STAR Offline Library Long Writeup

STAR C++ Class Library

User Guide and Reference Manual

Revision: 1.23

Date: 2006/08/15 21:42:22

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.
   - gcc/g++ 2.9.5.
2. Red Hat Linux 5.1 to Red Hat Linux 6.1:
   - gcc/g++ 2.91.66 or higher.
3. Solaris 2.4–2.6:
   - CC 4.2 and modified Object Space 2.0.2 Standard Library.
   - CC 5.0 or higher

Tests with Visual C++ 5.0 fail because of the lack of support of templated member functions and a broken overloading mechanism. Visual C++ 6.0 is not tested yet. Further tests on AIX and IRIX are not foreseen unless there is sufficient user demand.

3 Organization of the SCL

All documentation, code, and header files of the SCL is contained in a directory named StarClassLibrary. It contains 2 further sub-directories: ./examples and ./doc.

To summarize:

StarClassLibrary is the top directory,

- **./** contains the different header files (extension .h and .hh), and the referring source code (extension .cc),
- **./doc** contains all SCL documentations,. This directory is further divided up in ./doc/tex for the LaTeXguide (this document) and ./doc/html for a class browser. Both subdirectories contain makefiles in order to prepare the final document from the various sources.
- **./examples** contains small self-describing programs to test and demonstrate most the features of every classes. There is a GNUmakefile provided to compile and link the non-ROOT versions of the examples. Note that, except on Linux, you have to use `gmake` in order to process the makefile since it contains several GNU extension. Please read the examples/README file for further instructions.

4 Accessing the SCL

The SCL is part of the official STAR software distribution and is therefore present in the actual STAR software releases.

The *StarClassLibrary* is 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.bnl.gov/star/packages/repository`)
3. Check-out package into your current working directory:
   `cvs checkout StRoot/StarClassLibrary`

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 STAR environment 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: defined 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.

ST_SOLVE_TEMPLATES: force the instantiation of SCL templates when using `StTemplates.hh` (see reference section for more).

__ROOT__: This macro is automatically set if the header files are processed in a ROOT environment. If you want to use the SCL in a standalone mode you can either use the templated versions of the classes or undefine this flag in order to eliminate any ROOT dependencies.

6 Documentation

The SCL documentation consists of the *User Guide and Reference Manual* located in StarClassLibrary/doc/tex, i.e. this document. This directory contains a Makefile which allows to create either a PS version (`make`) or a PDF version (`make pdf`). The latter also includes bookmarks and allows to follow references in the text via hyperlinks. PDF is certainly the preferred format since it combines the advantages of HTML (links) and PostScript (printing) in one version. If you have trouble creating either the PS or the PDF version because of missing applications (LaTeX, dvips, dvipdf) you can get the current "`dev`" versions at the following URL: `http://www.star.bnl.gov/STARAFS/comp/root/special_docs.html`.

7 Known Problems

- The SCL should compile without any compiler warnings except on Sun platforms with CC4.2 due to shortcomings of the compiler.
- The Sun compiler verion CC4.2 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 10.18).
- In order to run programs built with the shared version of the SCL and standard library on Sun platforms with CC4.2, it is necessary to define an environment variable, `LD_LIBRARY_PATH`, to indicate the directory containing the shared library.
- If you are using an old `cfront` compiler who still uses template databases or repositories you might encounter unresolved symbols when linking with the SCL library. In order to avoid this problem one has to include `StTemplates.hh` once somewhere in the application. The only known platform where this is necessary is SUN with the CC4 compiler. Note, that one also has to set the `ST_SOLVE_TEMPLATES` flag in order to enable the template instantiation. By setting or omitting this macro in the referring Makefile one can selectively switch the forced instantiation on and off.
- The SCL uses heavily the Standard C++ Library and thus templates. In certain cases such as the CINT interpreter used by ROOT the use of the template SCL classes (or classes using templates) is not possible. Here one has to use the equivalent non-template version. Table 1 gives an overview of the available non-template/non-STL classes. Please note that the template versions should be used whenever possible.

All non-template classes listed in the table can be used in ROOT. The SCL library also contains the referring ROOT dictionary. Note that this increases the size of the classes by $\sim$ 12 byte depending on the platform.

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:

template class	**non-template class**
`StThreeVector<float>`	`StThreeVectorF`
`StThreeVector<double>`	`StThreeVectorD`
`StLorentzVector<float>`	`StLorentzVectorF`
`StLorentzVector<double>`	`StLorentzVectorD`
`StMatrix<float>`	`StMatrixF`
`StMatrix<double>`	`StMatrixD`
`StHelix`	`StHelixD`
`StPhysicalHelix`	`StPhysicalHelixD`

Table 1: Template classes and their non-template/non-STL equivalents.

- starsoft-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)

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 double`.

```
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
kaon_0_short_mass_c2 = 497.672  * MeV
pion_plus_mass_c2    = 139.5700 * MeV
pion_minus_mass_c2   = 139.5700 * MeV
lambda_mass_c2       = 1115.684 * MeV
antilambda_mass_c2   = 1115.684 * MeV
xi_minus_mass_c2     = 1321.32  * 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"`

Description Contains SCL-wide definitions. Since it also contains a few small template functions its use is restricted to environments which support templates. It also serves as an interface to CLHEP since it defines the relation between the basic STAR and CLHEP datatypes.

```
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"
```

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!

Note, that the base units are currently mapped to the ones in GEANT3 (cm, GeV, s). This is likely to change in future. Any changes to the base units, however, do not effect your code at all as long as you use `SystemOfUnits` consistently.

`namespace units {`

BASE UNITS:

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

```
millimeter  = 0.1
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 second, and Hertz)*

```
second      = 1
nanosecond  = 1.e-9*second
microsecond = 1.e-6*second
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 GeV)*

```
Megaelectronvolt = 1.e-3
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

millivolt     = 1.e-3*volt
```

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 centimeter)*
```
mm  = 0.1
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 second, and Hertz)*

```
s       = 1
ns      = 1.e-9*s
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 PhysicalConstants" << endl;
    cout << "--------------------------------------------------
             --------------------" << 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 obsolete file and should NOT be used**" << endl;
#endif

    return 0;
}
```

Program Output:

```
This program illustrates the use of SystemOfUnits
 and PhysicalConstants
-------------------------------------------------
 ---------------------
1 millimeter = 0.1
1 meter =      100
1 centimeter = 1
1 fermi =      1e-13
1 barn =       1e-24
1 degree =     0.0174533
1 second =     1
1 nanosecond = 1e-09
1 kHz =        1000
1 newton =    6.24151e+07
1 joule =     6.24151e+09
1 GeV =        1
1 Gigaelectronvolt = 1
The old CLHEP definitions are also supported:
---------------------------------------------
1 mm = 0.1
1 m  = 100
1 cm = 1
1 s  = 1
1 ns = 1e-09
```

9.4 Definition of Particles

9.4.1 Basic Concept

There are essentially two base classes which define the interface to STARs ”Definition of Particles”: `StParticleDefinition` and `StParticleTable`.

`StParticleDefinition` aggregates information to characterize particles property such as name, mass, spin and life time, while `StParticleTable` is a container class which holds a list of all available particle definitions and allows to query for particle definitions if only the name or the numeric identifier (GEANT3 or PDG) of a particle is known.

The StarClassLibrary provides the `StParticleDefinition` class to represent particles. Various particles such as electron, proton, and gamma have their own classes derived from `StParticleDefinition`. These concrete pre-defined particles, however, do not directly inherit from `StParticleDefinition` but indirectly through a layer of abstract classes which determins the particle ”type”. The following types are defined: `StMeson`, `StBaryon`, `StLepton`, `StIon` and `StBoson`. The UML diagrams for the underlying design are depicted in Fig. 9.1 in section 9.4.11.

9.4.2 Implementation of Particles

The idea and design of the `StParticleDefinition` class and all concrete particle classes derived from it is largely based on the design of the `G4ParticleDefinition` class from Geant4 (RD44). Although the code is in large parts different (modified or rewritten) and adapted to the STAR framework the basic idea stays the same.

An individual class is defined for each predefined particle. The object in each class is unique and defined as a static object (so-called singleton). Users can get pointers to such objects by using static methods in these classes.

9.4.3 StParticleDefinition

The `StParticleDefinition` class has ”read-only” properties which characterize the individual particle such as name, mass, charge, spin, and so on. These properties are set during initialization of each particle. Operators and methods to get these properties are listed below. Note, that the class itself is not a singleton but all derived concrete classes are.

Synopsis

```
#include "StParticleDefinition.hh"
```

Public Constructors

```
StParticleDefinition(const string  &  aName,
                     double           mass,
                     double           width,
                     double           charge,
                     int              iSpin,
                     int              iParity,
```

```
                    int              iConjugation,
                    int              iIsospin,
                    int              iIsospinZ,
                    int              gParity,
                    const string  &  pType,
                    int              lepton,
                    int              baryon,
                    int              encoding,
                    bool             stable,
                    double           lifetime);
```

Constructor for a particle with given parameters. For meaning and units of the arguments check the public member functions below.

Public Member Functions

`string   name() const;`
Name of the particle.

`double   mass() const;`
Mass of the particle, in units of equivalent energy [GeV/c^2].

`double   width() const;`
Decay width of the particle, usually the width of a Breit-Wigner function, assuming that you are near the mass center anyway (in units of equivalent energy, i.e. [GeV/c^2]).

`double   charge() const;`
Charge of the particle (in units of Coulomb). Divide by `eplus` from `PhysicalConstants.h` to get the charge in units of +e.

`double   spin() const;`
Total spin of the particle, in units of 1.

`int      iSpin() const;`
Total spin of the particle, also often denoted as capital J, in units of 1/2.

`int      iParity() const;`
Parity quantum number, in units of 1. If the parity is not defined for this particle, we will set this to 0.

`int      iConjugation() const;`
Charge conjugation quantum number in units of 1.

`double   isospin() const;`
Isospin in units of 1/2.

`double   isospin3() const;`
3rd-component of isospin in units of 1/2.

`int      iIsospin() const;`
Isospin in units of 1.

`int      iIsospin3() const;`
3rd-component of isospin in units of 1.

`int      iGParity() const;`
Value of the G-parity quantum number.

`string   type() const;`
General textual type description of the particle.

`int      leptonNumber() const;`
Lepton quantum number.

`int      baryonNumber() const;`
Baryon quantum number.

`int      pdgEncoding() const;`
Particle Data Group integer identifier of this particle

`int      antiPdgEncoding() const;`
Particle Data Group integer identifier of the corresponding anti-particle.

`bool     stable() const;`
True if particle is stable.

`double   lifeTime() const;`
Related to the decay width of the particle. The mean life time is given in seconds.

`StParticleTable* particleTable() const`
Returns pointer to the particle table.

