Vt++ Version 1.3

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

http://www.mpi-hd.mpg.de/herab/vt++

January 22, 2002

Abstract

Vt++ is the C++ reimplementation of the core part of the FORTRAN based vertex fitting package **Vt**, written by T. Lohse. It consists of the Kalman filter based vertex fit and the inverse Kalman filter. This part of **Vt** is well investigated, used for many years and known to be correct. The goal is to provide an easy-to-use vertexing class which can be used to explore different vertexing strategies and to cure limitations of the FORTRAN implementation. It is not the goal to provide 'the' final primary/secodary vertex finder, this is done by other vertexing software.

Contents

1 Introduction

Vt, the FORTRAN based vertex fitting package written by T. Lohse [1], has been used in HERA-B for more than five years and proven to be a very useful tool. Nevertheless, some disadvantages are known, too:

- As a consequence of the FORTRAN implementation, fixed buffer sizes for tracks and vertices exist, which can lead to problems while running on real data.
- Support of Vt is missing.
- The implementation has not been optimized with respect to execution speed.
- For the usage of Vt in C++ some interfaces have been written, which, however, cannot overcome the above mentioned Vt-limitations. Therefore it is not attractive to use Vt in C++ code.

From this it became clear that a C++ port of (at least the base part) of Vt would be very useful. It would make the interfaces obsolete and allow to cure the disadvantages of the FORTRAN implementation. Making use of state-of-the-art C++ coding and STL also allows to write a code which is much better to maintain and to extend. Porting Vt to C++ is easy and fast, as one can always compare with the original code.

Vt++ consists at the moment of the Kalman filter based vertex fit, the inverse Kalman filter (for track removal), the mass constrained vertex fit and some distance functions. Three simple analytical vertex computation routines are available, too. The classes can be used to write more complex data analysis routines. The documentation about Vt++ version 1.0 has been published as a HERA-B note [2].

The Vt++ key features are:

- Dynamic memory is used to store tracks and vertices - no limitations on the number of tracks or vertices does exist.
- Selecting tracks for vertexing is under user control. Vt++ can work with track objects which can be filled from RTRA, RSEG and MTRA.
- Vertex fits are not saved automatically in the RVER table - the user can decide which vertices should be kept.
- Kalman filter computation is based on a Matrix package which is independent from Vt++ and ARTE.
- Removable track-vertex object relations allow fast track-vertex assignment/removal.
- Three additional simple algorithms for vertex calculation are provided:
  1. unweighted 2-D minimal distance
  2. weighted 2-D minimal distance
  3. unweighted 3-D minimal distance
- A vertex object is a STL container for track-vertex relations. A STL-iterator is provided to loop over the relations (see examples in section 6.10).

2 Compilation and Versioning

From ARTE-03-07-r6 on Vt++ will be part of the ARTE distribution. In case you want to recompile the package, please note the following compile-flags:

-DVtDEBUG produces extensive debugging output.

-DVtFAST this compilation flag removes the `assert()` macros from the MATRIX package. This makes the code faster but also less safe. One should use this flag only if it is verified that MATRIX computation is correct. The assert statements are Heisenbug-free[1].

The Vt++ package has an individual versioning independent from the ARTE version number. The Vt++ version number plus some additional information is represented by the `VtIni` class and printed out when the VtIni object file is loaded into memory. In case of ARTE the version string is the first information which is printed by the ARTE job. For more information see section 6.2.

[1] A Heisenbug is a bug which disappears when the debugger is turned on. In case assert-macros generate side-effects they are a source of those bugs.

3 Speed

Up to now, Vt++ has not been optimized with respect to execution speed. In the current phase of development it is more important to assure correctness, flexibility and maintainability. The structure of the code will allow a detailed profiling such that considerable improvements in execution speed will be possible. See also [3]. A speedup factor in the range of 2-15 can be expected [5].

4 Vertexing

4.1 Classes

Vt++ consists of the following classes:

Vertex the vertexing class. A vertex can be constructed from **RVER**, **RTRA**, **MTRA**, **RSEG**, **Track** and **Vertex** objects. Tracks can be added via the **push_back** member function. A vertex can own tracks or use tracks from an external list of track objects. The latter is the optimal solution in case of pattern recognition algorithms. A Vertex object also represents the incoming track. In case you compute the track parameters of the incoming track, you can use Vertex objects again as an input for vertexing routines (see example 6.15).

