

SMatrix - A high performance library for Vector/Matrix calculation and Vertexing

Version 0-20

HERA-B Note 01-134, Software 01-017 (UPDATED)

Thorsten Glebe
Max-Planck-Institut für Kernphysik, Heidelberg
T.Glebe@mpi-hd.mpg.de

December 2, 2003

Abstract

SMatrix is a C++ based library for high performance vector and matrix computations. In addition the **SVertex** class is provided, which is a reimplementation of the **Vt++** library using the high performance vector and matrix classes. With **SVertex** objects, Kalman filter based vertex fits can be performed 20 times faster than with **Vt++** [1] and 5-10 times faster than with the **Grover** [2] vertexing package. The **SMatrix** library is part of the **BEE** analysis framework [3].

Contents

1	Introduction	1
2	Vectors	2
3	Matrix	4
3.1	Matrix - Matrix expressions	5
3.2	Vector - Matrix expressions	6
3.3	Matrix member functions	7
4	Performance comparison	8
5	Vertexing	9
5.1	Track and Vertex abstraction	9
5.2	Vertex fit methods	10
5.3	Computing the mother track	10
5.4	Accessing track and Kalman information	11
5.5	Mass calculation	11
5.6	Performance comparison to Vt++	12
6	Extending the SMatrix package	12
7	Remarks	13

1 Introduction

One of the main tasks of data analysis code in high energy physics experiments is to reconstruct decay chains and to compute properties of particles from a huge amount of measurements. In many cases the computations involve vectors (for e.g. track momentum or vertex position) and matrices (e.g. covariance matrices in a Kalman filter algorithm). Vector and matrix algebra can be perfectly implemented with an object oriented language like C++, which allows with its abstraction capabilities

to arrive at simple and readable code. In recent years there has been a proliferation of libraries which provides abstractions for a wide range of numerical problems.

However, the code generated by such libraries tends to be naive. For example, array objects implemented using operator overloading in C++ were originally 3-20 times slower than the corresponding low-level implementation. This was not because of poor design on the part of library developers, but rather because the language forced a style of implementation which was grossly inefficient. These performance problems are commonly called the *abstraction penalty* [4].

A promising approach to overcome the described problem is to construct libraries which provides both the abstractions, and control how they are optimized. This concept has been called *active libraries* [5]. `SMatrix` is such a library, it handles high-level optimizations themselves and leaves only low-level optimization (register allocation, instruction scheduling, software pipelining) to the optimizer.

The introduction of templates to C++ added a facility whereby the compiler can act as an interpreter. This makes it possible to write programs in a subset of C++ which are interpreted at compile time. This technique is called *template metaprogramming* [6] and is used to achieve the above mentioned high-level optimizations. C++ templates are a programming language by its own and can be used to implement vector and matrix expressions such that these expressions can be transformed at compile time to code which is equivalent to hand optimized code in a low-level language like FORTRAN or C [7, 8].

The `SMatrix` library has been written to exploit the template techniques for a high performance implementation of a Kalman filter algorithm for vertex fitting. Investigations of typical analysis jobs have shown that this is the place where 70-90% of the computing time is spent.

The `SMatrix` library is fully templated, that means it is sufficient to include the appropriate header files. No library to link with is needed.

2 Vectors

The `SVector` class represents n -dimensional vectors for objects of arbitrary type. The type (usually `float`, `double` or `int`) as well as the vector dimension must be given as a template argument:

```
#include <iostream>
#include "SVector.hh"

int main() {

    SVector<float,3> x(1,2,3); // 3-dimensional
    SVector<float,3> y(4,5,6); // 3-dimensional

    cout << "x + y: " << x + y << endl;

    return 0;
}
```

The result is, as expected:

```
x + y: 5, 7, 9
```

The `[]` and `()` operators can be used to access the individual elements of a vector:

```
SVector<float,3> a;
a[0] = 1.; a[1] = 2.; a[2] = 3.;
cout << "Elements: " << a[0] << ", " << a(1) << ", " << a[2]
    << endl;
```

