# S-Class

*An active library for high performance computations in C++*

T. Glebe, MPI fuer Kernphysik Heidelberg

# Contents

# 1 smatrix

*An active library for high performance vector and matrix computations in C++*

**Names**

**Author:** T. Glebe, MPI fuer Kernphysik Heidelberg

## 1.1 template <class T, unsigned int D> class **SVector**

*SVector*

**Public Members**

**— Typedefs —**

typedef T **value_type**

**— Constructors —**

**SVector** ()

template <class A>
**SVector** (const Expr<A, T, D>& rhs)

**SVector** (const SVector<T, D>& rhs)

template <unsigned int D1>
**SVector** (const SVector<T, D1>& rhs)
*$D1 \leq D$ required!*

template <unsigned int D1>
**SVector** (const T& a1, const SVector<T, D1>& rhs)
*$D1 \leq D - 1$ required!*

**SVector** (const T* a, unsigned int len)
*fill from array, len must be equal to D!*

**SVector** (const T& rhs)

**SVector** (const T& a1, const T& a2)

**SVector** (const T& a1, const T& a2, const T& a3)

**SVector** (const T& a1, const T& a2, const T& a3, const T& a4)

**SVector** (const T& a1, const T& a2, const T& a3, const T& a4, const T& a5)

**SVector** (const T& a1, const T& a2, const T& a3, const T& a4, const T& a5, const T& a6)

**SVector** (const T& a1, const T& a2,
const T& a3, const T& a4,
const T& a5, const T& a6,
const T& a7)

**SVector** (const T& a1, const T& a2,
const T& a3, const T& a4,
const T& a5, const T& a6,
const T& a7, const T& a8)

**SVector** (const T& a1, const T& a2,
const T& a3, const T& a4,
const T& a5, const T& a6,
const T& a7, const T& a8,
const T& a9)

**SVector** (const T& a1, const T& a2,
const T& a3, const T& a4,
const T& a5, const T& a6,
const T& a7, const T& a8,
const T& a9, const T& a10)

SVector <T, D> &
**operator=** (const T& rhs)

template <class A> SVector <T, D> &
**operator=** (const Expr<A, T, D>& rhs)

**— Access functions —**

inline static unsigned int
**dim** () *return dimension D*

T **apply** (unsigned int i) const
*access the parse tree*

const T* **Array** () const *return read-only pointer to internal array*

T* **Array** () *return pointer to internal array*

**— Operators —**

const T& **operator[]** (unsigned int i) const
*read-only access*

const T& **operator)** (unsigned int i) const *read-only access*

T& **operator[]** (unsigned int i)
*read/write access*

T& **operator)** (unsigned int i)
*read/write access*

SVector <T, D> &
**operator+=** (const SVector<T, D>& rhs)

template <class A> SVector <T, D> &
**operator+=** (const Expr<A, T, D>& rhs)

SVector <T, D> &
**operator-=** (const SVector<T, D>& rhs)

template <class A> SVector <T, D> &
**operator-=** (const Expr<A, T, D>& rhs)

SVector <T, D> &
**operator*=** (const SVector<T, D>& rhs)

template <class A> SVector <T, D> &
**operator*=** (const Expr<A, T, D>& rhs)

SVector <T, D> &
**operator/=** (const SVector<T, D>& rhs)

template <class A> SVector <T, D> &
**operator/=** (const Expr<A, T, D>& rhs)

**— Expert functions —**

SVector <T, D> &
**unit** () *transform vector into a vector of lenght 1*

template <unsigned int D2> SVector <T, D> &
**place_at** (const SVector<T, D2>& rhs, const unsigned int row)
*place a sub-vector starting at <row>*

template <class A, unsigned int D2> SVector <T, D> &
**place_at** (const Expr<A, T, D2>& rhs, const unsigned int row)
*place a sub-vector starting at <row>*

std::ostream&
**print** (std::ostream& os) const
*used by operator<<()*

SVector. A generic fixed size Vector class.