Track the track representation class. A track object can be constructed from **RTRA**, **MTRA**, **RSEG** and **Track** objects. It stores the track parameters, the ARTE-pointers and the Track-Vertex relation. In case of a global track list this allows to loop over all vertices a track is assigned to. Relations are optimized with respect to fast addition and removal of relations. Therefore no direct access to a certain relation is possible. An emulation of this behaviour is provided, but one has to keep in mind that this is in most cases less efficient.

Kalman the Kalman mathematics class. A Kalman object is constructed by a Relation and consists of all relevant Kalman filter objects (matrices and vectors) of a track belonging to a certain vertex. For each Track-Vertex relation a new Kalman object is generated automatically. As the Kalman filter information of all filter steps is saved in this way, track removal from a vertex is a fast operation in the Kalman language.

Relation the Track-Vertex-Kalman relation class. A Relation is constructed by a Track and a Vertex object. It consists of the references to a Track, a Vertex and a Kalman object. As FORTRAN doesn't know dynamic memory allocation, removable object relations are extremely difficult to program in FORTRAN. As a consequence it is not possible to remove objects from ARTE-tables as then relations would become invalid. Vt++ is an example which shows the implementation of a removable object relation in C++ (see example 6.14). Vt++ is an application with which this idea can be further developed and which can then be applied to other software (e.g. CLUE).

RelationList This is a container class for pointers to Relation objects. As the Track and the Vertex classes are inherited from RelationList, the many-to-many relation between them is automatically established. This class implements the STL compatible iterators which are needed to loop over the relations in a convienient and standardized way.

MassC the mass constraint class. A MassC object describes a single mass constraint. This class keeps track of the tracks which participate in a mass constraint. A Vertex object can have many mass constraints. For an example, see section 6.11.

4.2 Namespace

Vt++ is coded in the **VERTEX** namespace. Therefore one can abbreviate the class names by declaring the namespace like this:

```
using namespace VERTEX;
```

Otherwise one has to use the full classname, which means to specify the **VERTEX::** prefix.

4.3 Functions and Templates

Some global functions and templates are provided, too. They belong to the **VERTEX** namespace. Functions which are not specific for Vt++ can be found in the **VtUtil.hh** header file. The following functions are implemented:

sqr template to compute the square of a number: $\mathrm{sqr}(x) = x \times x$

sgn template to compute the sign of a number: $\mathrm{sgn}(x) = \begin{cases} 0 & x = 0 \\ -1 & x < 0 \\ 1 & x > 0 \end{cases}$

getIDX template to compute the index of an ArtePointer:

```
ArtePointer<RSEG> rseg(5);
unsigned int idx = getIDX<RSEG>(rseg);
```

getPTR template to compute the ArtePointer from an index:

```
ArtePointer<RSEG> = getPTR<RSEG>(idx);
```

The **VtDistance.hh** headerfile declares distance functions (part of namespace **VERTEX**). The most important are:

distanceChi2(const Track& t1, const Track& t2) χ^2 distance between two Track objects.

distance(const Track& t, const Vertex& v) spatial distance between a Track and a Vertex object.

distanceChi2(const Vertex& v, double x, double y, double z) χ^2 distance beween a Vertex object and a space point.

distanceChi2(const Vertex& v1, const Vertex& v2) χ^2 distance between two vertices.

Vertex::distance(const Track& t) χ^2 distance between track and vertex (only available as member function of the Vertex class).

Some more useful distance functions have been added. Please see the reference manual for a complete list of distance functions.

5 Matrix computation

A Kalman filter mainly consists of matrix operations. To exploit the abilities of C++ it is natural to base the Kalman filter computation on a Matrix class. Several packages with matrix computation abilities exist: CERNLIB, ROOT, LAPACK, CLHEP. The disadvantage of those packages is that they introduce a lot of overhead as they come with a lot of (unwanted) stuff, are difficult to compile (if the needed binary type is not available) and difficult to understand. In Vt++, the goal was to have a matrix computation package which contains only the most needed operations, is easy to oversee and simple to adopt to own needs. The matrix classes are designed to call CERNLIB functions. In case one wants to switch to other products (like eg. LAPACK), this is easy to achive due to the modularity of the code.

5.1 Classes

Vt-Matrix consists of the following classes:

VtVector simple vector class. It can be constructed from a dimension n, another VtVector object or from a couple of double precision numbers (between 2 and 6 constructor arguments). Operators `=`, `+=`, `-=`, `*=`, `/=`, `+`, `-`, `*` (scalar product) are defined.