Defined operators are `+`, `-` (both binary and unary), `*`, `/`, `+=`, `-=`, `*=`, `/=`, `fabs` (absolute values), `sqr` (square) and `sqrt` (square root):

```
SVector<float,3>    x(1,2,3);
SVector<float,3>    y(4,5,6);
SVector<float,3>    z(4,16,64);

// element wise multiplication
cout << "x * y: " << x * -y << endl;
```

```

x += z - y;
cout << "x += z - y: " << x << endl;

// element wise square root
cout << "sqrt(z): " << sqrt(z) << endl;

// element wise multiplication with constant
cout << "2 * y: " << 2 * y << endl;

// a more complex expression
cout << "fabs(-z + 3*x): " << fabs(-z + 3*x) << endl;

```

The output is:

```

x * y: -4, -10, -18
x += z - y: 1, 13, 61
sqrt(z): 2, 4, 8
2 * y: 8, 10, 12
fabs(-z + 3*x): 1, 23, 119

```

Special operators are: `dot` (scalar product), `mag2` (squared norm), `mag` (norm), `cross` (cross product) and `unit` (return vector of length 1):

```

SVector<float,3>    x(1,2,3);
SVector<float,3>    y(4,5,6);

// scalar product
cout << "dot(x,y): " << dot(x,y) << endl;

// norm of x
cout << "mag(x): " << mag(x) << endl;

// cross product of x and y
cout << "cross(x,y): " << cross(x,y) << endl;

// copy of x, rescaled to length = 1
cout << "unit(x): " << unit(x) << endl;

```

The output is:

```

dot(x,y): 32
mag(x): 3.74166
cross(x,y): -3, 6, -3
unit(x): 0.267261, 0.534522, 0.801784

```

Note: the cross product works only for vectors with dimension $n = 3$.

A vector \vec{a} can be placed in another vector \vec{b} via the `place_at` member function. For the vector dimensions the condition $\dim(\vec{b}) \geq \dim(\vec{a})$ must hold.

```

SVector<float,4> a;
SVector<float,2> b(1,2);

a.place_at(b,2);
cout << "a: " << a << endl;

```

The output is:

```

a: 0, 0, 1, 2

```

A big advantage of coding the vector dimension as a template parameter is that the correctness of dimensions in expressions can be checked already at compile time. In case of vector/matrix classes with dynamic dimensions the correctness of expressions can only be detected during run time¹.

¹In the best case a segmentation violation occurs, in the worst case the result is just wrong.

3 Matrix

The `SMatrix` class represents matrices with $n \times m$ dimensions of arbitrary type. Again, the type as well as the matrix dimensions must be specified as template arguments:

```
#include <iostream>
#include <SMatrix>

int main() {

    SMatrix<float,2,3> A = 15.; // 2x3 matrix

    cout << "A: " << endl << A << endl;

    return 0;
}
```

The output is:

```
A:
[      15      15      15
      15      15      15 ]
```

In case of an $n \times n$ matrix it is sufficient to specify the dimension only once:

```
SMatrix<float,3> A = 15.; // 3x3 matrix

cout << "A: " << endl << A << endl;
```

The output is:

```
A:
[      15      15      15
      15      15      15
      15      15      15 ]
```

Matrix elements can be accessed via the `()` operator:

```
SMatrix<float,2,3> A = 15.; // 2x3 matrix
A(0,0) = A(1,1) = A(0,2) = 5.;
```

The output is:

```
A:
[      5      15      5
      15      5      15 ]
```

Note: there is no check whether the indices point to a valid matrix element. The `SMatrix` class is designed for high performance and not for fool-proofness.