**Author:** T. Glebe

## 1.2

template <class T, unsigned int D1, unsigned int D2 = D1> class **SMatrix**

*SMatrix*

**Public Members**

**— Typedefs —**

typedef T **value_type**

**— Constructors —**

**SMatrix** ()

**SMatrix** (const SMatrix<T, D1, D2>& rhs)

template <class A>
**SMatrix** (const Expr<A, T, D1, D2>& rhs)

**SMatrix** (const T& rhs, const bool diagonal=false)
*2nd arg: set only diagonal?*

**SMatrix** (const SVector<T, D1>& rhs)
*constructor via dyadic product*

template <class A>
**SMatrix** (const Expr<A, T, D1>& rhs)
*constructor via dyadic product*

template <class T1>
**SMatrix** (const T1* a, const bool triang=false, const unsigned int len=D1*D2)
*constructor via array, triag=true: array contains only upper/lower triangular part of a symmetric matrix, len: length of array*

SMatrix <T, D1, D2> &

**operator=** (const T& rhs)

template <class A> SMatrix <T, D1, D2> &
**operator=** (const Expr<A, T, D1, D2>& rhs)

static const unsigned int
**rows**

static const unsigned int
**cols**

static const unsigned int
**size**

**— Access functions —**

T **apply** (unsigned int i) const
*access the parse tree*

const T* **Array** () const *return read-only pointer to internal array*

T* **Array** () *return pointer to internal array*

**— Operators —**

const T& **operator)** (unsigned int i, unsigned int j) const
*read-only access*

T& **operator)** (unsigned int i, unsigned int j)
*read/write access*

SMatrix <T, D1, D2> &
**operator+=** (const SMatrix<T, D1, D2>& rhs)

template <class A> SMatrix <T, D1, D2> &
**operator+=** (const Expr<A, T, D1, D2>& rhs)

SMatrix <T, D1, D2> &
**operator-=** (const SMatrix<T, D1, D2>& rhs)

template <class A> SMatrix <T, D1, D2> &
**operator-=** (const Expr<A, T, D1, D2>& rhs)

SMatrix <T, D1, D2> &
**operator*=** (const SMatrix<T, D1, D2>& rhs)

template <class A> SMatrix <T, D1, D2> &
**operator*=** (const Expr<A, T, D1, D2>& rhs)

SMatrix <T, D1, D2> &
**operator/=** (const SMatrix<T, D1, D2>& rhs)

template <class A> SMatrix <T, D1, D2> &

**operator/=** (const Expr<A, T, D1, D2>& rhs)

**— Expert functions —**

| | | | | |
|---|---|---|---|---|
| | bool | **sinvert** () | *invert symmetric, pos. def. Matrix via Dsinv* | |
| 1.2.1 | bool | **sdet** (T& det) | *determinant of symmetrc, pos* | 12 |
| | bool | **invert** () | *invert square Matrix via Dinv* | |
| 1.2.2 | bool | **det** (T& det) | *determinant of square Matrix via Dfact* ...................... | 12 |

template <unsigned int D> SMatrix <T, D1, D2> &
**place_in_row** (const SVector<T, D>& rhs,
const unsigned int row,
const unsigned int col)
*place a vector in a Matrix row*

template <class A, unsigned int D> SMatrix <T, D1, D2> &
**place_in_row** (const Expr<A, T, D>& rhs,
const unsigned int row,
const unsigned int col)
*place a vector expression in a Matrix row*

template <unsigned int D> SMatrix <T, D1, D2> &
**place_in_col** (const SVector<T, D>& rhs,
const unsigned int row,
const unsigned int col)
*place a vector in a Matrix column*

template <class A, unsigned int D> SMatrix <T, D1, D2> &
**place_in_col** (const Expr<A, T, D>& rhs,
const unsigned int row,
const unsigned int col)
*place a vector expression in a Matrix column*

template <unsigned int D3, unsigned int D4> SMatrix <T, D1, D2> &
**place_at** (const SMatrix<T, D3, D4>& rhs,
const unsigned int row,
const unsigned int col)
*place a matrix in this matrix*

template <class A, unsigned int D3, unsigned int D4> SMatrix <T, D1, D2> &

**place_at** (const Expr<A, T, D3, D4>& rhs, const unsigned int row, const unsigned int col)
*place a matrix expression in this matrix*

SVector <T, D2>
**row** (const unsigned int therow) const
*return a Matrix row as a vector*

SVector <T, D1>
**col** (const unsigned int thecol) const
*return a Matrix column as a vector*

std::ostream&
**print** (std::ostream& os) const
*used by operator<<()*

SMatrix. A generic fixed size n x m Matrix class.q

**Author:** T. Glebe

### 1.2.1

bool **sdet** (T& det)

*determinant of symmetrc, pos*

determinant of symmetrc, pos. def. Matrix via Dsfact. **Note:** this will destroy the contents of the Matrix!

### 1.2.2

bool **det** (T& det)

*determinant of square Matrix via Dfact*

determinant of square Matrix via Dfact. **Note:** this will destroy the contents of the Matrix!

## 1.3

template <class ExprType, class T, unsigned int D, unsigned int D2 = 0> class **Expr**

*Expr.*

**Public Members**

**Expr** (const ExprType& rhs)

~**Expr** ()

inline T **apply** (unsigned int i) const

static const unsigned int **rows**

static const unsigned int **cols**

Expr. An Expression wrapper class.