Public Member Operators

`int operator==(const StParticleDefinition &) const;`
Test two particles for identity. Note, that the concrete particles are implemented as singletons. In this case also the values of the pointers are identical.

`int operator!=(const StParticleDefinition &) const;`
True if two particles are not identical.

Global Operators

```
ostream&
operator<<(ostream& os, const StParticleDefinition& p);
```
Prints the particle defined by *p* to output stream `os`.

9.4.4 Predefined Particles

The StarClassLibrary provide currently 77 predefined particles. They are listed in table 2-6 in section 9.4.10. Note that the PDG encoding is defined for elementary particles only and not for ions. GEANT3 IDs are listed for completeness only. They are not part of a particle definition. See section 9.4.7 on how to get a particle definition for a given GEANT3 ID.

Each concrete particle class is defined in one header file. The name of the header file is simply the name of the class plus the .hh extension. As mentioned above all particles are implemented as singletons, i.e. only one instance (a static object) exist. This minimizes the memory usage and ensures coherent definitions.

Each of the predefined classes has a member function `instance()` which returns a pointer to the only existing object. All constructors, including the copy constructor and the assignment operator are private to ensure that no second copy can be created. The following code demonstrates how to obtain a π^-.

```
#include "StPionMinus.hh"
```

```
void example1()
{
     StPionMinus *pim = StPionMinus::instance();
     cout << *pim << endl;
}
```

Please remember that no copy of an instance can be made. The following code does not compile.

```
StPionMinus pim = *(StPionMinus::instance()); // ERROR, assignment operator is private
```

As already mentioned above there is an additional layer of abstract classes between `StParticleDefinition` and the concrete classes (see also Fig. 9.1). This layer allows to distinguish particles according to their type (baryon, meson, etc). They do not add any data member or overwrite any functions; with other words there's no overhead involved. The idea behind this is best illustrated in the following example:

```
void fillHistos(StMeson*);    // version 1
void fillHistos(StBaryon*);   // version 2

void MyMCAnalysis(StParticleDefinition* particle)
{
    cout << "Got a " << particle->name() << endl;
    cout << "The pdg mass of this guy is: " << particle->mass() << endl;
    fillHisto(particle);   // calls version 1 if particle is a meson, 2 if its a baryon
}
```

9.4.5 Header Files

Each particle is defined in a separate header file. This results in an enormous amount of header files and include statements one has to deal with. To make life easier there is one header file which contains them all: `StParticleTypes.hh`. Note that you still have to include `StParticleTable.hh` if you want to use the particle table.

But remember that in many situations the header files of the pre-defined particles do not necessarily have to be included. Since the base class already defines the entire interface it is often sufficient to include `StParticleDefinition.hh` only. This reduces the dependencies of your code and therefore the compile time. Here are two example where you don't need the header of the specific particles:

```
#include "StParticleDefinition.hh"
#include "StParticleTable.hh"

void example2(StParticleDefinition* p)
{
     if (p == StParticleTable::instance()->findParticle("pi+")) {
           cout << "Got pi+" << endl;
           cout << "Mass is: " << p->mass() << endl;
     }
}

void example3(StParticleDefinition* p)
{
```

```
    if (p->baryonNumber() != 0 && p->type() == "meson") {
        cout << "Oops, this cannot be ..." << endl;
        cout << "Check this funny particle:" << endl;
        cout << *p << endl;
    }
}
```

9.4.6 How to Define a New Particle

This is easy. All you need is the data to define the particle: mass, lifetime, spin, etc. The safest (and fastest) way to proceed is to use one of the existing particle classes of the *same* type (ion, meson, baryon, lepton, boson) as a template. All you have to do is to replace all occurrences of the old class name with the new name in the header (.hh) and source (.cc) file. All what is left now is to replace the arguments passed to the constructor of the static data member with the new ones.

There's no need to register the new particle definition in the particle table. The constructor of the base class (`StParticleDefinition`) does this automatically for you.

9.4.7 The Particle Table

All particles are automatically registered in the particle table. The particle table is also a singleton, i.e., only one instance can be created. To obtain an pointer to the one and only instance one has to use the `instance()` member function or obtain the pointer through any predefined particle using its `StParticleDefinition::particleTable()` member function.
The class actually holds several maps which allow fast access to available particle definitions by name, PDG encoding, or GEANT3 Id. Remember that some particles have no corresponding GEANT3 Id and PDG encoding does not exist for ions. STAR extended the list of particle IDs in GSTAR to overcome the limited coverage of elementary particles in GEANT3. They are taken into account.

9.4.8 StParticleTable

Synopsis

```
#include "StParticleTable.hh"
typedef vector<StParticleDefinition*> StVecPtrParticleDefinition;
```

Public Constructors

None.
All constructors, including the copy constructor and the assignment operator are private to ensure that no second copy can be created.

Public Member Functions

`static StParticleTable* instance();`
Returns pointer to the only existing instance of self.

`static StParticleTable* particleTable();`
Returns pointer to the only existing instance of self. Same as `instance()`.

`unsigned int entries() const;`
Number of entries in the table, i.e. number of defined particles.

`unsigned int size() const;`
Number of entries in the table, i.e. number of defined particles. Same as `entries()`.

`bool contains(const string & pname) const;`
Returns true if particle with name *pname* is defined.

`bool contains(int pdgId) const;`
Returns true if particle with PDG encoding *pdgId* is defined.

`bool containsGeantId(int gId) const;`
Returns true if particle with GEANT3 ID *gId* is defined.

`StParticleDefinition* findParticle(const string& pname) const;`
Returns pointer to particle with name *pname* or a null pointer if the particle is not defined.

`StParticleDefinition* findParticle(int pdgId)  const;`
Returns pointer to particle with PDG encoding *pdgId* or a null pointer if the particle is not defined.

`StParticleDefinition* findParticleByGeantId(int gId) const;`
Returns pointer to particle with GEANT3 ID encoding *gId* or a null pointer if the particle is not defined.

`void insert(StParticleDefinition* part);`
Insert new particle definition *part* ”by hand”. Note, that this is not necessary since each defined particle is automatically registered (using this member function).

`void erase(StParticleDefinition* part);`
Removes given particle definition *part* from the table.

`void dump(ostream& os = cout);`
Dumps (prints) all particles defined in the table to ostream *os*. Note, that *os* defaults to cout (stdout).

`StVecPtrParticleDefinition allParticles() const;`
Returns a vector of pointers to all particle definitions stored in the table.

Warnings The order of the particles in the table is arbitrary and system dependent. Adding new particles might alter the order as well. Do not write code that relies on indices into a vector obtained from `allParticles()`

9.4.9 Examples

What follows is a list of simple programs which should help to get familiar with the particle definition classes. In some cases the shown output of the program is shortened (marked by [...]) to save space.

Example1 – Program Code:

```
#include <iostream.h>
#include "StParticleTable.hh"
#include "StParticleTypes.hh"

int main()
```

```
{
    //
    //  Write all particle definitions to stdout
    //
    StParticleTable::instance()->dump();
    return 0;
}
```

Programs Output:

```
Particle Name :          B+
PDG particle code :      521
Mass [GeV/c2] :          5.2789
Width [GeV/c2] :         0
Lifetime [nsec] :        0.00162
Charge [e] :             1
Spin :                   0/2
Parity :                 -1
Charge conjugation :     0
Isospin : (I,Iz):        (1/2 , 1/2 )
GParity :                0
Lepton number :          0
Baryon number :          0
Particle type :          meson
Is stable :              no

[ ... ]

Particle Name :          xi_c0
PDG particle code :      4132
Mass [GeV/c2] :          2.4703
Width [GeV/c2] :         0
Lifetime [nsec] :        9.8e-05
Charge [e] :             0
Spin :                   1/2
Parity :                 1
Charge conjugation :     0
Isospin : (I,Iz):        (1/2 , -1/2 )
GParity :                0
Lepton number :          0
Baryon number :          1
Particle type :          baryon
Is stable :              no
```

Example2 – Program Code:

```
#include <iostream.h>
#include "StParticleTable.hh"
#include "StParticleTypes.hh"

int main()
{
     //
     //  Print all particles without PDG encoding
     //
     cout << "The following particles have no PDG encoding:" << endl;
```

```
     StParticleTable &table = *(StParticleTable::instance());
     StVecPtrParticleDefinition vec = table.allParticles();
     for (int i=0; i<vec.size(); i++) {
          if (vec[i]->pdgEncoding() == 0)
          cout << vec[i]->name().c_str() << endl;
     }
     return 0;
}
```

Programs Output:

```
The following particles have no PDG encoding:
He3
alpha
deuteron
opticalphoton
triton
```

Example3 – Program Code:

```
#include <iostream.h>
#include "StParticleTable.hh"
#include "StParticleTypes.hh"

int main()
{
     //
     //  Check consistency of GEANT3 lookup table
     //
     StParticleTable *table = StParticleTable::instance();
     StParticleDefinition *p1, *p2;
     for (int i=0; i<100; i++)
          p1 = table->findParticleByGeantId(14);
          if (p1) {
              p2 = table->findParticle(p1->name());
              if (*p1 != *p2)
                    cerr << "WARNING: inconsistency in lookup table" << endl;
          }
     }
     return 0;
}
```

Programs Output:
None

Example4 – Program Code:

```
#include <iostream.h>
#include <iomanip.h>
#include <typeinfo>
#include "StParticleTable.hh"
#include "StParticleTypes.hh"

int main()
    //
    //  This is a small program which I used to create the
```

```
    //  the TeX tables of all defined particles in the next
    //  section. It saves a lot of typing and prevents typos.
    //  In order to get the class name I had to use
    //  the typeid() function which is only available
    //  if RTTI is switched on.
    //
    StParticleTable &table = *(StParticleTable::instance());
    StVecPtrParticleDefinition vec = table.allParticles();

    for (int i=0; i<vec.size(); i++) {
        cout << setw(14) << vec[i]->name() << "\t& ";
        if (vec[i]->pdgEncoding() == 0)
            cout << "N/A" << "\t& ";
        else
            cout << vec[i]->pdgEncoding() << "\t& ";
        int k = 0;
        for (int j=0; j<100; j++) {
            if (vec[i] == table.findParticleByGeantId(j)) {
                k = j;
                break;
            }
        }
        if (k != 0)
            cout << j << "\t& ";
        else
            cout << "N/A" << "\t& ";

        cout << setw(20) << typeid(*vec[i]).name() << "\t& ";

        if (dynamic_cast<StMeson*>(vec[i]))
            cout << "StMeson " << "\t";
        else if (dynamic_cast<StBoson*>(vec[i]))
            cout << "StBoson " << "\t";
        else if (dynamic_cast<StLepton*>(vec[i]))
            cout << "StLepton" << "\t";
        else if (dynamic_cast<StBaryon*>(vec[i]))
            cout << "StBaryon" << "\t";
        else if (dynamic_cast<StIon*>(vec[i]))
            cout << "StIon   " << "\t";
        else {
            cout << "ERROR: unknown base class" << endl;
            return 2;
        }
        cout << " \\\\ \\hline" << endl;
    }
    return 0;
}
```

Programs Output:

```
      B+  & 521   & N/A   &        StBMesonPlus  & StMeson        \\ \hline
      B-  & -521  & N/A   &       StBMesonMinus  & StMeson        \\ \hline
      B0  & 511   & N/A   &        StBMesonZero  & StMeson        \\ \hline
     Bs0  & 531   & N/A   &       StBsMesonZero  & StMeson        \\ \hline
      D+  & 411   & 35    &        StDMesonPlus  & StMeson        \\ \hline
      D-  & -411  & 36    &       StDMesonMinus  & StMeson        \\ \hline
      D0  & 421   & 37    &        StDMesonZero  & StMeson        \\ \hline
```

```
Ds+   & 431    & 39    &        StDsMesonPlus   & StMeson         \\ \hline
Ds-   & -431   & 40    &       StDsMesonMinus   & StMeson         \\ \hline
[ ... ]
```

9.4.10 List of Predefined Particle Definitions

Tables 2 to 6 list[3] all predefined particle definitions. The meaning of the different columns are:

Particle Name: An unique, descriptive name assigned to each particle. This name can be used in `StParticleTable` to obtain a pointer to the actual particle definition.