A row or a column of a `SMatrix` object can be returned as a `SVector` object:

```
SMatrix<double,3> A;
A(0,0) = A(1,0) = 1;
A(0,1) = 3;
A(1,1) = A(2,2) = 2;
cout << "A: " << endl << A << endl;

// 1st row
SVector<double,3> x = A.row(0);
cout << "x: " << x << endl;

// 2nd column
SVector<double,3> y = A.col(1);
cout << "y: " << y << endl;
```

The output is:

```

A:
[
    1      3      0
    1      2      0
    0      0      2 ]
x: 1, 3, 0
y: 3, 2, 0

```

Vector or vector expressions can be placed inside a matrix. This is useful if you want to compose matrices from different (sub-) expressions. The `place_in_row` member function places vectors in matrix rows, `place_in_col` places vectors in matrix columns. A smaller matrix can be placed inside a bigger matrix via the `place_at` member function:

```

SVector<float,3> x(4,5,6);
SVector<float,2> y(2,3);
cout << "x: " << x << endl;
cout << "y: " << y << endl;

SMatrix<float,4,3> A;
SMatrix<float,2,2> B = 1;

A.place_in_row(y, 1, 1);
A.place_in_col(x + 2, 1, 0);
A.place_at(B * -4, 2, 1);
cout << "A: " << endl << A << endl;

```

The output is:

```

x: 4, 5, 6
y: 2, 3
A:
[
    0      0      0
    6      2      3
    7     -4     -4
    8     -4     -4 ]

```

3.1 Matrix - Matrix expressions

Defined operators are `+`, `-` (both binary and unary), `times`, `/`, `+=`, `-=`, `*=`, `/=`, `fabs` (absolute values), `sqr` (square) and `sqrt` (square root). All these operators work element wise (are applied to each element of the matrix):

```

SMatrix<float,2,3> x = 4.;
SMatrix<float,2,3> y = 5.;
SMatrix<float,2,3> z = 64.;

// element wise multiplication
cout << "x * y: " << endl << times(x, -y) << endl;

x += z - y;
cout << "x += z - y: " << endl << x << endl;

// element wise square root
cout << "sqrt(z): " << endl << sqrt(z) << endl;

// element wise multiplication with constant
cout << "2 * y: " << endl << 2 * y << endl;

// a more complex expression
cout << "fabs(-z + 3*x): " << endl << fabs(-z + 3*x) << endl;

```

The output is:

```

x * y:
[
    -20     -20     -20
    -20     -20     -20 ]

```

```

x += z - y:
[      63      63      63
  63      63      63 ]

sqrt(z):
[      8      8      8
  8      8      8 ]

2 * y:
[     10     10     10
  10     10     10 ]

fabs(-z + 3*x):
[     125     125     125
  125     125     125 ]

```

Matrix multiplication is done via the * operator:

```

SMatrix<float,4,2> A;
A(0,0) = A(0,1) = A(1,1) = A(2,0) = A(3,1) = 4.;
cout << "A: " << endl << A << endl;

SMatrix<float,2,3> S;
S(0,0) = S(0,1) = S(1,1) = S(0,2) = 1.;
cout << " S: " << endl << S << endl;

// matrix multiplication
SMatrix<float,4,3> C = A * S;
cout << " C: " << endl << C << endl;

```

The output is:

```

A:
[      4      4
  0      4
  4      0
  0      4 ]

S:
[      1      1      1
  0      1      0 ]

C:
[      4      8      4
  0      4      0
  4      4      4
  0      4      0 ]

```

3.2 Vector - Matrix expressions

Expressions including `SMatrix` and `SVector` objects are allowed, too. The matrix - vector multiplication is implemented via the * operator:

```

SMatrix<float,4,3> A;
A(0,0) = A(0,1) = A(1,1) = A(2,2) = 4.;
A(2,3) = 1.;
cout << "A: " << endl << A << endl;
SVector<float,4> x(1,2,3,4);
cout << " x: " << x << endl;
SVector<float,3> a(1,2,3);
cout << " a: " << a << endl;

// Matrix-Vector expression
SVector<float,4> y = x + A * a;
cout << " y: " << y << endl;

```

The output is:

```

A:
[
    4      4      0
    0      4      0
    0      0      4
    1      0      0 ]
x: 1, 2, 3, 4
a: 1, 2, 3
y: 13, 10, 15, 5

```

The operators are able to work with expressions, too (take above definitions of matrix A and vector x):

```

// we add 1 to each component of x and A
SVector<float,3> b = (x+1) * (A+1);
cout << " b: " << b << endl;