**Author:** T. Glebe

## 1.4

template <class Operator, class LHS, class RHS, class T> class **BinaryOp**

*BinaryOp*

**Public Members**

**BinaryOp** ( Operator op, const LHS& lhs, const RHS& rhs)

~**BinaryOp** ()

inline T **apply** (unsigned int i) const

BinaryOp. A class representing binary operators in the parse tree.

**Author:** T. Glebe

## 1.5

template <class Operator, class RHS, class T> class **UnaryOp**

*UnaryOp*

**Public Members**

| | |
|---|---|
| | **UnaryOp** ( Operator op, const RHS& rhs) |
| | ~**UnaryOp** () |
| inline T | **apply** (unsigned int i) const |

UnaryOp. A class representing unary operators in the parse tree.

**Author:** T. Glebe

## 1.6

template <class T> class **Constant**

*Constant*

**Public Members**

| | |
|---|---|
| | **Constant** ( const T& rhs ) |
| | ~**Constant** () |
| inline T | **apply** (unsigned int i) const |

Constant. A class representing constant expressions (literals) in the parse tree.

**Author:** T. Glebe

**1.7**

template <class T> inline const T **square** (const T& x)

*square*

square. Template to compute $x \cdot x$

**Author:** T. Glebe

**1.8**

template <class T> inline const T **maximum** (const T& lhs, const T& rhs)

*maximum*

maximum. Template to compute $\max(i, j)$

**Author:** T. Glebe

**1.9**

template <class T> inline const T **minimum** (const T& lhs, const T& rhs)

*minimum*

minimum. Template to compute $\min(i, j)$

**Author:** T. Glebe

**1.10**

template <class T> inline int **round** (const T& x)

*round*

round. Template to compute nearest integer value.

**Author:** T. Glebe

**1.11**

template <class T> inline const int **sign** (const T& x)

*sign*

sign. Template to compute the sign of a number sgn$(i)$.

**Author:** T. Glebe

**1.12**

template <class T, unsigned int D> inline T **dot** (const SVector<T, D>& lhs, const SVector<T, D>& rhs)

*dot*

dot. Template to compute $\vec{a} \cdot \vec{b} = \sum_i a_i \cdot b_i$.

**Author:** T. Glebe

**1.13**

template <class T, unsigned int D> inline T **mag2** (const SVector<T, D>& rhs)

*mag2*

mag2. Template to compute $|\vec{v}|^2 = \sum_i v_i^2$.

**Author:** T. Glebe

**1.14**

template <class T, unsigned int D> inline T **mag** (const SVector<T, D>& rhs)

*mag*

mag. Length of a vector: $|\vec{v}| = \sqrt{\sum_i v_i^2}$.

**Author:** T. Glebe

**1.15**

template <class T> inline T **Lmag2** (const SVector<T, 4>& rhs)

*Lmag2*

Lmag2. Template to compute $|\vec{v}|^2 = v_0^2 - v_1^2 - v_2^2 - v_3^2$.

**Author:** T. Glebe

**1.16**

template <class T> inline T **Lmag** (const SVector<T, 4>& rhs)

*Lmag*

Lmag. Length of a vector Lorentz-Vector: $|\vec{v}| = \sqrt{v_0^2 - v_1^2 - v_2^2 - v_3^2}$.

**Author:** T. Glebe

**1.17**

template <class T> inline SVector <T,3> **cross** (const SVector<T, 3>& lhs, const SVector<T, 3>& rhs)

*cross*

cross. Cross product of two 3-dim vectors: $\vec{c} = \vec{a} \times \vec{b}$.

**Author:** T. Glebe

**1.18**

template <class T, unsigned int D> inline SVector <T,D> **unit** (const SVector<T, D>& rhs)

*unit*

unit. Return a vector of unit lenght: $\vec{e}_v = \vec{v}/|\vec{v}|$.

**Author:** T. Glebe

**1.19**

template <class Matrix, unsigned int n, unsigned int idim> bool **Dfact** (Matrix& rhs, typename Matrix::value_type& det)

*Dfact*

Dfact. Function to compute the determinant from a square matrix ($\det(A)$) of dimension $idim$ and order $n$.

**Author:** T. Glebe

**1.20**

template <class Matrix, unsigned int n, unsigned int idim> bool **Dfactir** (Matrix& rhs, typename Matrix::value_type& det, unsigned int* ir)

*Dfactir*

Dfactir. Function to compute the determinant from a square matrix ($\det(A)$) of dimension $idim$ and order $n$. A working area $ir$ is returned which is needed by the Dfinv routine.