VtMatrix simple $m \times n$ Matrix class which can be constructed from a VtMatrix object or by specifying the dimensions m, n. The matrix elements can be accessed via the bracket operators:

```
VtMatrix a(2,3);  // 2x3 matrix
a(0,0)   = -2.5;
a[0][1] = 5.2;
cout << a;
```

The first access method is more efficient than the second. The following operations are defined: `=`, `+=`, `-=`, `*=`, `/=`, `+`, `-` (both unary and binary), `*`. Transponation is provided via a member function:

```
const VtMatrix b = a.T(); // a is not modified
a.VtT(); // transform a into it's transpose
```

VtSqMatrix representation of square $n \times n$ matrices. This class is inherited from VtMatrix. Additional functionality is the computation of the inverse (via CERNLIB dinv() function) and the determinant (via CERNLIB dfact() function). Two different member functions for matrix inversion exist, one for returning the inverse without modifying the matrix object, another one for transforming the matrix object into it's inverse.

VtSymMatrix representation of symmetric matrices. This class is inherited from VtSqMatrix. Operators `=`, `+`, `-`, `*`, `+=`, `-=`, `*=`, `/=` are optimized by exploiting the symmetry. The member function `product(const VtVector& v)` allows to compute the product $v^t * A * v$ using an optimized expression. Matrix inversion is done via CERNLIB function dsinv().

CMatrix covariance matrix class specific to HERA-B. This class is derived from VtSymMatrix and can be constructed from RTRA, RSEG, MTRA or another VtSymMatrix, VtSqMatrix, VtMatrix or CMatrix object. The CMatrix dimension is 5. **Note:** the CMatrix class makes the transformation $\sigma(Q/p) \rightarrow \sigma(p)$ (Q is the charge) because $\sigma(p)$ is needed by Vt. The `propagate(double z)` member function allows to propagate the covariance matrix to a given z-position. Each element of the covariance matrix can be accessed by a dedicated member function:

```
ArtePointer<RTRA> rtra(5);
CMatrix cov(rtra);
cout << " rtra->cvf[0]: " << cov.x()
     << " rtra->cvf[1]: " << cov.x_y()
     << " rtra->cvf[2]: " << cov.x_tx()
     << endl;
```

Matrix elements can be modified in the following way:

```
cov.set_x_y(0.00034);
cov.set_x_tx(1e-4);
```

To get a propagated matrix element:

```
double dz = -5.;
double new_x = cov.x_prop(dz);
double new_x_tx = cov.x_tx_prop(dz);
```

For a full list of access methods see the reference manual.

VtNegMatrix class for negative matrices. This class is inherited from VtMatrix. It optimizes computations like $C = -A*B$ where the negative of a matrix is involved. A FORTRAN implementation would look like this:

```
do  i = 1, 3
  do  j = 1, 3
    C(i,j) = 0.
    do  m = 1, 3
      C(i,j) = C(i,j) - A(i,m)*B(m,j)
```

```
      enddo
    enddo
  enddo
```

Using the VtMatrix class, the computation would look like this:

```
for(unsigned int i=0; i<3; ++i)
  for(unsigned int j=0; j<3; ++j)
      A(i,j) *= -1;

for(unsigned int i=0; i<3; ++i)
  for(unsigned int j=0; j<3; ++j) {
    C(i,j) = 0;
    for(unsigned int m=0; m<3; ++m)
      C(i,j) += A(i,m)*B(m,j);
  }
```

With the VtNegMatrix class, computation can be as efficient as in the FORTRAN example and as compact as the notation $C = -A*B$.

5.2 Namespace

Vt-Matrix is coded in the **MATRIX** namespace. Therefore one can abbreviate the class names by declaring the namespace like this:

```
using namespace MATRIX;
```

Otherwise one has to use the full classname, which means to specify the **MATRIX::** prefix.

6 VERTEX Examples

Some examples on how to use Vt++ are presented in the following subsections. A full list of available member functions can be found in the Vt++ reference manual. In this section the symbols `r1`, `r2`, ... represent RTRA, RSEG or MTRA ArtePointers.