PDG Encoding: The number assigned to this particle by the Particle Data Group.

GEANT3 Ids: The number GEANT3 uses to reference this type of particle. The table also includes IDs defined for GSTAR (STAR specific).

Class Name: Name of the class. Use `St`*ClassName*`::instance()` to get a pointer to the referring particle definition.

Derived From: Name of the base class. This defines also the type of the particle.

Particle Name	PDG Encoding	GEANT3 Ids	Class Name	Derived From
anti_nu_e	-12	N/A	StAntiNeutrinoE	StLepton
anti_nu_mu	-14	N/A	StAntiNeutrinoMu	StLepton
anti_nu_tau	-16	N/A	StAntiNeutrinoTau	StLepton
e+	-11	2	StPositron	StLepton
e-	11	3	StElectron	StLepton
mu+	-13	5	StMuonPlus	StLepton
mu-	13	6	StMuonMinus	StLepton
nu_e	12	4	StNeutrinoE	StLepton
nu_mu	14	N/A	StNeutrinoMu	StLepton
nu_tau	16	N/A	StNeutrinoTau	StLepton
tau+	-15	33	StTauPlus	StLepton
tau-	15	34	StTauMinus	StLepton

Table 2: List of defined leptons

[3] See example 3 in section 9.4.9 on how this list was created.

Particle Name	PDG Encoding	GEANT3 Ids	Class Name	Derived From
alpha	N/A	47	StAlpha	StIon
deuteron	N/A	45	StDeuteron	StIon
He3	N/A	49	StHe3	StIon
triton	N/A	46	StTriton	StIon

Table 3: List of defined ions

Particle Name	PDG Encoding	GEANT3 Ids	Class Name	Derived From
gamma	22	1	StGamma	StBoson
opticalphoton	N/A	50	StOpticalPhoton	StBoson

Table 4: List of defined bosons

Particle Name	PDG Encoding	GEANT3 Ids	Class Name	Derived From
anti_B0	-511	N/A	StAntiBMesonZero	StMeson
anti_Bs0	-531	N/A	StAntiBsMesonZero	StMeson
anti_D0	-421	38	StAntiDMesonZero	StMeson
anti_kaon0	-311	156	StAntiKaonZero	StMeson
B+	521	N/A	StBMesonPlus	StMeson
B-	-521	N/A	StBMesonMinus	StMeson
B0	511	N/A	StBMesonZero	StMeson
Bs0	531	N/A	StBsMesonZero	StMeson
D+	411	35	StDMesonPlus	StMeson
D-	-411	36	StDMesonMinus	StMeson
D0	421	37	StDMesonZero	StMeson
Ds+	431	39	StDsMesonPlus	StMeson
Ds-	-431	40	StDsMesonMinus	StMeson
eta	221	17	StEta	StMeson
eta_prime	331	N/A	StEtaPrime	StMeson
J/psi	443	N/A	StJPsi	StMeson
kaon+	321	11	StKaonPlus	StMeson
kaon-	-321	12	StKaonMinus	StMeson
kaon0	311	155	StKaonZero	StMeson
kaon0L	130	10	StKaonZeroLong	StMeson
kaon0S	310	16	StKaonZeroShort	StMeson
omega	223	150	StOmegaMeson	StMeson
phi	333	151	StPhi	StMeson
pi+	211	8	StPionPlus	StMeson
pi-	-211	9	StPionMinus	StMeson
pi0	111	7	StPionZero	StMeson
rho+	213	153	StRhoPlus	StMeson
rho-	-213	154	StRhoMinus	StMeson
rho0	113	152	StRhoZero	StMeson

Table 5: List of defined mesons

Particle Name	PDG Encoding	GEANT3 Ids	Class Name	Derived From
anti_lambda	-3122	26	StAntiLambda	StBaryon
anti_lambda_c+	-4122	N/A	StAntiLambdacPlus	StBaryon
anti_neutron	-2112	25	StAntiNeutron	StBaryon
anti_omega-	-3334	32	StAntiOmegaMinus	StBaryon
anti_omega_c0	-4332	N/A	StAntiOmegacZero	StBaryon
anti_proton	-2212	15	StAntiProton	StBaryon
anti_sigma+	-3222	27	StAntiSigmaPlus	StBaryon
anti_sigma-	-3112	29	StAntiSigmaMinus	StBaryon
anti_sigma0	-3212	28	StAntiSigmaZero	StBaryon
anti_sigma_c+	-4212	N/A	StAntiSigmacPlus	StBaryon
anti_sigma_c++	-4222	N/A	StAntiSigmacPlusPlus	StBaryon
anti_sigma_c0	-4112	N/A	StAntiSigmacZero	StBaryon
anti_xi-	-3312	31	StAntiXiMinus	StBaryon
anti_xi0	-3322	30	StAntiXiZero	StBaryon
anti_xi_c+	-4232	N/A	StAntiXicPlus	StBaryon
anti_xi_c0	-4132	N/A	StAntiXicZero	StBaryon
lambda	3122	18	StLambda	StBaryon
lambda_c+	4122	41	StLambdacPlus	StBaryon
neutron	2112	13	StNeutron	StBaryon
omega-	3334	24	StOmegaMinus	StBaryon
omega_c0	4332	N/A	StOmegacZero	StBaryon
proton	2212	14	StProton	StBaryon
sigma+	3222	19	StSigmaPlus	StBaryon
sigma-	3112	21	StSigmaMinus	StBaryon
sigma0	3212	20	StSigmaZero	StBaryon
sigma_c+	4212	N/A	StSigmacPlus	StBaryon
sigma_c++	4222	N/A	StSigmacPlusPlus	StBaryon
sigma_c0	4112	N/A	StSigmacZero	StBaryon
xi-	3312	23	StXiMinus	StBaryon
xi0	3322	22	StXiZero	StBaryon
xi_c+	4232	N/A	StXicPlus	StBaryon
xi_c0	4132	N/A	StXicZero	StBaryon

Table 6: List of defined baryons

9.4.11 UML Diagrams

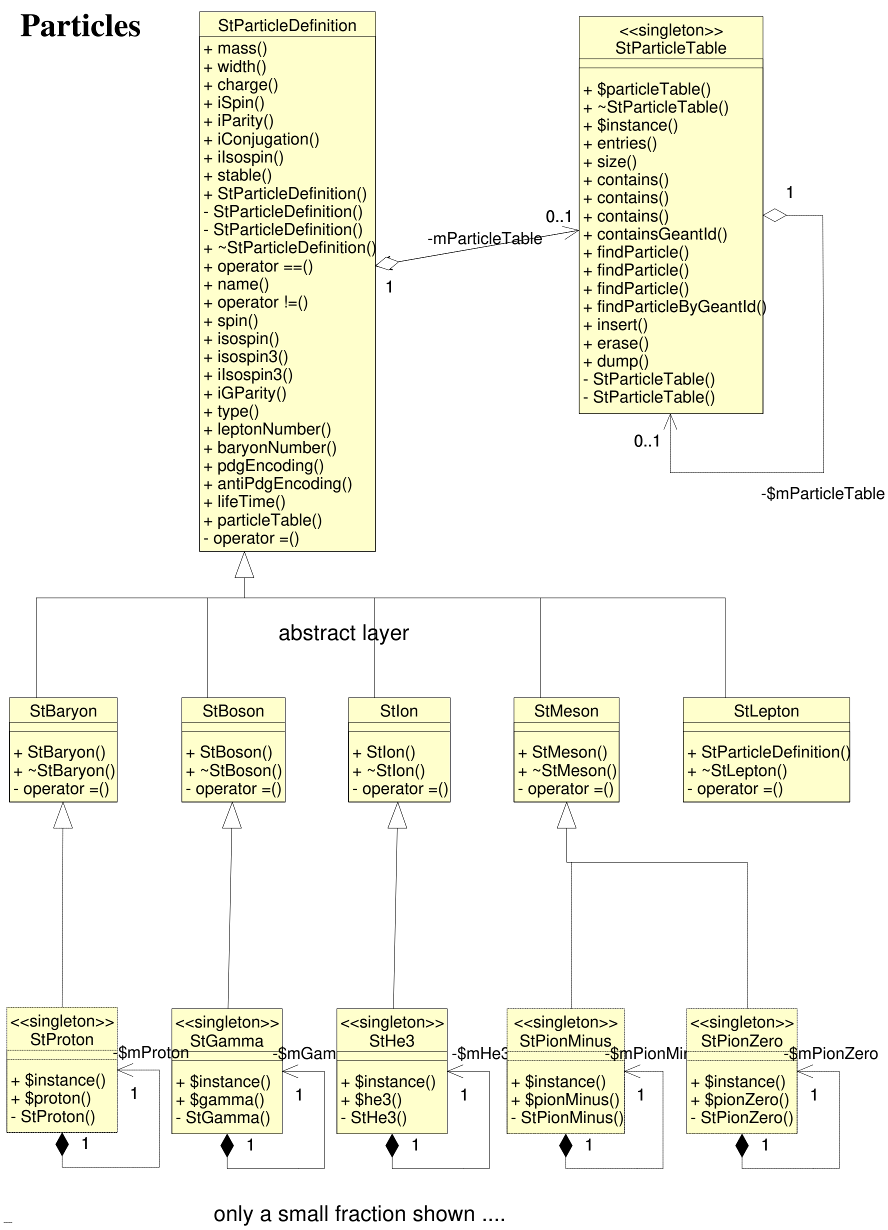


Figure 9.1: UML diagram for particle definitions

10 Class Reference

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

10.1 StAngle

Summary The StAngle class represents an angle in radians and provide all essential operation in an unambigious way.