```

The output is:

```
b: 27, 34, 30
```

A special operator is the `product` operator, which calculates expressions of type $\alpha = \vec{a}^T \cdot A \cdot \vec{a}$:

```

SMatrix<double,3> A;
A(0,0) = A(0,1) = A(1,0) = 1;
A(1,1) = A(2,2) = 2;
cout << " A: " << endl << A << endl;

SVector<double,3> x(1,2,3);
cout << "x: " << x << endl;

// we add 1 to each component of x and A
cout << " (x+1)^T * (A+1) * (x+1): " << product(x+1,A+1) << endl;

```

The output is:

```

A:
[
    1      1      0
    1      2      0
    0      0      2 ]
x: 1, 2, 3
(x+1)^T * (A+1) * (x+1): 147

```

3.3 Matrix member functions

The following `SMatrix` member functions for special calculations are provided:

`det` calculate determinant of a square matrix

`sdet` calculate determinant of a symmetric, positive definite matrix

`invert` invert a square matrix

`sinvert` invert a symmetric, positive definite matrix

Note: all these member functions will modify the contents of the matrix. In case of `invert()` and `sinvert()` the matrix is transformed into its inverse. Example:

```

SMatrix<double,3> A;
A(0,0) = A(0,1) = A(1,0) = 1;
A(1,1) = A(2,2) = 2;
cout << "A: " << endl << A << endl;

double det = 0.;
// determinant of symmetric, pos. def. matrix
A.sdet(det);
cout << "Determinant: " << det << endl;

// WARNING: A has changed!!
cout << "A again: " << endl << A << endl;

```

The output is:

```
A:
[      1      1      0
      1      2      0
      0      0      2 ]
Determinant: 2
A again:
[      1      1      0
      1      1      0
      0      0      0.5 ]
```

The `det()` and `sdet()` functions are derived from the CERNLIB functions `dfact` and `dsfact`[10].

Example for matrix inversion:

```
SMatrix<double,3> A;
A(0,0) = A(0,1) = A(1,0) = 1;
A(1,1) = A(2,2) = 2;

SMatrix<double,3> B = A; // save A in B
cout << "A: " << endl << A << endl;

A.sinvert();
cout << "A^-1: " << endl << A << endl;

// check if this is really the inverse:
cout << "A^-1 * B: " << endl << A * B << endl;
```

The output is:

```
A:
[      1      1      0
      1      2      0
      0      0      2 ]
A^-1:
[      2     -1      0
     -1      1      0
      0      0      0.5 ]
A^-1 * B:
[      1      0      0
      0      1      0
      0      0      1 ]
```

The `invert()` and `sinvert()` functions are derived from the CERNLIB `dinv` and `dsinv` routines.

4 Performance comparison

A performance comparison between the matrix and vector classes of `Vt++` and `SMatrix` has been carried out. The `Vt++` implementation of matrix and vector classes is suffering from the C++ abstraction penalty (see section 1), from dynamic storage allocation and from run time dimension evaluation. So the comparison shows nicely the impact of the template based implementation of `SMatrix` on the numerical performance.

Note: the performance depends on the dimension of vectors and matrices. For small dimensions the impact of loop overheads and storage allocation is dominant, whereas for large dimensions the vector or matrix computation will dominate the processing time. As a Kalman filter based vertex fit is based on low-dimensional vectors and matrices, a low dimension has been chosen for the comparison.

Conditions:

1. Intel Celeron 466 MHz, 256 MB RAM
2. Compiler: g++ 2.95.2
3. Compiler flags: g++ -O2 -funroll-loops -finline-functions

Operation ($\times 10^6$), dim = 3	t_{Vt} [s]	t_{SM} [s]	t_{Vt}/t_{SM}
dsinv() (CERNLIB, A^{-1})	3.73	0.84	4.44
dsfact() (CERNLIB, det A)	2.98	0.53	5.62
$C = A \cdot B$	8.09	1.29	6.27
$D = A + B \cdot C$	12.44	1.43	8.70
$\vec{y} = A \cdot \vec{x}$	4.40	0.28	15.71
$D = A + B + C$	11.29	0.43	26.25
$D = -A + B + C$	13.27	0.47	28.23
$a = \vec{x}^T \cdot A \cdot \vec{x}$	2.56	0.07	36.57
$a = (\vec{x} + \vec{y})^T \cdot (A + B + C) \cdot (\vec{x} + \vec{y})$	18.27	0.39	46.85
-funroll-loops	+3%	+30%	