**Author:** T. Glebe

**1.21**

template <class Matrix, unsigned int n, unsigned int idim> bool **Dfinv** (Matrix& rhs, unsigned int* ir)

*Dfinv*

Dfinv. Function to compute the inverse of a square matrix ($A^{-1}$) of dimension $idim$ and order $n$. The routine Dfactir must be called before Dfinv!

**Author:** T. Glebe

**1.22**

## template <unsigned int idim, unsigned int n = idim> class **Invert**

*Invert*

**Public Members**

template <class Matrix> static bool
**Dinv** (Matrix& rhs)

Invert. Class to specialize calls to Dinv. Dinv computes the inverse of a square matrix if dimension $idim$ and order $n$. The content of the matrix will be replaced by its inverse. In case the inversion fails, the matrix content is destroyed. Invert specializes Dinv by the matrix order. E.g. if the order of the matrix is two, the routine Invert<2> is called which implements Cramers rule.

**Author:** T. Glebe

**1.23**

## template <> class Invert **0**

*Invert<0>*

**Public Members**

template <class Matrix> inline static bool
**Dinv** (Matrix& rhs)

Invert<0>. In case of zero order, do nothing.

**Author:** T. Glebe

**1.24**

## template <> class Invert **1**

*Invert<1>*

**Public Members**

template <class Matrix> static bool
**Dinv** (Matrix& rhs)

Invert<1>. $1 \times 1$ (sub-)matrix. $a_{11} \to 1/a_{11}$

**Author:** T. Glebe

**1.25**

## template <> class Invert **2**

*Invert<2>*

**Public Members**

template <class Matrix> static bool
**Dinv** (Matrix& rhs)

Invert<2>. $2 \times 2$ (sub-)matrix. Use Cramers rule.

**Author:** T. Glebe

**1.26**

template <> class Invert **3**

*Invert<3>*

**Public Members**

template <class Matrix> static bool
**Dinv** (Matrix& rhs)

Invert<3>. $3 \times 3$ (sub-)matrix. Use pivotisation.

**Author:** T. Glebe

**1.27**

template <class Matrix, unsigned int n, unsigned int idim> bool **Dsfact** (Matrix& rhs, typename Matrix::value_type& det)

*Dsfact*

Dsfact. Compute determinant of a symmetric, positive definite matrix of dimension $idim$ and order $n$.

**Author:** T. Glebe

**1.28**

template <class T, int n, int idim> bool **Dsinv** (T* a)

*Dsinv*

Dsinv. Compute inverse of a symmetric, positive definite matrix of dimension $idim$ and order $n$.

**Author:** T. Glebe

**1.29**

template <class T> T **erf_0_** (int n, const T& rhs)

*erf_0_*

erf_0_. Routine which does the basic work.

**Author:** T. Glebe

**1.30**

template <class T> T **erf** (const T& rhs)

*Error function*

Error function.

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

**Author:** T. Glebe

**1.31**

template <class T> T **erfc** (const T& rhs)

*Error function*

Error function.

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

**Author:** T. Glebe

**1.32**

template <class T> T **Prob** (const T& rhs, int n)

*Upper Tail Probability of Chi-Squared Distribution*

Upper Tail Probability of Chi-Squared Distribution.

$$Q(X|N) = \frac{1}{\sqrt{2^N}\Gamma(\frac{1}{2}N)} \int_X^\infty e^{-\frac{1}{2}t} t^{\frac{1}{2}N-1} dt$$

**Author:** T. Glebe

**1.33**

static bool **errfunc** (const char* expr, const char* function, const unsigned int line)

*errfunc*

errfunc. Function which prints an error message and then exits the program.

**Author:** T. Glebe

**1.34**

#define **Sassert** (expr)

*Sassert*

Sassert. An assert macro which is invoced only once (in contrast to standard assert(), which is repeatedly called). Sassert is more performant in case it is placed in a routine which is called more than once. Otherwise assert() is more performant.

**Author:** T. Glebe

**1.35**

template <class T, class P> bool **Rminfc** ( T (*f)(P), P a, P b, double eps, double delta, P& x, P& y)

*Rminfc*

Rminfc. Function which finds a single, local minimum of a function with one variable in a given interval. The "golden section search" is applied. The method uses a fixed number $n$ of function evaluations, where $n = [2.08 \cdot \ln(|a-b|/\epsilon) + 1/2] + 1$.