Synopsis

```
#include "StAngle.hh"
class StAngle;
```

Description This class represents an angle in units of radians. It behaves pretty much like a `double` but ensures that its value always stay in the range $[-\pi, \pi]$. Essentially all operations as multiplication, addition, subtraction, or scaling are overloaded. This class is especially useful if you deal with azimuthal symmetric problems since it handles differences between angles correctly and avoids the usual problems related to the $0 \rightarrow 2\pi$ transition. This saves a lot of checks within the code as compared to using simple `floats` or `doubles` to represent angles.

However, this class should not be used when speed is crucial since it carries quite some overhead. Use it when reliability and safety are major issues.

Public Constructors

`StAngle();`
Default constructor.

`StAngle(const StAngle&);`
Copy constructor.

`StAngle(double alpha);`
Create angle with `alpha` radians. `alpha` is automatically adjusted to fall in the range $[-\pi, \pi]$.

Public Member Functions

`double degree();`
Returns the angle in degree. The return value is always in the range $[0° - 360°]$. Please note that this is the only safe method to obtain the angle in degree. See "pitfalls" below.

Public Member Operators

`operator double() const;`
Transforms a `StAngle` into a `double` (type cast operator). Since this operator is defined one can use an StAngle everywhere where a `double` or `float` can be used. Essentially one can ignore this operator. It works for you in the background.

`StAngle operator= (double angle);`
Assignement operator. Initiate self with an angle (in radian) keeping self in the range $[-\pi, \pi]$.

`StAngle operator+= (StAngle angle);`
Add another `angle` to self, keeping self in the range $[-\pi, \pi]$

`StAngle operator-= (StAngle angle);`
Subtract `angle` from self. The difference will always be the smallest possible difference. To give an example (for simplicity in degrees): $380° - 10°$ is $10°$ not $370°$.

```
StAngle operator*= (double angle);
StAngle operator/= (double angle);
```
Multiply (divide) self by `angle`. Self stays in the range $[-\pi, \pi]$.

```
int operator== (const StAngle& angle);
```
Comparison operator. Note that x equals $x + n2\pi$.

Global Functions

```
StAngle average(StAngle alpha, StAngle beta);
```
Returns the average of two angles `alpha` and `beta`. To give an example (for simplicity in degrees): the average of 380° and 10° is 15° not 195°.

Global Operators

```
StAngle operator+ (StAngle alpha, StAngle beta);
StAngle operator- (StAngle alpha, StAngle beta);
StAngle operator* (StAngle alpha, double c);
StAngle operator/ (StAngle alpha, double c);
```
Operators to allow to add, subtract two angles as well as scale and divide angles by a scalar. The results always will be a `StAngle` with a value in the range $[-\pi, \pi]$.

Pitfalls

`StAngle` will always try to stay within its range ($[-\pi, \pi]$). This way you can scale angles. The pitfall here is if you for example want to calculate its value in degree without using the `degree()` method.

```
StAngle a = 2.7;
cout << a*180/pi << endl;  // doesn't work
```

will **not** work. The expression `a*180/pi` consist of a StAngle times a double which matches the overloaded operator `StAngle operator* (StAngle alpha, double c)` that returns type `StAngle` which of course never gets larger than π.

The correct way is: `StAngle a = 2.7;`
`cout << a*degree() << endl;  // works`
or `cout << static_cast<double>(a)*180/pi << endl;  // also works`

Whenever you need the result of the multiplication of a `double` and a `StAngle` to be of type `double` you have to use the type cast.

10.2 StFastCircleFitter

Summary Fast circle fitting functor.

Synopsis

```
#include "StFastCircleFitter.hh"
class StFastCircleFitter;
```

Description Fast fitting functor using a iterational linear regression method (ILRM). Reference: *N.Chernov, G.A.Ososkov, Computer Physics Communication 33 (1984) 329-333.*

The functor fits the origin (x, y) and the radius. No stored point is excluded from the fit, i.e. all points are used.

Return codes:

0 fit ok

1-4 error occured, no results

`StFastCircleFitter::fit()` returns true only if rc = 0

Public Constructors

`StFastCircleFitter();`
Default constructor.

Public Member Functions

`void addPoint(double x, double y);`
Add point with coordinates `x`, `y`.

`void clear();`
Reset all fit results. Deletes all points stored so far.

`unsigned int numberOfPoints() const;`
Returns number of points stored and used in the fit.

`bool fit();`
Performs the fit.

`double radius() const;`
Returns fitted radius.

`double xcenter() const;`
Returns x of fitted center.

`double ycenter() const;`
Returns y of fitted center.

`double variance() const;`
Returns variance estimate. See reference for more on the validity of the variance.

`int rc() const;`
Return code of fit. Returns 0 in case of success else values between 1–4.

Examples

```
#include "StFastCircleFitter.hh"
#include <iostream.h>
#include <cstdlib>

int main()
```

```
{
    StFastCircleFitter fitter;

    const unsigned int nPoints = 42;
    const double x0 = 10.23;
    const double y0 = 0.018;
    const double r  = 19.91;
    double phi;
    int i;

    cout << "Input:    x0 = " << x0 << endl;
    cout << "          y0 = " << y0 << endl;
    cout << "          r  = " << r << endl;
    cout << "          n  = " << nPoints << endl;

    for (i=0; i<nPoints; i++) {
        phi = drand48();
        fitter.addPoint(r*cos(phi)+x0, r*sin(phi)+y0);
    }

    fitter.fit();
    cout << "Fit 1:    x0 = " << fitter.xcenter() << endl;
    cout << "          y0 = " << fitter.ycenter() << endl;
    cout << "          r  = " << fitter.radius() << endl;
    cout << "          n  = " << fitter.numberOfPoints() << endl;

    fitter.clear();

    for (i=0; i<nPoints; i++) {
        phi = drand48();
        fitter.addPoint(r*cos(phi)+x0, r*sin(phi)+y0);
    }

    fitter.fit();
    cout << "Fit 2:    x0 = " << fitter.xcenter() << endl;
    cout << "          y0 = " << fitter.ycenter() << endl;
    cout << "          r  = " << fitter.radius() << endl;
    cout << "          n  = " << fitter.numberOfPoints() << endl;

    return 0;
}
```

Programs Output:

```
Input:    x0 = 10.23
          y0 = 0.018
          r  = 19.91
          n  = 42
Fit 1:    x0 = 10.23
          y0 = 0.018
          r  = 19.91
          n  = 42
Fit 2:    x0 = 10.23
          y0 = 0.018
          r  = 19.91
          n  = 42
```

10.3 StGetConfigValue

Summary

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

Synopsis

`#include "StGetConfigValue.hh"`

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
```

10.4 StHbook

Summary

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

Synopsis

`#include "StHbook.hh"`

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`. 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.

2-dimensional Histograms

```
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 $y1 < y < y2$.

[4] http://wwwcn1.cern.ch/asd/index.html

```
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;

Row Wise N-tuples
```
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

1-dimensional Histograms
```
StHbookHisto& operator= (const StHbookHisto&);
```
Assignment operator.

2-dimensional Histograms
```
StHbookHisto2& operator= (const StHbookHisto2&);
```
Assignment operator.

Row Wise N-tuples
```
operator<<
```
Assignment of tags to the ntuple.

Public Member Functions

Hbook Files
```
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*.

1-dimensional histograms
```
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*.

<u>`Row Wise N-Tuples`</u>

`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
```

10.5 StHelix

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

Synopsis

```
#include "StHelix.hh"
class StHelix;
```

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 parametrization is taken from Bock[5]. 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 parametrization. The interface provides access functions for all parameters that define the helix.
Important: For $B = 0$ the sense of rotation is ill defined. All what matters is that $\Phi_0 = \Psi - h\pi/2$ is done correctly, i.e. with the same arbitrary h as passed to the constructor as last argument.

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,
        const StThreeVector<double>& o, int h=-1);
```

Constructs a helix with curvature=c, dip angle=dip, phase=phase, origin=o, and charge=-h. Note that for $B = 0$ (c = 0) the sense of rotation is ill defined. All what matters is that $\Phi_0 = \Psi - h\pi/2$ is done correctly, i.e. with the same arbitrary h as passed to the constructor as last argument.

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

`StHelix(const StHelix& hlx)`
Copy Constructor. Constructs a helix with the content of `hlx`.

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 p in space, returns the path length of the helix at the distance of closest approach to the helix.

`double pathLength(double x, double y) const;`
Given a position in the xy-plane, returns the path length of the helix at the distance of closest approach to the helix in the plane. Note, that the result differs from that of the previous method which returns the distance of closest approach in 3 dimensions. Only for helices with zero dip angle both methods will return the same values.

```
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()`.

`pair<double, double> pathLengths(const StHelix& h) const;`
Returns the path lengths at the distance-of-closest-approach between self and h, i.e., the DCA between two helices. The first element of the pair is the referring path length of self and the second element the path length of h. The method is fast and robust but execution time will increase slightly for two helices with very different dip angles and/or for large distances. The following pseudo-code explains how to get the actual dca:

```
// given helix 'myhelix' and 'otherHelix'
pair<double, double>  s = myhelix.pathLengths(otherHelix);
double dca = abs(myhelix.at(s.first)-otherHelix.at(s.second));
```

`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)`.
Note that this method returns the distance of closest approach in 3 dimensions, i.e. not in the plane. The following pseudo-code explains how to get the distance in the plane:

```
// given helix 'myhelix' and a point p
double s = myhelix.pathLength(p.x(), p.y()); // s at dca in xy-plane
double b = abs(p - myhelix.at(s));           // myhelix.distance(p) for 3d
```

`bool valid() const;`
checks for a valid parametrization. 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
```

10.6 StHelixD

Summary StHelixD is similar to `StHelix` (see 10.5) but does not contain or use templates nor does it make any use of the Standard C++ library.

Synopsis

```
#include "StHelixD.hh"
StHelixD;
```

Description The member functions, operators and non-member functions are identical to those of StHelix with the exception that whenever `StHelix` returns a `StThreeVector<double>` a `StThreeVectorD` is returned. The STL structure `pair<double, double>` used in this context is replaced by a similar but non-template structure `pairD`. The templated version should be preferred where possible.

Related Classes The equivalent of StPhysicalHelix is the class StPhysicalHelixD which is derived from StHelixD. StHelixD inherits from TObject if the SCL was compiled with the __ROOT__ flag set.

Persistence Within the ROOT framework.

10.7 StLorentzVector

Summary

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

Synopsis

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

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 the significant difference that it is a templated vector. 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 uses **StThreeVector** as data member.

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);
T operator() (size_t i) const;
```

Returns components by index. The first version can also be used as lvalue. 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);
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.