Note: t_{Vt} denotes the time consumption of the Vt++ code and t_{SM} the time consumption of the SMatrix code. The third column shows the ratio. As one can see, loop unrolling on the compiler level (initiated by the `-funroll-loops` compiler flag) is much more efficient in case of the SMatrix library than for Vt++. The reason for this is explained in section 1.

5 Vertexing

The SVertex class is a reimplementaion of the Vt++ Vertex class, based on the high performance vector and matrix classes SVector and SMatrix. The SKalman class represents a track in the Kalman filter process. The SVertex class implements a fixed sized vertex object, that means the number of tracks in the vertex must be known already at compile time. This is usually not a restriction for an analysis job, as the signature of decaying particles is known in advance. The only limitation comes from primary vertices, where the number of tracks depend on the data and is not known at compile time.

5.1 Track and Vertex abstraction

The SVertex class is based on an abstraction of tracks and vertices and thus allows to be interfaced to an arbitrary amount of (non curved) track and vertex classes without changing a single line of SVertex code. The interface classes can be found in the following header files of the BEE interfaces project:

TrackIf.hh interface class for non curved tracks

VertexIf.hh interface class for vertices.

In other words: the SVertex class can work with any track class which is inherited from the TrackIf interface class:

```
#include "TrackIf.hh"

class Line: public TrackIf {
public:
    Line(double x, double y, ...);

    float x() const { return x_; };
    float y() const { return y_; };
    ...
private:
    double x_, y_,...;
}
```

The newly defined Line class has now the TrackIf interface and can be used by SVertex :

```
#include "Line.hh"
#include "SVertex.hh"

int main() {

    Line a(...);
    Line b(...);
```

```
SVertex<2> vtx(a,b);
}
```

The `SVertex` class is inherited from the `VertexIf` interface, thus allowing you to write code which is independent from the exact type of the `SVertex` class and can be used for any class inherited from `VertexIf`:

```
class VertexIf;

// function to return spatial distance between
// VertexIf objects
inline double Sdistance(const VertexIf& v1, const VertexIf& v2) {
    return mag(v1.vpos() - v2.vpos());
}
```

Example:

```
int main() {

    SVertex<2> vtx1(...);
    SVertex<4> vtx2(...);

    vtx1.findVertexVt(); // fit vertex
    vtx2.findVertexVt(); // fit vertex

    // compute spatial distance
    double dist = Sdistance(vtx1,vtx2);
}
```

Note: for a list of defined χ^2 -distance and spatial distance functions between vertex, track and wire objects, see the reference manual or the `SDistance.hh` header file.

5.2 Vertex fit methods

Four different methods to compute a vertex position are provided:

`findVertexVt` Kalman filter based vertex fit based on T. Lohses algorithm [9]. **Note:** before using this routine, make sure that the tracks are propagated to $z = 0$.

`findVertex2D` analytical vertex computation in 2 dimensions. Returns the vertex position but doesn't change the internal state of the `SVertex` object. The `calcVertex2D()` member function computes the vertex and sets the vertex position.

`findVertex3D` analytical vertex computation in 3 dimensions. Returns the result in a `SVector` object which must be given as an argument. It does not change the internal state of the `SVertex` object.

`EstimateVertex` analytical vertex computation in 2 dimensions by using covariance matrix elements as weights. Returns result in a `SVector` object. The internal state of the `SVertex` object is not changed.

In general, a vertex position can be set by hand using the `set_vpos()` member function:

```
SVertex<double,2> vtx(...);
vtx.set_vpos(vtx.EstimateVertex());
```

Note: the covariance matrix of the vertex is only computed with the Kalman filter based vertex fit.