**Parameters:**
`f` — one-dimensional function
`a,b` — end-points of search interval
`eps` — accuracy parameter $\epsilon$
`delta` — tolerance interval $\delta$ near $a$ and $b$. Suggested value: $\delta = 10\epsilon$
`x` — computed approximation to the abscissa of a minimum of the function $f$
`y` — value of $f(x)$

**1.36**

template <class C, class T, class P> bool **RminfcM** (const C& part, T (C::*f)(P) const, P a, P b, double eps, double delta, P& x, P& y)

*RminfcM*

RminfcM. Rminfc version for const class member functions.

Function which finds a single, local minimum of a function with one variable in a given interval. The "golden section search" is applied. The method uses a fixed number $n$ of function evaluations, where $n = [2.08 \cdot \ln(|a-b|/\epsilon) + 1/2] + 1$.

**Parameters:**

`f` — one-dimensional function
`a,b` — end-points of search interval
`eps` — accuracy parameter $\epsilon$
`delta` — tolerance interval $\delta$ near $a$ and $b$. Suggested value: $\delta = 10\epsilon$
`x` — computed approximation to the abscissa of a minimum of the function $f$
`y` — value of $f(x)$

# 2 SVertex

*A high-speed C++ vertex fit class (formerly Vt++).*

**Names**

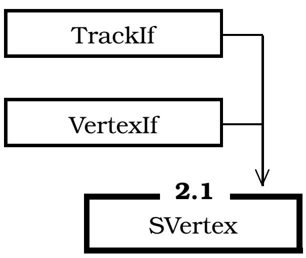


**Author:** T. Glebe, MPI fuer Kernphysik Heidelberg

## 2.1 template <unsigned int NTR> class **SVertex** : public VertexIf, public TrackIf

*Fixed size Vertex class*

**Inheritance**



**Public Members**

**— Constructors —**

**SVertex** ()

**SVertex** (const SVertex<NTR>& rhs)

**SVertex** (const TrackIf& t)
*create Vertex object & set mother track by t*

**SVertex** (const TrackIf& t1, const TrackIf& t2)

**SVertex** (const TrackIf& t1, const TrackIf& t2, const TrackIf& t3)

**SVertex** (const TrackIf& t1, const TrackIf& t2, const TrackIf& t3, const TrackIf& t4)

**SVertex** (const SVertex<NTR-1>& vtx, const TrackIf& t)
*add a track*

template <unsigned int NTR2> SVertex <NTR> &
**operator=** (const SVertex<NTR2>& rhs)
*copy contents from other Vertex object*

**— static variables —**

static const unsigned int
**size**

**— VertexIf Access methods —**

| | | |
|---|---|---|
| float | **vx** () const | *vertex $x$ position* |
| float | **vy** () const | *vertex $y$ position* |
| float | **vz** () const | *vertex $z$ position* |
| float | **chi2** () const | *vertex $\chi^2$* |
| unsigned short int | **ndf** () const | *degrees of freedom of vertex fit* |
| float | **prob** () const | *upper tail $\chi^2$ probability* |
| unsigned short int | **ntracks** () const | *no of tracks in vertex* |
| const TrackIf* | | |

**track** (unsigned int i) const
*read only track access*

const TrackIf*&
**track** (unsigned int i)
*read/write track access*

SVector <double, 3>
**vpos** () const $\vec{v} = (x,\ y,\ z)$

const SMatrix <double, 3> &
**VCOV** () const *vertex cov. matrix*

float **vtx_cov_x** () const
$\sigma_x^2$ *of vertex*

float **vtx_cov_y** () const
$\sigma_y^2$ *of vertex*

float **vtx_cov_z** () const
$\sigma_z^2$ *of vertex*

**— SVertex members —**

const SVector <double, 3> &
**vposR** () const $\vec{x} = (x,\ y,\ z)$ *(fast readonly access)*

SVector <double, 3> &
**vposR** () $\vec{x} = (x,\ y,\ z)$ *(fast read/write access)*

void **set_track** (unsigned int i, const TrackIf& t)
*set track list*

const SKalman <NTR> &
**kalman** (unsigned int i) const
*read only access to Kalman objects*

SKalman <NTR> &
**kalman** (unsigned int i)
*read/write access to Kalman objects*