```
template<class X>;
void setVect(StThreeVector<X> 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.

```
template<class X>
StLorentzVector<T> boost(const StLorentzVector<X> &pfr)
```
Returns boosted Lorentz vector. Here `self` is the CM 4-momentum in the moving frame and `pfr` is the 4-momentum of the moving frame in the lab. See Example 2 for a better understanding. 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.

Example 1

```
#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
```

Example 2

```
#include <iostream.h>
#include <cmath>
#include "StGlobals.hh"
#include "StLorentzVector.hh"
#include "SystemOfUnits.h"
#include "PhysicalConstants.h"
#include "Randomize.h"

int main()
{
    //
    //  Generate 2-body decay: phi->e+e-
    //  This program generates a phi meson with random momentum
    //  px, py, pz and lets it decay into two electrons. Their
    //  momenta are then boosted into the lab frame.
    //

    //
    //  Setup the random generator
    //
    HepJamesRandom engine;
    RandFlat rflat(engine);

    //
    //  Create parent particle, here a phi(1020)
```

```
//  with momentum random momenta 0-1 GeV/c
//
StThreeVector<double> p(rflat.shoot()*GeV,
                        rflat.shoot()*GeV,
                        rflat.shoot()*GeV);
StLorentzVector<double> parent(p, p.massHypothesis(1020*MeV));

cout << "parent particle: " << endl;
cout << "\t4-momentum: " << parent << endl;
cout << "\tinvariant mass: " << abs(parent) << endl;

//
//  Let the phi decay into two electrons.
//  The easiest way to do this is in the CM of the parent.
//
double mass1 = electron_mass_c2;
double mass2 = electron_mass_c2;
double massParent  = abs(parent);
double E1  = (massParent*massParent + mass1*mass1 - mass2*mass2)/
             (2.*massParent);
double E2  = massParent - E1;
double p1  = sqrt((E1 + mass1)*(E1 - mass1));
double p2  = sqrt((massParent*massParent-(mass1+mass2)*
                  (mass1+mass2))*
             (massParent*massParent-(mass1-mass2)*
             (mass1-mass2)))/(2.*massParent);

//
//  Orientation in decaying particle rest frame
//
double costheta = 2*rflat.shoot() - 1;
double sintheta = sqrt((1 + costheta)*(1 - costheta));
double phi      = 2*pi*rflat.shoot();

//
//  Create daughters
//
StThreeVector<double>   momentum(p1*sintheta*cos(phi),
                                 p1*sintheta*sin(phi),
                                 p1*costheta);
StLorentzVector<double> daughter1(E1, momentum);
StLorentzVector<double> daughter2(E2, -momentum);

cout << "decay particles in CM frame of parent: " << endl;
cout << "daughter1: " << endl;
cout << "\t4-momentum: " << daughter1 << endl;
cout << "\tinvariant mass: " << abs(daughter1) << endl;
cout << "daughter2: " << endl;
cout << "\t4-momentum: " << daughter2 << endl;
cout << "\tinvariant mass: " << abs(daughter2) << endl;

//
//  Boost secondary particles into the lab
//
daughter1 = daughter1.boost(parent);
daughter2 = daughter2.boost(parent);
```

```
    cout << "decay particles in lab frame: " << endl;
    cout << "daughter1: " << endl;
    cout << "\t4-momentum: " << daughter1 << endl;
    cout << "\tinvariant mass: " << abs(daughter1) << endl;
    cout << "daughter2: " << endl;
    cout << "\t4-momentum: " << daughter2 << endl;
    cout << "\tinvariant mass: " << abs(daughter2) << endl;

    //
    //  Cross-check: reconstruct parent from daughters
    //
    StLorentzVector<double> check = daughter1+daughter2;
    cout << "cross-check: reconstructed parent" << endl;
    cout << "\t4-momentum: " << check << endl;
    cout << "\tinvariant mass: " << abs(check) << endl;

    return 0;
}
```

Programs Output:

```
parent particle:
        4-momentum: ((995.292, 363.878, 657.671),1611.19)
        invariant mass: 1020
decay particles in CM frame of parent:
daughter1:
        4-momentum: ((194.897, 205.896, -423.936),510)
        invariant mass: 0.510999
daughter2:
        4-momentum: ((-194.897, -205.896, 423.936),510)
        invariant mass: 0.510999
decay particles in lab frame:
daughter1:
        4-momentum: ((688.868, 386.491, -97.5296),795.881)
        invariant mass: 0.510999
daughter2:
        4-momentum: ((306.425, -22.6126, 755.2),815.313)
        invariant mass: 0.510999
cross-check: reconstructed parent
        4-momentum: ((995.292, 363.878, 657.671),1611.19)
        invariant mass: 1020
```

10.8 StLorentzVectorD

Summary StLorentzVectorD is a non-template version of StLorentzVector<double> (see 10.7). The code does not contain templates nor does it make use of the Standard C++ library. All data member are of type double.

Synopsis

```
#include "StLorentzVectorD.hh"
StLorentzVectorD;
```

Description The member functions, operators and non-member functions are identical to those in StLorentzVector<T> but might be slightly slower in execution speed since the inline mechanism cannot be used as extensive as for the template version. Operations can be mixed with the non-template single precision version StLorentzVectorF and with StThreeVectorF, StThreeVectorD, StMatrixF, and StMatrixD but not with any instance of StLorentzVector<T>, StThreeVector<T> or StMatrix<T>.. StLorentzVector<T> should be preferred where possible.

Related Classes StLorentzVectorD is similar to the templated version but is persistent capable in ROOT. It does, however, not inherit from TObject.

Persistence Within the ROOT framework.

10.9 StLorentzVectorF

Summary StLorentzVectorF is a non-template version of StLorentzVector<float> (see 10.7). The code does not contain templates nor does it make use of the Standard C++ library. All data member are of type float.

Synopsis

```
#include "StLorentzVectorF.hh"
StLorentzVectorF;
```

Description The member functions, operators and non-member functions are almost identical to those of StLorentzVector<T> but might be slightly slower in execution speed since the inline mechanism cannot be used as extensive as for the template version. Operations can be mixed with the non-template double precision version StLorentzVectorD and with StThreeVectorF, StThreeVectorD, StMatrixF, and StMatrixD but not with any instance of StLorentzVector<T>, StThreeVector<T> or StMatrix<T>.. StLorentzVector<T> should be preferred where possible.

Related Classes StLorentzVectorF is similar to the templated version but is persistent capable in ROOT. It does, however, not inherit from TObject.

Persistence Within the ROOT framework.

10.10 StMath

Summary `StMath.hh` is header file which contains declaration of various math function which are not defined in the standard math library `<cmath>` (`<math.h>`) but are useful for STAR offline purposes.

Synopsis

```
#include "StMath.hh"
```

Functions

`double probChiSquared(double chi2, unsigned int n)`
Computes the probability that a random variable, having a χ^2-distribution with `n` ≥ 1 degrees of freedom, assumes a value which is larger than a given value $\chi^2 \geq 0$. The algorithm was taken from the FORTRAN function `prob()` (CERNLIB G100).

10.11 StMatrix

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

Synopsis

```
// for bound checking on access operator
#define MATRIX_BOUND_CHECK
#include "StMatrix.hh"
template<class T> StMatrix;
```

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 $1 \leq i \leq p$ where p is the number of rows and $1 \leq j \leq q$ 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 $1 \leq i \leq p$ where p is the number of rows and $1 \leq j \leq q$ 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 $0 \leq c \leq (p-1)$ where p is the number of rows
and $0 \leq c \leq (q-1)$ where q is the number of columns.

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

10.12 StMatrixD

Summary StMatrixD is a non-template version of `StMatrix<double>` (see 10.11). The code does not contain templates nor does it make use of the Standard C++ library. All data member are of type `double`.

Synopsis

```
#include "StMatrixD.hh"
StMatrixD;
```

Description The member functions, operators and non-member functions are almost identical to those in StMatrix but might be slightly slower in execution speed since the inline mechanism cannot be used as extensive as for the template version. Operations can be mixed with the non-template single precision version `StMatrixF` and with `StThreeVectorF`, `StThreeVectorD`, `StLorentzVectorF`, and `StLorentzVectorD` but not with any instance of `StMatrix<T>`, `StThreeVector<T>` or `StLorentzVector<T>`.. Operations with STL based containers are not implemented. `StMatrix<T>` should be preferred where possible.

Related Classes StMatrixD is similar to the templated version but is persistent capable in ROOT. It does, however, not inherit from `TObject`.

Persistence Within the ROOT framework.

10.13 StMatrixF

Summary StMatrixF is a non-template version of `StMatrix<float>` (see 10.11). The code does not contain templates nor does it make use of the Standard C++ library. All data member are of type `float`.

Synopsis

```
#include "StMatrixF.hh"
StMatrixF;
```

Description The member functions, operators and non-member functions are almost identical to those of StMatrix but might be slightly slower in execution speed since the inline mechanism cannot be used as extensive as for the template version. Operations can be mixed with the non-template double precision version `StMatrixD` and with `StThreeVectorF`, `StThreeVectorD`, `StLorentzVectorF`, and `StLorentzVectorD` but not with any instance of `StMatrix<T>`, `StThreeVector<T>` or `StLorentzVector<T>`.. Operations with STL based containers are not implemented. `StMatrix<T>` should be preferred where possible.

Related Classes StMatrixF is similar to the templated version but is persistent capable in ROOT. It does, however, not inherit from `TObject`.

Persistence Within the ROOT framework.

10.14 StMemoryInfo

Summary Provides information on the memory allocator.

Synopsis

```
#include "StMemoryInfo.hh"
class StMemoryInfo;
```

Description Provides instrumentation describing space (memory) usage. The class is implemented as a singleton. The obtained information is platform dependent because of the different ways memory is allocated and managed by the different compilers and architectures. All memory information is obtained through a `mallinfo()` system call (usually in malloc.h).

The class is not functional on SUN (CC4.2). It will nevertheless compile and link but will provide no memory statistics.

Public Constructors StMemoryInfo is a singleton and therefore has no public constructor. The only way to obtain an instance, (or better an pointer to the only instance) of StMemoryInfo is through the static member function instance().

`static StMemoryInfo* instance();`
Returns pointer to the only instance of StMemoryInfo.

Public Member Functions

`void snapshot();`
Take a snapshot of the current memory usage.

`void print(ostream& = cout);`
Prints the memory usage as recorded by the snapshot() member function. The output is platform dependent. In any case the difference to the previous snapshot is printed as well.