5.3 Computing the mother track

The `SVertex` class is inherited from the `TrackIf` class and is thus able to represent the mother track², which can be computed from the vertex position and the momentum sum of the outgoing tracks. For performance reasons the mother track is never computed implicitly, but only if it is explicitly wanted by the user. The `SVertex` class is designed for high performance and not for fool-proofness, therefore the user has to make sure that the mother track has been calculated before accessing the `TrackIf` member functions of an `SVertex` object.

The following member functions exist to calculate the mother track:

²The incoming track is usually called *mother track*.

`calc_mother_tr()` calculate the track parameters (t_x, t_y, p) of the mother track by using the Kalman information of the tracks in the vertex. **Note:** the vertex must have been fitted with `findVertexVt()` when using this function.

`calc_mother_cov()` calculate the covariance matrix of the mother track. **Note:** the vertex must have been fitted with `findVertexVt()` when using this function.

`calc_mother()` same as calling both member functions `calc_mother_tr()` and `calc_mother_cov()`.

`calc_mother_trtr()` calculate the track parameters of the mother track by using the information from the track objects. No Kalman information is used, the vertex can be fitted with any routine (or set by hand).

A `SVector` with the track parameters x, y, z, t_x, t_y, p is returned by the `mother()` member function.

Example:

```
SVertex<double,2> vtx1(...);

// if fit is OK
if(vtx1.findVertexVt() == true) {
    vtx1.calc_mother(); // compute mother track
    cout << "mother track: " << vtx1.mother() << endl;
}
```

5.4 Accessing track and Kalman information

The `TrackIf` and `SKalman` information of tracks fitted to a vertex can be accessed via the `track()` and `kalman()` information. The track in question is specified by an index:

```
SVertex<double,4> vtx(...); // create vertex object

// fit vertex
if(vtx.findVertexVt() == true) {

    // get 3rd track
    const TrackIf* t3 = vtx.track(2);
    // get Kalman info of 3rd track
    const SKalman& k3 = vtx.kalman(2);

    // print Kalman information
    cout << "Kalman: " << k3 << endl;
}
```

The `SKalman` class is inherited from `TrackIf`, too. Therefore the Kalman filter objects represent the refitted tracks and can be used in the same way as any other `TrackIf` based object. **Note:** some of the `SKalman` functions are dummy functions in order to comply with the `TrackIf` interface. See the reference manual for more details.

Tracks can be set not only by calling the `SVertex` constructor, but also later:

```
SVertex<double,2> vtx(); // default CTOR

TrackIf& t1 = ...; // get a track from somewhere
TrackIf& t2 = ...; // get a track from somewhere

vtx.track(0) = &t1; // set track pointer
vtx.track(1) = &t2; // set track pointer
```

5.5 Mass calculation

Two member functions exist to compute an invariant mass of the vertex object and one to compute an error of the invariant mass:

`mass` computes the invariant mass using the Kalman filter information from the refitted track objects.

Note: the vertex must have been fitted with `findVertexVt()` when using this function.

`mass_tr` computes the invariant mass using the measured track momenta. There is no need to fit/compute the vertex when using this function.

`massError` computes an error of the invariant mass by a simplified formula which just takes the error of the refitted momentum into account. **Note:** the vertex must have been fitted with `findVertexVt()` when using this function.

In all cases, a `SVector` object which contains the (assumed) rest masses of the daughter tracks must be given as an argument. The rest mass vector must have the same dimension as the `SVertex` object:

```
SVertex<double,3> vtx(...); // vertex with 3 tracks
const double pimass = 0.134;
const double Kmass = 0.497;

if(vtx.findVertexVt() == true) {
    // 1st track Kaon, others pion
    SVector<double,3> massvec(Kmass,pimass,pimass);

    // mass computed with refitted momenta
    double mass1 = vtx.mass(massvec);
    // mass computed with measured track momenta
    double mass2 = vtx.mass_tr(massvec);
    // mass error
    double masserr = vtx.massError(massvec);

    cout << "inv. mass: " << mass1 << ", " << mass2
         << " err: " << masserr << endl;
}