const SMatrix <double, 3> &
**VCINV** () const *inverse vertex cov. matrix*

const SVector <double, 6> &
**mother** () const *mother track parameters*

const SMatrix <double, 3*NTR+3> &

| | | |
|---|---|---|
| | **covn** () const | |
| | **— Vertex computations —** | |
| SVector <double, 3> | **findVertex2D** () const | *return 2D vertex* |
| void | **calcVertex2D** () | *set 2D vertex* |
| bool | **findVertex3D** (SVector<double, 3>& pos) const | *return 3D vertex* |
| bool | **findVertexVt** () | *Vt based kalman filter vertex fit* |
| bool | **findVertexNe** () | *Vt based kalman filter vertex fit, without first estimation* |
| SVector <double, 2> | **rmsDistAngle** () const | |
| SVector <double, 3> | **EstimateVertex** () const | *weighted 2D vertex* |
| double | **mass** (const SVector<double, NTR>& rm) const | *compute invariant mass using refitted track momenta* |
| double | **massError** (const SVector<double, NTR>& rm) const | *compute error invariant mass cause by momentum error* |
| double | **mass_tr** (const SVector<double, NTR>& rm) const | *compute invariant mass using track momenta* |
| | **— TrackIf Access methods —** | |
| float | **x** () const | *$x$ of reconstructed track* |
| float | **y** () const | *$y$ of reconstructed track* |
| float | **x** (float z) const | *linear extrapolation of $x$ to z* |
| float | **y** (float z) const | *linear extrapolation of $y$ to z* |
| float | **z** () const | *$z$ of reconstructed track* |
| float | **tx** () const | *Track slope $t_x$* |
| float | **ty** () const | *Track slope $t_y$* |

| | | |
|---|---|---|
| float | **p** () const | *Track momentum* $p$ |
| float | **pt** () const | *transverse Track momentum* $p_t$ |
| float | **pz** () const | *momentum along* $z$*:* $p/\sqrt{t_x^2+t_y^2+1}$ |
| float | **phi** () const | *azimuthal angle* $\phi$ *[deg]* |
| float | **theta** () const | *polar angle* $\theta = \cos^{-1}(e_z)$ *[deg]* |
| float | **eta** () const | *rapidity* $\eta = -\log\tan(\theta/2.)$ |
| int | **charge** () const | *track charge (=sum of track charges in vertex)* |
| float | **energy** (double mass = 0.) const | $E = \sqrt{m^2+p^2}$ |
| float | **xf** (double mass = 0.) const | $x_f = (p_z - E(m) \cdot \sqrt{1 - \frac{2m_p}{920GeV/c+m_p}})/m_p$, $m_p$ *= Proton mass,* $m$ *= Particle mass* |
| float | **rap** (double mass = 0.) const | *Rapidity* $y = \frac{1}{2}\ln(\frac{E+p_z}{E-p_z})$ |
| float | **cov_x** (double dz=0.) const | *get* $\sigma_x^2\|_{z+dz}$ |
| float | **cov_y** (double dz=0.) const | *get* $\sigma_y^2\|_{z+dz}$ |
| float | **cov_tx** () const | *get* $\sigma_{t_x}$ |
| float | **cov_ty** () const | *get* $\sigma_{t_y}$ |
| float | **cov_p** () const | *get* $\sigma_p^2$ |
| bool | **isValid** () const | *is track valid? valid=true only if findVertexVt() succeeded* |
| | **— Operators —** | |
| bool | **operator==** (const TrackIf& rhs) const | *compare TrackIf pointers* |
| | **— Expert methods —** | |
| void | **valid** () | *mark vertex as valid* |
| void | **invalid** () | *mark vertex as invalid* |

const SMatrix <double, 5> &

| | | |
|---|---|---|
| | **COV** () const | *return track cov. matrix* |
| const SMatrix <double, 5> & | **CINV** () const | *return inverse track cov. matrix* |
| SVector <double, 3> | **xvec** () const | $\vec{x} = (x,\ y,\ z)$ |
| SVector <double, 3> | **tvec** () const | $\vec{t} = (t_x,\ t_y,\ 1)$ |
| SVector <double, 3> | **evec** () const | $\vec{e} = (e_x,\ e_y,\ ez)$ *unit vector along* $\vec{p}$ |
| SVector <double, 3> | **pvec** () const | $\vec{p} = (p_x,\ p_y,\ p_z)$ |
| | **— Expert methods —** | |
| bool | **filter** () | *do a filter step* |
| bool | **smooth** () | *smooth vertex parameters* |
| bool | **smoothC** () | *needed for bigcov()* |
| inline double | **bk13** () const | |
| bool | **calc_mother** () | *calculate mother track + cov. matrix* |
| bool | **calc_mother_tr** () | *calculate mother track using kalman info* |
| bool | **calc_mother_trtr** () | *calculate mother track using track info* |
| bool | **calc_mother_cov** () | *calculate covariance matrix of mother track* |
| bool | **bigcov** () | *construct* $(3 \cdot n + 3) \times (3 \cdot n + 3)$ *cov. matrix* |
| void | **collect** (vector<TrackIf*>& c) const | *collect pointers* |
| void | **propagate** (const double z) | *propagate reconsructed track to* $z$ |
| std::ostream& | | |

**print** ( std::ostream& ) const
*used by operator<<()*

Fixed size Vertex class.