Example

```
#include "StMemoryInfo.hh"

int main()
{
    char *buf[2];
    StMemoryInfo *info = StMemoryInfo::instance();

    info->snapshot();
    info->print();

    buf[0] = new char[10000];
    cout << "\n===> 10000 bytes allocated\n" << endl;

    info->snapshot();
    info->print();

    buf[1] = new char[10000];
    cout << "\n===>  10000 bytes allocated\n" << endl;

    info->snapshot();
    info->print();

    return 0;
```

```
}
```

Programs Output (Linux):

```
---------- Memory Status (snapshot #1) ----------
total space allocated from system       2832 (+0)
number of non-inuse chunks              1 (+0)
number of mmapped regions               0 (+0)
total space in mmapped regions          0 (+0)
total allocated space                   88 (+0)
total non-inuse space                   2744 (+0)
top-most, releasable space              2744 (+0)

===> 10000 bytes allocated

---------- Memory Status (snapshot #2) ----------
total space allocated from system       15120 (+12288)
number of non-inuse chunks              1 (+0)
number of mmapped regions               0 (+0)
total space in mmapped regions          0 (+0)
total allocated space                   10096 (+10008)
total non-inuse space                   5024 (+2280)
top-most, releasable space              5024 (+2280)

===>  10000 bytes allocated

---------- Memory Status (snapshot #3) ----------
total space allocated from system       27408 (+12288)
number of non-inuse chunks              1 (+0)
number of mmapped regions               0 (+0)
total space in mmapped regions          0 (+0)
total allocated space                   20104 (+10008)
total non-inuse space                   7304 (+2280)
top-most, releasable space              7304 (+2280)
```

10.15 StMemoryPool

Summary

Utility class which can be used to allocate a large number of small objects efficiently.

Synopsis

```
#include "StMemoryPool.hh"
class StMemoryPool(size_t);
```

Description

The idea is taken from: B. Stroustrup, The C++ Programming Language, third edition, Chapter 19.4.2 (page 570).

The class should be used if a large number of small objects has to be frequently allocated from heap. It is much more efficient and faster than the standard new/delete mechanism. Memory gets allocated in larger chunks and partitioned according to the size of the objects. The class manages the memory itself avoiding too frequent usage of new/delete (malloc/free). On order to use `StMemoryPool` the referring class (here called `X`) must be modified by adding the following lines:

```
// in the header file X.h
class X {
public:
    void* operator new(size_t) { return mPool.alloc(); }
    void  operator delete(void* p)  { mPool.free(p); }
    // ...
private:
    static StMemoryPool mPool;
    ...
}

// in the source file X.cc (X.cxx)
StMemoryPool X::mPool(sizeof(X));
```

Note, that the size of the class X does not increase sonce the pool is contained as a static member.

Only single objects may be created, i.e, new X[100] will not work; however, in these cases the default operators new/delete will be called.

Public Constructors

`StMemoryPool(unsigned int n);`
Creates a memory pool for objects of size `n`. The best way is to call the constructor as follows:

```
    StMemoryPool mPool(sizeof(X));
```

Public Member Functions

`void* alloc();`
Returns memory to hold one element.

`void  free(void*);`
Put the memory of one element back in the pool.

10.16 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"
class StPhysicalHelix;
```

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 10.5.

Persistence

None

Related Classes

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

Public Constructors

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

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

Don't forget to multiply the arguments with the units there are in. The units are defines in `SystemOfUnits.h` (see 9.3).

```
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.
Don't forget to provide the units when you pass the field. For example `0.5*tesla`. The units are defines in `SystemOfUnits.h` (see 9.3).

```
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.

Don't forget to provide the units when you pass the field. For example `0.5*tesla`. The units are defines in `SystemOfUnits.h` (see 9.3).

`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. Don't forget to provide the units when you pass the field. For example `0.5*tesla`. The units are defines in `SystemOfUnits.h` (see 9.3).

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
```

10.17 StPhysicalHelixD

Summary StPhysicalHelixD is similar to `StPhysicalHelix` (see 10.16) but does not contain or use templates nor does it make any use of the Standard C++ library.

Synopsis

```
#include "StPhysicalHelixD.hh"
StPhysicalHelixD;
```

Description The member functions, operators and non-member functions are identical to those of StPhysicalHelix with the exception that whenever `StPhysicalHelix` returns a `StThreeVector<double>` a `StThreeVectorD` is returned. The STL structure `pair<double, double>` used in this context is replaced by a similar but non-template structure `pairD`. The templated version should be preferred where possible.

Related Classes StPhysicalHelixD is derived from StHelixD. StPhysicalHelixD inherits also from TObject (through StPhysicalHelixD) if the SCL was compiled with the __ROOT__ flag set.

Persistence Within the ROOT framework.

10.18 StPrompt

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

Synopsis `#include "StPrompt.hh"`

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 [6].

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

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

```
        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
```

10.19 StRandom

Summary `StRandom` is an interface to the various random generators provided by CLHEP (see 10.25). It eases their use and avoids common problems. It is not intended not provide the full functionality of all the related CLHEP classes but the most common ones.

Synopsis

```
#include "StRandom.hh"
class StRandom;
```

Description CLHEP provides a rich set of random generators and random engines which can be mixed freely (see 10.25). However, their use is sometimes cumbersome and error-prone. The aim of `StRandom` is to provide an interface[7] which is simple to use and appropriate for most standard applications. No new algorithm is added; all methods are simple access functions. Only the most commonly used methods are interfaced. For example the generation of arrays of random numbers is omitted. If you need these features you have to use the original classes directly.

`StRandom` allows to generate all kinds of distributions (flat, Gaussian, Poissonian, exponential, and Breit-Wigner) without switching classes as one has to do with in CLHEP.

All internal engines and generators used in `StRandom` are kept as static data members and so are all methods. This allows their use without explicitly creating an instance of `StRandom`. For example, to generate a flat and a Gaussian random number you can either use:

```
StRandom rndm;
cout << rndm.flat() << endl;
cout << rndm.gauss(1, 0.3) << endl;
```

or without creating an instance explicitly:

```
cout << StRandom::flat() << endl;
cout << StRandom::gauss(1.7,0.3) << endl;
```

`StRandom` uses internally the most advanced random engine available, the RanluxEngine.

Note, that you are able to use StRandom everywhere in your code withour having to worry about getting the same series of random numbers. This happens if you create an instances of the CLHEP 'engines' locally, e.g. in a function which you invoke several times. The proper way to deal with CLHEP random generators is to have the engine in global scope and then create instances of the generators with the global engine as arguments.

Persistence None

[7] In terms of "Design Patterns" `StRandom` is a "Facade".

Related Classes

`StRandom` uses internally the following classes: `RanluxEngine`, `RandFlat`, `RandBreitWigner`, `RandExponential`, `RandGauss`, and `RandPoisson`.

Public Constructors

`StRandom();`

Constructor. Not needed (see example) unless you don't like the `StRandom::flat()` syntax.

Public Member Functions

`static void setSeed(long seed);`

Set the seed for the Ranlux engine. Ranlux internally translates this to a 24 words seed. Note that, since the engine is a static member, the seed is set for all instances used in the application.

```
static double flat();
static double flat(double w);
static double flat(double a, double b);
static long flatInt(long n);
static long flatInt(long m, long n);
static double exponential();
static double exponential(double mean);
static double gauss();
static double gauss(double mean, double stdDev);
static long   poisson(double mean);
static double breitWigner(double a=1.0, double b=0.2);
static double breitWigner(double a, double b, double c);
static double breitWignerM2(double a=1.0, double b=0.2);
static double breitWignerM2(double a, double b, double c);
```

Static methods to generate random numbers according to various distributions. All are implemented as inline functions and simply call the referring CLHEP methods.

Examples

```
#include <iostream.h>
#include "StRandom.hh"
#include "StGlobals.hh" // for PR() macro

int main()
{
    StRandom  rndm;
    int       i;
    const int n = 1;

    rndm.setSeed(101);
    for (i=0; i<n; i++)         PR(rndm.flat());
    for (i=0; i<n; i++)         PR(rndm.flat(2));
    for (i=0; i<n; i++)         PR(rndm.flatInt(10));
    for (i=0; i<n; i++)         PR(rndm.flatInt(20,30));
    for (i=0; i<n; i++)         PR(rndm.flat(1, 2));
    for (i=0; i<n; i++)         PR(rndm.exponential());
    for (i=0; i<n; i++)         PR(rndm.exponential(123.));
    for (i=0; i<n; i++)         PR(rndm.gauss());
    for (i=0; i<n; i++)         PR(rndm.gauss(10, 1.67));
    for (i=0; i<n; i++)         PR(rndm.poisson(6.4));

    StRandom::setSeed(1001);
```

```
    for (i=0; i<n; i++)         PR(StRandom::flat());
    for (i=0; i<n; i++)         PR(StRandom::flat(2));
    for (i=0; i<n; i++)         PR(StRandom::flat(1, 2));
    for (i=0; i<n; i++)         PR(StRandom::flatInt(10));
    for (i=0; i<n; i++)         PR(StRandom::flatInt(20,30));
    for (i=0; i<n; i++)         PR(StRandom::exponential());
    for (i=0; i<n; i++)         PR(StRandom::exponential(123.));
    for (i=0; i<n; i++)         PR(StRandom::gauss());
    for (i=0; i<n; i++)         PR(StRandom::gauss(10, 1.67));
    for (i=0; i<n; i++)         PR(StRandom::poisson(6.4));

    return 0;
}
```

Programs Output:

```
rndm.flat() = 0.995292
rndm.flat(2) = 0.727757
rndm.flatInt(10) = 6
rndm.flatInt(20,30) = 20
rndm.flat(1, 2) = 1.12937
rndm.exponential() = 0.80462
rndm.exponential(123.) = 21.5562
rndm.gauss() = 0.499951
rndm.gauss(10, 1.67) = 9.49933
rndm.poisson(6.4) = 9
StRandom::flat() = 0.0506576
StRandom::flat(2) = 1.71778
StRandom::flat(1, 2) = 1.87986
StRandom::flatInt(10) = 4
StRandom::flatInt(20,30) = 24
StRandom::exponential() = 0.614351
StRandom::exponential(123.) = 17.5155
StRandom::gauss() = -0.12305
StRandom::gauss(10, 1.67) = 11.2643
StRandom::poisson(6.4) = 4
```

10.20 StTemplates

Summary StTemplates is a header file needed on old cfront compilers to avoid unresolved symbols when linking with the SCL library.

Synopsis `#include "StTemplates.hh"`

Description If you are using an old `cfront` compiler who still uses template databases or repositories you might encounter unresolved symbols when linking with the SCL library. In order to avoid this problem one has to include `StTemplates.hh` once somewhere in the application. The only known platform where this is necessary is SUN with the CC4 compiler. Note, that one also has to set the `ST_SOLVE_TEMPLATES` flag in order to enable the template instatiation. This allows the use of `StTemplates.hh` also on non-cfront compilers. By setting or omitting this macro in the referring Makefile one can selectively switch the forced instantiation on and off.