```

5.6 Performance comparison to Vt++

A performance comparison between the reimplemented vertex fit algorithms in `SVertex` and in `Vt++` has been carried out. The implementations were compared on a system with the same characteristics as in section 4, the data used was from the HERA-B year 2000 run, run 17137, reprocessing 2, Cloneremove [12] used.

fit algorithm	events	t_{Vt}	t_{SM}	t_{Vt}/t_{SM}
findVertex2D	15000	17m12.01s	0m40.18s	25.68
EstimateVertex	15000	17m11.31s	0m44.86s	23.01
findVertex3D	15000	17m22.72s	0m44.70s	23.32
findVertexVt	1500	9m17.14s	0m26.11s	21.34
+ mother track	1500	10m14.87s	0m37.04s	16.60

Note: t_{Vt} denotes the time consumption of the `Vt++` code and t_{SM} the time consumption of the `SMatrix` code. The third column shows the ratio. In the last row, the Kalman filter based vertex fit including the computation of the mother track (track parameters and covariance matrix) is compared.

6 Extending the SMatrix package

Most of the binary and unary operators in the `SMatrix` library are generated automatically by using the `CreateUnaryOp.sh` and the `CreateBinaryOp.sh` shell scripts. Further operators can be defined by adding definitions at the top of the scripts in the `OPLIST` variable.

Example: In case we want to define a unary `cos()` operator it is sufficient to add the line

```
Cos,cos,cos
```

to the `OPLIST` variable in the `CreateUnaryOp.sh` script. Executing the `CreateUnaryOp.sh` script regenerates the `UnaryOperators.hh` header file with the new definitions. Now it is possible to write code which takes the cosine element wise from a vector or matrix:

```
SVector<double,3> x(0.1, 0.2, 0.3);
cout << "cos: " << cos(x) << endl;
```

For binary operators, the analog steps have to be taken, now with the `CreateBinaryOp.sh` script.

7 Remarks

1. At the moment, the **SVertex** class does not provide mass constrained vertex fits, decay chain fits and pointing constrained vertex fits. The reason is that up to now the data quality didn't allow to make use of these features in an analysis. Due to manpower constraints a data-driven software development scheme evolved in the analysis framework **BEE**.
2. The **SMatrix** project is not available in the HERA-B reconstruction framework **ARTE** [11]. It is easily possible to make use of it in **ARTE** by writing an interface class which is inherited from **TrackIf** and interfaces **RTRA** to the **SVertex** class. As already mentioned in sections 1 and 5.1, such an interface can be established without changing a single line of code in the **SMatrix** project.

References

- [1] T. Glebe, Vt++ Version 1.0, HERA-B Note 00-175, Software 00-013
- [2] <http://wwwhera-b.mppmu.mpg.de/analysis/grover.html>
- [3] <http://www.mpi-hd.mpg.de/herab/clue> or
[/afs/desy.de/group/hera-b/BEE](http://afs.desy.de/group/hera-b/BEE)
- [4] Todd L. Veldhuizen, C++ Templates as Partial Evaluation, 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99).
- [5] Todd L. Veldhuizen, Active Libraries: Rethinking the roles of compilers and libraries, SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, October 21-23, 1998.
- [6] Todd Veldhuizen and Kumaraswamy Ponnambalam, Linear algebra with C++ template metaprograms, Dr. Dobb's Journal, August 1996.
- [7] Todd Veldhuizen, Expression Templates, C++ Report, June 1995.
- [8] Todd Veldhuizen and M. Ed Jernigan, Will C++ be faster than Fortran?, Proceedings of the 1st International Scientific Computing in Object Oriented Parallel Environments (ISCOPE'97)
- [9] T. Lohse, Vertex Reconstruction And Fitting, HERA-B Note 95-013, Software 95-002.
- [10] <http://wwwinfo.cern.ch/asd/cernlib>
- [11] <http://www-hera-b.desy.de/subgroup/software>
- [12] M.-A. Pleier, Cloneremove V1.0, HERA-B Note 1-062, Software 01-009.