where here and in the following:

s is the path length along the helix

x_0, y_0, z_0 is the starting point at $s = s_0 = 0$

λ is the dip angle

κ is the curvature, i.e. $\kappa = 1/R$

B is the z component of the homogeneous magnetic field ($B = (0, 0, B_z)$)

q is charge of the particle in units of positron charge

h is the sense of rotation of the projected helix in the xy-plane,
i.e. $h = -\mathrm{sign}(qB) = \pm 1$

Φ_0 is the azimuth angle of the starting point (in cylindrical coordinates) with respect to the helix axis ($\Phi_0 = \Psi - h\pi/2$)

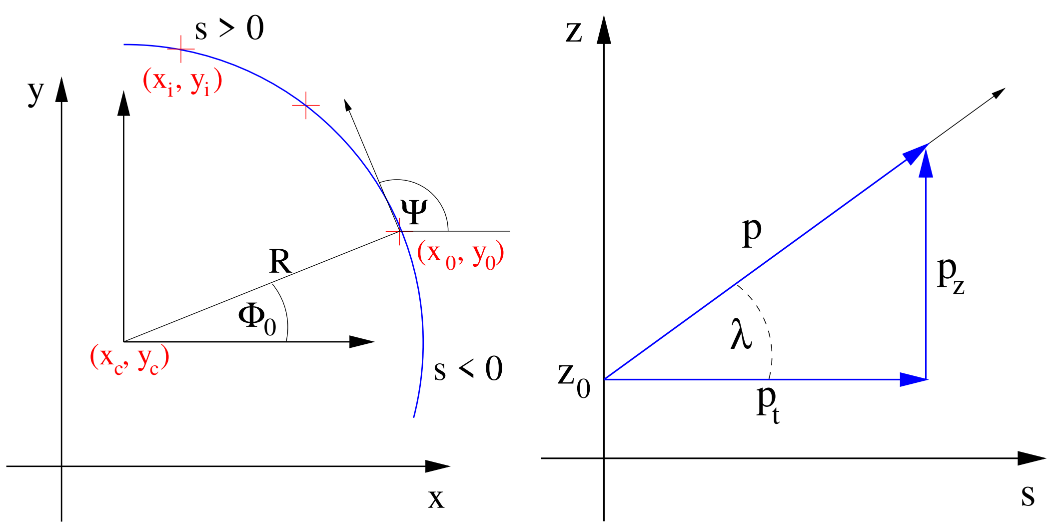
Ψ is the $\arctan(\mathrm{d}y/\mathrm{d}x)_{s=0}$, i.e. the azimuthal angle of the track direction at the starting point.

The meaning of the different parameters is visualized in Fig. 1.

A.1 Calculation of the particle momentum

The circle fit in the xy-plane gives the center of the fitted circle (x_c, y_c) and the curvature $\kappa = 1/R$ while the linear fit gives z_0 and $\tan\lambda$. The phase of the helix is defined as follows:

$$\Phi_0 = \arctan\left(\frac{y_1 - y_c}{x_1 - x_c}\right) \quad (4)$$



(a) Projection of a helix on the xy plane. The crosses mark possible data points.

(b) Projection of a helix on the sz plane.

Figure 1: Helix parametrization

The reference point (x_0, y_0) is then calculated as follows:

$$x_0 = x_c + \frac{\cos \Phi_0}{\kappa} \tag{5}$$

$$y_0 = y_c + \frac{\sin \Phi_0}{\kappa} \tag{6}$$

and the helix parameters can be evaluated as:

$$\Psi = \Phi_0 + h\pi/2 \tag{7}$$

$$p_\perp = c\, q\, B/\kappa \tag{8}$$

$$p_z = p_\perp \tan \lambda \tag{9}$$

$$p = \sqrt{p_\perp^2 + p_z^2} \tag{10}$$

where κ is the curvature in [m^{-1}], B the value of the magnetic field in [Tesla], c the speed of light in [m/ns] (≈ 0.3) and $p_\perp$ and p_z are the transverse and longitudinal momentum in [GeV/c].