10.21 StThreeVector

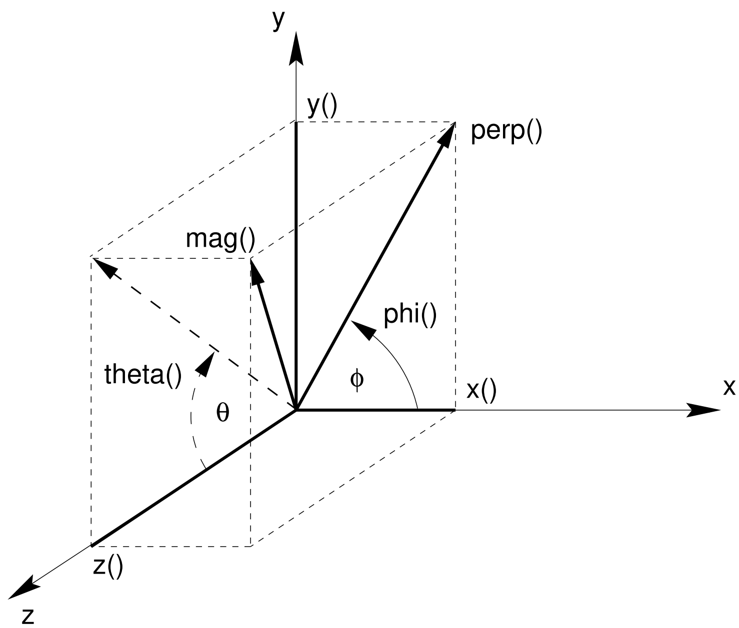


Figure 10.1: Components of three vector: x, y, z - basic components, θ - azimuth angle, ϕ - polar angle, mag = $\sqrt{x^2 + y^2 + z^2}$ - magnitude, perp = $\sqrt{x^2 + y^2}$ - transverse component.

Summary StThreeVector is a templated general 3-vector class defining vectors in three dimension (see Fig. 10.1).

Synopsis

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

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 X *avec)
```
Constructs a 3-vector from a given array `avec` of type X (usually `float` or `double`). This is especially useful when a C-style array has to be transformed into a StThreeVector. No checks on the array bounderies can be made. It is up to the user to make sure that the array has the correct size.

```
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);
T operator() (size_t i) const;
```
Returns components by index. The first version can be used also as lvalue. 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);`
`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 azimuthal angles constant.

`void setMagnitude(T r);`
Set the magnitude of the vector keeping the polar and azimuthal 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
```

10.22 StThreeVectorD

Summary StThreeVectorD is a non-template version of `StThreeVector<double>` (see 10.21). The code does not contain templates nor does it make use of the Standard C++ library. All data member are of type `double`.

Synopsis

```
#include "StThreeVectorD.hh"
StThreeVectorD;
```

Description The member functions, operators and non-member functions are identical to those of StThreeVector but might be slightly slower in execution speed since the inline mechanism cannot be used as extensive as for the template version. Operations can be mixed with the non-template single precision version `StThreeVectorF` but not with any instance of `StThreeVector<T>`. The templated version should be preferred where possible.

Related Classes StThreeVectorD is similar to the templated version but is persistent capable in ROOT. It does, however, not inherit from `TObject`.

Persistence Within the ROOT framework.

10.23 StThreeVectorF

Summary StThreeVectorF is a non-template version of `StThreeVector<float>` (see 10.21). The code does not contain templates nor does it make use of the Standard C++ library. All data member are of type `float`.

Synopsis

```
#include "StThreeVectorF.hh"
StThreeVectorF;
```

Description The member functions, operators and non-member functions are identical to those of StThreeVector but might be slightly slower in execution speed since the inline mechanism cannot be used as extensive as for the template version. Operations can be mixed with the non-template double precision version `StThreeVectorD` but not with any instance of `StThreeVector<T>`. The templated version should be preferred where possible.

Related Classes StThreeVectorF is similar to the templated version but is persistent capable in ROOT. It does, however, not inherit from `TObject`.

Persistence Within the ROOT framework.

10.24 StTimer

Summary CPU timer

Synopsis

```
#include "StTimer.hh"
StTimer timer;
```

Description This class can measure elapsed CPU (user) time. The timer has two states: running and stopped. The timer measures the total amount of time spent in the ”running” state since it was either constructed or reset.

The timer is put into the ”running” state by calling member function `start()`. It is put into the ”stopped” state by calling `stop()`.

StTimer uses the system-dpendent function `clock()` which returns the number of ”ticks” since it was first called. As a result, StTimer will not be able to measure intervals longer than some system-dependent value. (For instance, on several common UNIX systems, this value is just under 36 minutes.)

The resolution of the timer is given by the number of ”ticks” per second. This value is system dependent and can be checked with the `resolution()` member function. Make sure that the CPU time between the start and stop of the timer is significantly larger than the resolution.

N.B. The interface of this class and its functionality were adapted from the tools.h++ class library from RogueWave.

Related Classes None

Persistence None

Public Constructors

`StTimer();`
Constructs a new timer. The timer will not start running until until `start()` is called.

Public Member Functions

`double resolution() const`
Returns the minimal amount of time in seconds which can be measured by the timer. This value is system dependent. Typical values range from 10 μs to 10 ms.

`double elapsedTime() const`
Returns the amount of (CPU) time that has accumulated while the timer was in the running state.

`void reset()`
Resets (and stops) the timer.

`void start()`
Puts the timer in the ”running” state. Time accumulates while in this state.

`void stop()`
Puts the timer in the ”stopped” state. Time will not accumulate while in this state.

Example

```
#include "StTimer.hh"
#include <time.h>
#include <iostream.h>
#include <math.h>

int main()
{
    StTimer timer, totalTimer;

    totalTimer.start();
    timer.start();

    // Spend 5 busy sec.
    time_t begin = time(0);
    time_t now = begin;
    while (now - begin < 5) now = time(0);

    timer.stop();

    cout << "Test 1:" << endl;
    cout << "This test should require less than 5 sec CPU time. \n"
         << "The exact amount depends strongly on the system." << endl;
    cout << "The measured elapsed CPU time is: "
         << timer.elapsedTime() << " sec\n" << endl;

    timer.reset();
    timer.start();

    const size_t NumSqrt = 1000000;
    double x;
    for (int i=0; i<NumSqrt; i++) x = sqrt(double(i));

    timer.stop();

    cout << "Test 2:" << endl;
    cout << "The CPU time to calculate " << NumSqrt
         << " square roots is: "
         << timer.elapsedTime() << " sec\n" << endl;

    cout << "The total amount of CPU seconds used \n"
         << "to execute this program is: ";
    totalTimer.stop();
    cout << totalTimer.elapsedTime() << " sec." << endl;

    return 0;
}
```

Programs Output:

```
Test 1:
This test should require less than 5 sec CPU time.
The exact amount depends strongly on the system.
The measured elapsed CPU time is: 4.81 sec

Test 2:
The CPU time to calculate 1000000 square roots is: 0.25 sec

The total amount of CPU seconds used
```

```
to execute this program is: 5.06 sec.
```

10.25 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"