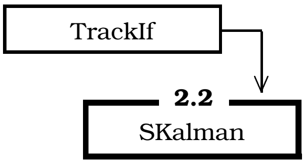
**Author:** T. Glebe

## 2.2

template <unsigned int NTR> class **SKalman** : public TrackIf

*Fixed size Kalman filter class*

### Inheritance




### Public Members

**— Constructors —**

**SKalman** ()

**SKalman** (const TrackIf& t,
const SVertex<NTR>& v)

**— TrackIf methods —**

| | | |
|---|---|---|
| float | **x** () const | *refitted Track $x$ position (= vertex $x$ position)* |
| float | **y** () const | *refitted Track $y$ position (= vertex $y$ position)* |
| float | **x** (float z) const | *refitted Track $x$ position at $z$* |
| float | **y** (float z) const | *refitted Track $y$ position at $z$* |

| | | |
|---|---|---|
| float | **z** () const | *refitted Track $z$ position (= vertex $z$ position)* |
| float | **tx** () const | *refitted track slope $t_x$* |
| float | **ty** () const | *refitted track slope $t_y$* |
| float | **p** () const | *refitted momentum* |
| float | **pt** () const | *refitted transverse Track momentum $p_t$* |
| float | **pz** () const | *momentum along $z$: $p/\sqrt{t_x^2+t_y^2+1}$* |
| float | **chi2** () const | *Kalman $\chi^2$. Use only if you know what it is!* |
| float | **phi** () const | *refitted azimuthal angle $\phi$ [deg]* |
| float | **theta** () const | *refitted polar angle $\theta = \cos^{-1}(e_z)$ [deg]* |
| float | **eta** () const | *refitted rapidity $\eta = -\log\tan(\theta/2.)$* |
| int | **charge** () const | *particle charge: -1 for neg. +1 for pos. & neutrals* |
| float | **energy** (double mass = 0.) const | $E=\sqrt{m^2+p^2}$ |
| float | **xf** (double mass = 0.) const | $x_f = (p_z - E(m)\cdot\sqrt{1-\frac{2m_p}{920GeV/c+m_p}})/m_p$, $m_p$ = *Proton mass*, $m$ = *Particle mass* |
| float | **rap** (double mass = 0.) const | *Rapidity* $y=\frac{1}{2}\ln(\frac{E+p_z}{E-p_z})$ |
| SVector <double, 3> | **xvec** () const | *vertex position* |
| SVector <double, 3> | **tvec** () const | $\vec{v} = (t_x,\ t_y,\ 1.)$ *refitted slope vector* |
| SVector <double, 3> | **evec** () const | $\vec{v} = (e_x,\ e_y,\ e_z)$ *unit vector along refitted track* |
| SVector <double, 3> | | |

| | | |
|---|---|---|
| | **pvec** () const | $\vec{v} = (p_x,\ p_y,\ p_z)$ *refitted mom. vector* |
| void | **collect** (vector<TrackIf*>& c) const | *collect pointers* |
| | **— Dummy TrackIf functions —** | |
| bool | **isValid** () const | *returns always true* |
| void | **valid** () | *does nothing* |
| void | **invalid** () | *does nothing* |
| const SMatrix <double, 5> & | **COV** () const | *returns a 0 matrix* |
| const SMatrix <double, 5> & | **CINV** () const | *returns a 0 matrix* |
| void | **propagate** (const double dz) | *dummy function: do nothing* |
| unsigned short int | **ndf** () const | *returns always 0* |
| float | **cov_x** (double dz=0.) const | *returns 0* |
| float | **cov_y** (double dz=0.) const | *returns 0* |
| float | **cov_tx** () const | *returns 0* |
| float | **cov_ty** () const | *returns 0* |
| float | **cov_p** () const | *returns 0* |
| | **— Operators —** | |
| bool | **operator==** (const TrackIf& rhs) const | *compare TrackIf pointers* |
| | **— Access methods —** | |
| const SMatrix <double, 3> & | **KCOV** () const | |
| const SMatrix <double, 3> & | **KCINV** () const | |
| const SVector <double, 3> & | **xv** () const | |
| const SMatrix <double, 3> & | **F** () const | |
| const SMatrix <double, 3> & | | |

| | | |
|---|---|---|
| | **ES** () const | |
| const SMatrix <double, 3> & | **DS** () const | |
| | **— Expert methods —** | |
| bool | **filter** (const unsigned int I) | *filter step* |
| bool | **smooth** () | *smoother step* |
| bool | **smoothC** () | *cov. matrices of smoothed parameters* |
| std::ostream& | **print** ( std::ostream& ) const | *used by operator<<()* |

Fixed size Kalman filter class.