A.2 Distant measure

The minimal squared distance M_i between a helix and a point i with position (x_i, y_i, z_i) is given by

$$\begin{aligned} M_i &= M_i^{(xy)} + M_i^{(z)} && (11) \\ M_i &= (x_i - x(s'))^2 + (y_i - y(s'))^2 + (z_i - z(s'))^2 && (12) \\ & && (13) \end{aligned}$$

In literature one finds the following approach to solve this problem analytically by neglecting $M_i^{(z)}$ in the derivatives.

$$\frac{\mathrm{d}M_i^{xy}}{\mathrm{d}s} = 0 \tag{14}$$

This formula can only serve to derive an approximation for the real distance. For large dip angles the errors become large depending also on the actual helix parameters. The advantage is that s' can be calculated analytically:

$$s' = \frac{1}{h\kappa\cos\lambda} \arctan\left(\frac{(y_i - y_0)\cos\Phi_0 - (x_i - x_0)\sin\Phi_0}{1/\kappa + (x_i - x_0)\cos\Phi_0 + (y_i - y_0)\sin\Phi_0}\right) \tag{15}$$

Note, that this formula can **not** be used to derive the distance of closest approach to a point. In order to derive the distance of closest approach the following equation has to be solved:

$$\frac{\mathrm{d}M_i}{\mathrm{d}s} = 0 \tag{16}$$

which can be written as

$$\begin{aligned} &2\left(x_i - x_0 - \frac{\cos(\Phi_0 + hs\kappa\cos\lambda) - \cos\Phi_0}{\kappa}\right)\sin(\Phi_0 + hs\kappa\cos\lambda)\, h\cos\lambda - \\ &2\left(y_i - y_0 - \frac{\sin(\Phi_0 + hs\kappa\cos\lambda) - \sin\Phi_0}{\kappa}\right)\cos(\Phi_0 + hs\kappa\cos\lambda)\, h\cos\lambda - \\ &2\,(z_i - z_0 - s\sin\lambda)\sin\lambda = 0 \end{aligned} \tag{17}$$

The root of eq. 17 can easily be found with the Newton or *regula falsi* method with s' from eq. 15 as starting value. For the Newton method the second derivative is needed as well.

$$\frac{\mathrm{d}^2 M_i}{\mathrm{d}s^2} = 0 \tag{18}$$

which is

$$\begin{aligned}
2 \quad & (\sin(\Phi_0 + hs\kappa\cos\lambda))^2\, h^2 \cos^2\lambda + \\
2 \quad & \left(x_i - x_0 - \frac{\cos(\Phi_0 + hs\kappa\cos\lambda) - \cos\Phi_0}{\kappa}\right) \\
& \cos(\Phi_0 + hs\kappa\cos\lambda) h^2 \kappa \cos^2\lambda + \\
2 \quad & (\cos(\Phi_0 + hs\kappa\cos\lambda))^2\, h^2 \cos^2\lambda + \\
2 \quad & \left(y_i - y_0 - \frac{\sin(\Phi_0 + hs\kappa\cos\lambda) - \sin\Phi_0}{\kappa}\right) \\
& \sin(\Phi_0 + hs\kappa\cos\lambda) h^2 \kappa \cos^2\lambda + \\
2 \quad & \sin^2\lambda = 0
\end{aligned} \tag{19}$$