// Note: all Random header files are contained in:
#include "Randomize.h"
```

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 [8] 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. $h = -\text{sign}(qB) = \pm 1$

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.

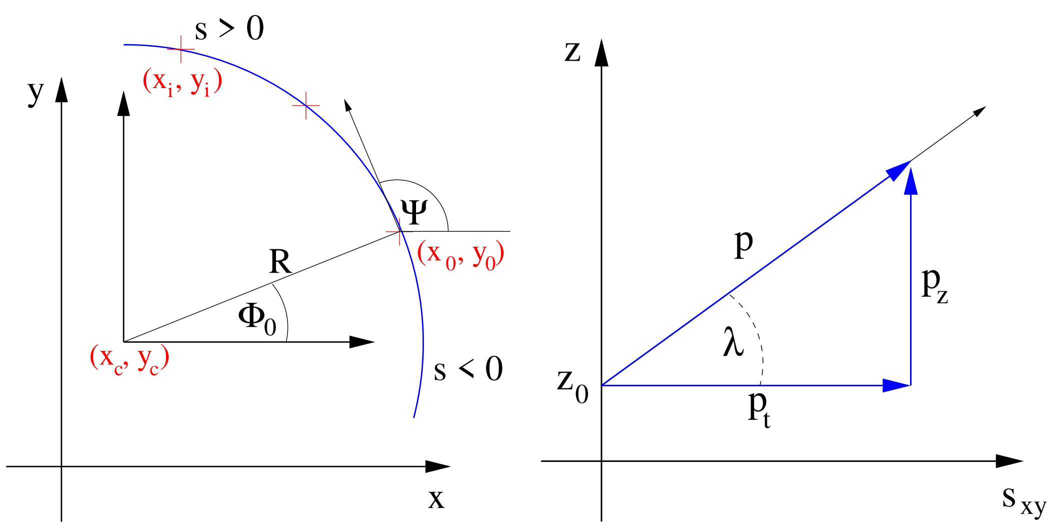
The meaning of the different parameters is visualized in Fig. A.1.

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 (see Fig. A.1) is defined as follows:

$$\Phi_0 = \arctan\left(\frac{y_0 - y_c}{x_0 - x_c}\right) \quad (4)$$

[8] see A.8 for a detailed discussion on the number of parameters needed.



(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 A.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

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

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

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

A.3 Distance of closest approach between two helices

The closest distance between two helices H_1 and H_2 is a problem which again can be solved analytically only in 2 dimensions, i.e., in the xy-plane. The solution in 3 dimensions cannot even be solved by standard numerical methods (as the Newton method) but requires more sophisticated method since we have to find 2 unknown parameters s_1 and s_2 in

$$\frac{\mathrm{d}^2 M(s_1, s_2)}{\mathrm{d}s_1 \mathrm{d}s_2} = 0 \tag{20}$$

where M is the distance between the two helices at s_1 and s_2.

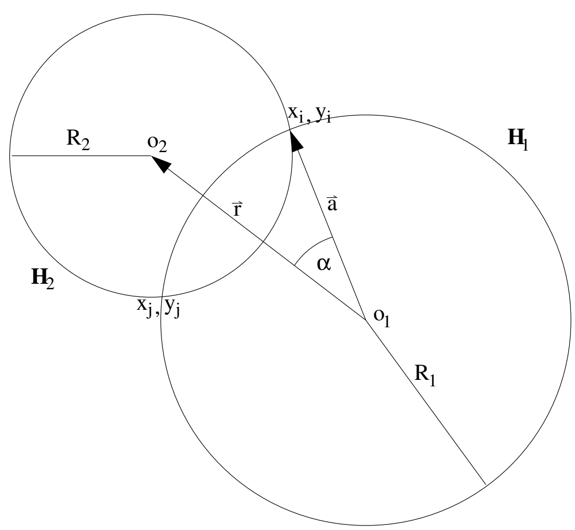


Figure A.2: Two intersecting helices in the xy-plane

In the xy-plane:
Given two helices with radii R_1 and R_2 and centers in the xy-plane $o_1 = (x_{c1}, y_{c1})$ and $o_2 = (x_{c2}, y_{c2})$ we have to find vector $\vec{a}$ as depicted in Fig.A.3. The angle α can be calculated as:

$$\cos\alpha = \frac{R_1^2 + |\vec{r}|^2 - R_2^2}{2R_1|\vec{r}|} \tag{21}$$

where $\vec{r}$ is the vector between the two centers. The absolute coordinates of one intersection point (measured from o_1) can be obtained by calculating vector $\vec{a}$ and adding o_1.

$$x_i = x_{c1} + R_1[(x_{c2} - x_{c1})\cos\alpha - (y_{c2} - y_{c1})\sin\alpha]/|\vec{r}|; \tag{22}$$
$$y_i = y_{c1} + R_1[(x_{c2} - x_{c1})\sin\alpha + (y_{c2} - y_{c1})\cos\alpha]/|\vec{r}|; \tag{23}$$

If $\cos\alpha$ is exactly 1 we have only one solution. For the case $\cos\alpha < 1$ we get two valid intersection points (x_i, y_i) and (x_j, y_j) where the latter is simply given by:

$$x_j = x_{c1} + R_1[(x_{c2} - x_{c1})\cos\alpha + (y_{c2} - y_{c1})\sin\alpha]/|\vec{r}|; \tag{24}$$
$$y_j = y_{c1} + R_1[(y_{c2} - y_{c1})\cos\alpha - (x_{c2} - x_{c1})\sin\alpha]/|\vec{r}|; \tag{25}$$

In the case $\cos\alpha > 1$ the circles do not intersect. Then the distance of closest approach is simply given by the intersection of a line between the two centers and the two helices. For helix H_1 we get:

$$x = x_{c1} + R_1(x_{c2} - x_{c1})/|\vec{r}|; \tag{26}$$
$$y = y_{c1} + R_1(y_{c2} - y_{c1})/|\vec{r}|; \tag{27}$$

In 3 dimensions:
Usually an iteration method is applied which uses the intersection points in the xy-plane as start values. Care has to be taken if both helices have different dip angle λ since the start values then significantly deviate from the actual solution.

A.4 Intersection with a cylinder (ρ=const)

In order to obtain the path length s at which the helix intersects with a cylinder of given radius ρ we have to solve the following equation:

$$\rho^2 = x(s)^2 + y(s)^2 \tag{28}$$

Using eq. 1 and 2 we obtain the two analytic solutions for s_1 and s_2:

$$\begin{aligned} s_{1/2} = & -\big(\Phi_0 + 2\arctan\big[\big(2\,y_0\,\kappa - 2\sin\Phi_0 \pm \big[-\kappa^2\big(-4\,\rho^2 + 4\,{y_0}^2 - 2\,\rho^2\kappa^2 x_0^2 - \\ & 2\,\rho^2\kappa^2{y_0}^2 + 2\,{x_0}^2\kappa^2{y_0}^2 + \rho^4\kappa^2 + {x_0}^4\kappa^2 + {y_0}^4\kappa^2 - 4\,{x_0}^3\kappa\cos\Phi_0 + \\ & 4\,{x_0}^2\cos^2\Phi_0 - 4\,{y_0}^2\cos^2\Phi_0 - \\ & 4\,{y_0}^3\kappa\sin\Phi_0 + 4\,\rho^2\kappa\,x_0\cos\Phi_0 + 4\,\rho^2\kappa\,y_0\sin\Phi_0 - 4\,{x_0}^2\kappa\,y_0\sin\Phi_0 - \\ & 4\,{y_0}^2\kappa\,x_0\cos\Phi_0 + 8\,x_0\cos\Phi_0\,y_0\sin\Phi_0\big)\big]^{1/2}\Big)\,/ \\ & \big(-\rho^2\kappa^2 + 2 + {x_0}^2\kappa^2 + 2\cos\Phi_0 + {y_0}^2\kappa^2 - \\ & 2\,x_0\,\kappa - 2\,x_0\,\kappa\cos\Phi_0 - 2\,y_0\,\kappa\sin\Phi_0)\big]\big)\,h^{-1}\kappa^{-1}(\cos\lambda)^{-1} \end{aligned} \tag{29}$$

A.5 Intersection with a plane

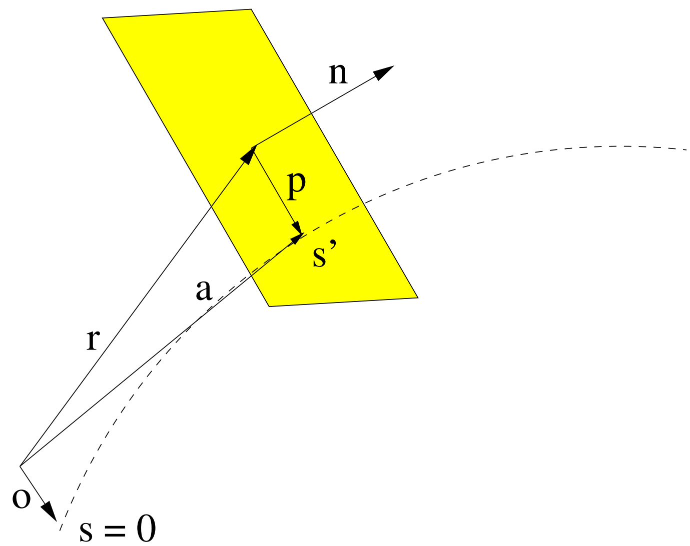


Figure A.3: Intersection of a helix with a plane

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:

$$\vec{p} \cdot \vec{n} = 0. \tag{30}$$

Hence:

$$(\vec{a} - \vec{r}) \cdot \vec{n} = 0. \tag{31}$$

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:

$$x(s)n_x + y(s)n_y + z(s)n_z - \vec{r} \cdot \vec{n} \quad = \tag{32}$$

$$A + n_x \cos S + n_y \sin S + \kappa n_z s \sin \lambda \quad = \quad 0 \tag{33}$$

where:

$$A \quad = \quad \kappa(\vec{o} \cdot \vec{n} - \vec{r} \cdot \vec{n}) - n_x \cos \Phi_0 - n_y \sin \Phi_0 \tag{34}$$

$$S \quad = \quad hs\kappa \cos \lambda + \Phi_0 \tag{35}$$

The root of eq. 33 can now easily be determined by a suitable numerical method (Newton).

A.6 Limitations

The only non-numerical limitations of this parametrization are:

$$
\begin{aligned}
-\pi/2 < \lambda &< \pi/2 && (36)\\
\kappa &> 0 && (37)
\end{aligned}
$$

A.7 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:

$$
\begin{aligned}
x(s) &= x_0 - sh\cos\lambda\sin\Phi_0 && (38)\\
y(s) &= y_0 + sh\cos\lambda\cos\Phi_0 && (39)\\
z(s) &= z_0 + s\ \sin\lambda && (40)
\end{aligned}
$$

Important: For $B = 0$ the sense of rotation is ill defined. All what matters is that $\Phi_0 = \Psi - h\pi/2$ is done correctly, i.e. with the same arbitrary h. In the following we assume $h = +1$ for convenience.

Eq. 15 then reads as:

$$
s' = \frac{1}{\cos\lambda}\left[(y_i - y_0)\cos\Phi_0 - (x_i - x_0)\sin\Phi_0\right] \qquad (41)
$$

Eq. 17 can now be solved analytically;

$$
\frac{\mathrm{d}M_i^{dca}}{\mathrm{d}s} = 0 \qquad (42)
$$

gives:

$$
\begin{aligned}
s^{dca} = \ & \cos\lambda\cos\Phi_0(y_i - y_0) - \\
& \cos\lambda\sin\Phi_0(x_i - x_0) + \\
& \sin\lambda(z_i - z_0) \qquad (43)
\end{aligned}
$$

The solution for the intersection with a cylinder (eq. 29) now reads:

$$
\begin{aligned}
s_{1/2} = \ & [x_0\ \cos\lambda\ \sin\Phi_0\ - y_0\ \cos\lambda\ \cos\Phi_0 \pm && (44)\\
& \Big[-\cos^2\lambda\ \big(2\,x_0\ \cos\Phi_0\, y_0\ \sin\Phi_0\ - \rho^2 + && (45)\\
& y_0^2 - y_0^2\ \cos^2\Phi_0\ + x_0^2\cos^2\Phi_0\big)\Big]^{1/2}\Big]\ \cos^2\lambda && (46)
\end{aligned}
$$

The same holds for the intersection of a helix with a plane where in case of zero curvature eq. 33 can be solved analytically.

$$
s' = \frac{\vec{r}\cdot\vec{n} - \vec{o}\cdot\vec{n}}{-n_x\cos\lambda\sin\Phi_0 + n_y\cos\lambda\cos\Phi_0 + n_z\sin\lambda} \qquad (47)
$$

A.8 Why are there only 5 independent helix parameters?

Imagine an arbitrary helix sitting in 3D space. What is required to completely specify it ?

1. the line coinciding with the axis - if its oriented in any arbitrary direction, then this requires 4 parameters, or 2 direction angles (theta and phi in usual spherical coordinates) plus 2 more coordinates to locate the line in a plane perpendicular to this direction.

   For the special case in STAR we always fix the direction parallel to the z-axis so this reduces the number of this subset of parameters from 4 to 2.

   So these are the (x,y) coordinates of the center of the circular projection onto the x-y plane.

2. Then we must give the radius of the circular projection - 1 more param,

3. Next we must specify the pitch and the handedness. - 2 more params.

4. Finally, we have to give a phase angle or some single number that tells us where this thing is actually sitting w.r.t. some given plane. For STAR this could be the phase angle at the point where the helix intersects the x-y plane. - 1 more parameter.

So, in general there are 7 continuously varying parameters plus a handedness switch. For the special case STAR uses there are then 5 independent parameters plus the left handed/right handed switch. So yes, 6 parameters are required. But for track fitting purposes only 5 are relevant. The handedness of the particle's trajectory will be determined by the sign of $B_z\times$ charge (using the actual charge) and the sign of the z-component of momentum (using the actual p_z momentum value). However, in general the charge sign and p_z direction are not known, based on track fitting alone. These signs must be assumed using some selection criteria, usually that the particle is moving "outward and away" from the general area of the beam. These two signs are not independent of each other but must be chosen to give a path consistent with the handedness that the space point positions require. So there is one algebraic sign that is ambiguous and that we have to choose. Having done this there remain 5 independent fitting parameters. In our track parametrization we put the choice of algebraic sign into the both that of the charge and the tanl parameter, consistently we hope.

So to summarize, there are 6 parameters, one is a sign selected by some criteria, the remaining 5 are varied to fit the space points. The track parameter error matrix is then 5×5 symmetric, and thus includes 15 distinct quantities.

Text from Lanny Ray written during an email exchange on this very topic.

Index