**Author:** T. Glebe

# 3 SDistance

*Distance functions for Track and Vertex objects.*

**Names**

**Author:** T. Glebe, MPI fuer Kernphysik Heidelberg

**3.1**

inline double **Sdistance** (const SVector<double, 3>& x1, const SVector<double, 3>& x2)

*Spatial distance between two space points*

Spatial distance between two space points.

$$d = |\vec{x}_1 - \vec{x}_2|$$

**3.2**

inline double **Sdistance** (const VertexIf& v1, const VertexIf& v2)

*Spatial distance between two VertexIf objects*

Spatial distance between two VertexIf objects.

$$d = |\vec{v}_1 - \vec{v}_2|$$

**3.3**

inline double **distance** (const VertexIf& v, const SVector<double, 3>& x)

*Spatial distance between Vertex and space point*

Spatial distance between Vertex and space point.

$$d = |\vec{x}_v - \vec{x}|$$

**3.4**

inline double **distance** (const TrackIf& t, const VertexIf& v)

*Clostest spatial distance between Track and Vertex object*

Clostest spatial distance between Track and Vertex object.

$$d = \frac{|(\vec{x}_t - \vec{x}_v) \times \vec{t}|}{|\vec{t}|}$$

$g : \vec{x}_t + m \cdot \vec{t}$ track definition
$\vec{x}_v$: vertex position

**3.5**

double **Tdistance** (const TrackIf& t1, const TrackIf& t2)

*Clostest spatial distance between two Track objects*

Clostest spatial distance between two Track objects.

$$d\vec{x} = \vec{x}_2 - \vec{x}_1$$
$$f_1 = d\vec{x} \cdot \frac{\vec{t}_1}{|\vec{t}_1|}$$
$$f_2 = d\vec{x} \cdot \frac{\vec{t}_2}{|\vec{t}_2|}$$
$$d = \left| d\vec{x} + f_2 \cdot \frac{\vec{t}_2}{|\vec{t}_2|} - f_1 \cdot \frac{\vec{t}_1}{|\vec{t}_1|} \right|$$

$g : \vec{x}_1 + m \cdot \vec{t}_1$ track definition 1st track
$g : \vec{x}_2 + m' \cdot \vec{t}_2$ track definition 2nd track

**3.6**

inline double **distance** (const TrackIf& t, const Wire& w)

*Closest spatial distance between Track and Wire*

Closest spatial distance between Track and Wire.

$$d = \frac{(\vec{t}_w \times \vec{t}_t) \cdot (\vec{x}_t - \vec{x}_w)}{|\vec{t}_w \times \vec{t}_t|^2}$$

$g : \vec{x}_t + m \cdot \vec{t}_t$ track definition
$w : \vec{x}_w + m' \cdot \vec{t}_w$ wire definition

**3.7**

double **SGNdistance** (const TrackIf& t, const VertexIf& v)

*Signed closest distance between Track and Vertex object*

Signed closest distance between Track and Vertex object. The sign is determined by the z component and tells whether the point of closest approach is upstream or downstream of the vertex position.

$$\rho = \frac{\vec{t} \cdot (\vec{x}_t - \vec{x}_v)}{|\vec{t}|}$$

$$d\vec{x} = \vec{x}_t - \rho \cdot \vec{t} - \vec{x}_v$$

$$d = \mathrm{sgn}(d\vec{x}_z) \cdot |d\vec{x}|$$

$g : \vec{x}_t + m \cdot \vec{t}$ track definition
$\vec{x}_v$: vertex position
$\vec{x}_t - \rho \cdot \vec{t}$: point of track closest to vertex

**3.8**

inline double **Chi2distance** (const TrackIf& t1, const TrackIf& t2)

*$\chi^2$ distance between two tracks*

$\chi^2$ distance between two tracks. Determined by Kalman filter vertex fit.

**3.9**

template <unsigned int NTR> inline double **Chi2distance** (const SVertex<NTR>& v, const SVector<double, 3>& x)

*$\chi^2$ distance between SVertex and space point*

$\chi^2$ distance between SVertex and space point.
Assumption: SVertex has been fitted with findVertexVt().

**3.10**

template <unsigned int NTR, unsigned int NTR2> inline double **Chi2distance** (const SVertex<NTR>& v1, const SVertex<NTR2>& v2)

*$\chi^2$ distance between two SVertex objects*

$\chi^2$ distance between two SVertex objects.
Assumption: Both SVertex objects have been fitted with findVertexVt().

# Class Graph