A.3 Intersection with a plane

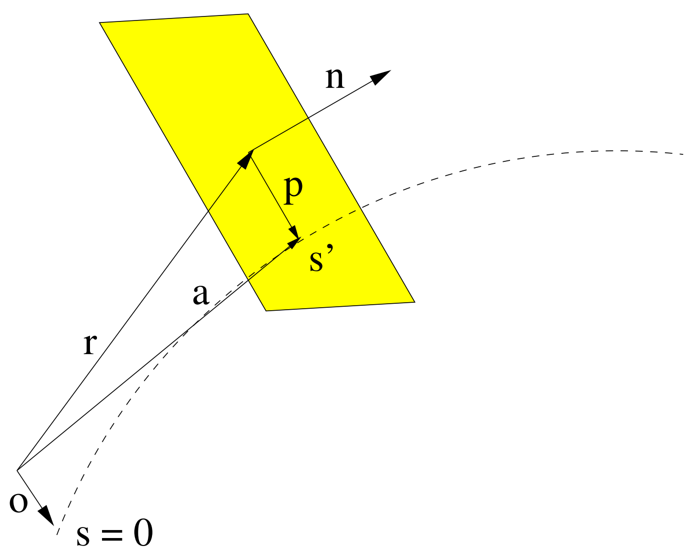


Figure 2: Intersection of a helix with a plane

Any plane can be described by its normal vector $\vec{n}$ (orientation) and an arbitrary point in this plane $\vec{r}$ (position). The vector $\vec{p}$ which describes the intersection point must fulfil:

$$\vec{p} \cdot \vec{n} = 0. \tag{20}$$

Hence:

$$(\vec{a} - \vec{r}) \cdot \vec{n} = 0. \tag{21}$$

where $\vec{a}$ is given by $\vec{a} = (x(s'), y(s'), z(s'))$ as described in eq. 1–3. In order to obtain the path length s' where the helix intersects with the plane the following equation has to be solved:

$$\begin{aligned} x(s)n_x + y(s)n_y + z(s)n_z - \vec{r}\cdot\vec{n} &= && (22) \\ A + n_x \cos S + n_y \sin S + \kappa n_z s \sin\lambda &= 0 && (23) \end{aligned}$$

where:

$$\begin{aligned} A &= \kappa(\vec{o}\cdot\vec{n} - \vec{r}\cdot\vec{n}) - n_x\cos\Phi_0 - n_y\sin\Phi_0 && (24) \\ S &= hs\kappa\cos\lambda + \Phi_0 && (25) \end{aligned}$$

The root of eq. 23 can now easily be determined by a suitable numerical method (Newton).

A.4 Limitations

The only non-numerical limitations of this parametrization are:

$$\begin{aligned} -\pi/2 < \lambda &< \pi/2 && (26) \\ \kappa &> 0 && (27) \end{aligned}$$

A.5 Case $\mathrm{B} = 0$

For the special case $B = 0$ the trajectory becomes a straight line, i.e. $\kappa = 0$ and $R = \infty$. Care must be taken in the numerical calculation of the parametrization because of the singularity in eq. 1 and 2. The correct form is:

$$\begin{aligned} x(s) &= x_0 - s\cos\lambda\sin\Phi_0 && (28) \\ y(s) &= y_0 + s\cos\lambda\cos\Phi_0 && (29) \\ z(s) &= z_0 + s\ \sin\lambda && (30) \end{aligned}$$

Eq. 15 then reads as:

$$s' = \frac{1}{\cos\lambda}\left[(y_i - y_0)\cos\Phi_0 - (x_i - x_0)\sin\Phi_0\right] \tag{31}$$

Eq. 17 can now be solved analytically;

$$\frac{\mathrm{d}M_i^{dca}}{\mathrm{d}s} = 0 \tag{32}$$

gives:

$$\begin{aligned} s^{dca} = \;& \cos\lambda\cos\Phi_0(y_i - y_0) - \\ & \cos\lambda\sin\Phi_0(x_i - x_0) + \\ & \sin\lambda(z_i - z_0) && (33) \end{aligned}$$

The same holds for the intersection of a helix with a plane where in case of zero curvature eq. 23 can be solved analytically.

$$s' = \frac{\vec{r} \cdot \vec{n} - \vec{o} \cdot \vec{n}}{-n_x \cos\lambda \sin\Phi_0 + n_y \cos\lambda \cos\Phi_0 + n_z \sin\lambda} \tag{34}$$

Index