6.1 Using the header file

In order to make use of Vt++, one has to include the appropriate headerfile:

```
#include "vt++/Vt++.hh"
```

6.2 Version information

Information about the Vt++ package can be obtained via the `VtIni` class:

```
VERTEX::VtIni pkg;
cout << pkg << endl;
```

A string is printed which contains the name of the package, the version number and the creation time of this version. Example:

```
vt++ version /1-2 loaded (Mit Okt 25 18:29:27 MEST 2000).
```

It is also possible to get the information separatly:

```
VERTEX::VtIni pkg;
cout << "Package name: "  << pkg.name()
     << "Version: "       << pkg.version()
     << "Creation time: " << pkg.date()
     << endl;
```

6.3 Example of two track vertices

A complete example on how to use the Vertex class to fit two-track vertices is shown here:

```
#include "vt++/Vt++.hh"
...

for(ArteTable<RTRA>::const_iterator
      it1  = ArteTable<RTRA>::begin();
      it1 != ArteTable<RTRA>::end();
    ++it1) {
    ArtePointer<RTRA> r1 = *it1;

    if(r1->hsi < 6) continue;  // simple track selection

    ArteTable<RTRA>::const_iterator tmp = it1;
    for(ArteTable<RTRA>::const_iterator it2 = ++tmp;
        it2 != ArteTable<RTRA>::end(); ++it2) {
        ArtePointer<RTRA> r2 = *it2;

       if(r2->hsi < 6) continue;  // simple track selection

        VERTEX::Vertex v(r1,r2);
        if(v.findVertexVt() == true && v.chi2() < 4.) {
          cout << "vertex found: " << v << endl;
        }
    }
}
```

6.4 Vertex fit methods

Five different methods to compute a vertex are available:

```
VERTEX::Vertex v(r1,r2);
v.findVertexVt();      // Vt Kalman filter method
v.VtMass();            // Vt mass constrained Kalman filter
v.findVertex2D();      // unweighted 2-D min. distance
v.findVertex3D();      // unweighted 3-D min. distance
v.VtEstimateVertex(); // weighted 2-D min. distance
```

Note: the functions return `true` in case of success, else `false`. For mass constrained fits, see also section 6.11.

6.5 Using momentum in the vertex fit

Per default, the Kalman filter uses the track momentum for the vertex fit. To make a vertex fit without momentum (straight line fit) do the following:

```
VERTEX::Vertex v(r1,r2);
v.use_momentum(false);
v.findVertexVt();
```

To turn on momentum usage again, type

```
v.use_momentum(true);
v.findVertexVt(); // vertex fit with momentum
```

You can also set the momentum usage for tracks individually:

```
ArtePointer<RTRA> r1 = ...;
ArtePointer<RTRA> r2 = ...;
VERTEX::Vertex v(r1,r2);

// turn off momentum usage for tracks without OTR segment!
for(RelationList::iterator it = v.begin(); it != v.end();
    ++it) {
   // get back ArtePointer from Track object
   ArtePointer<RTRA> rtra = it->track.rtra();

   if(rtra->hou == 0)
     it->kalman.use_momentum(false);
}

if(v.findVertexVt() == true) {
  cout << "Vertex found: " << v << endl;
}
```

For more information about the Vertex-Iterator see section 6.10.

Note: The other three vertexing routines do not use momentum information.

6.6 Adding more Tracks

Because of performance reasons, a vertex is not automatically refitted if a track is added. The user can decide when the vertex fit should be done:

```
VERTEX::Vertex v(r1,r2);
v.push_back(r3);
v.push_back(r4);
v.findVertexVt();
if(v.valid()) cout << v << endl;
```

To use an efficient way to decide wether a track is compatible with the vertex or not, one can calculate the χ^2-distance between the track and the vertex:

```
VERTEX::Vertex v(r1,r2);
if(v.findVertexVt() == true) {
  Track t(r3);

  // add track if chi2-dist is < 4. (ndf = 2)
  if( v.distance(t) < 4.) {
    v.push_back(t);
    v.findVertexVt();
  }
}
```

6.7 Vertex members functions

Vertex information can be accessed like this:

```
VERTEX::Vertex v(r1,r2);
if(v.findVertexVt() == false) return;

cout << " x: " << v.vx()
     << " y: " << v.vy()
     << " z: " << v.vz()
     << " error on x: " << v.vxerr()  // sqrt(rver->cv[0])
     << " error on y: " << v.vyerr()  // sqrt(rver->cv[2])
     << " error on z: " << v.vzerr()  // sqrt(rver->cv[5])
     << " chi2: " << v.chi2() // only available for Kalman-fit
     << " no of tracks: " << v.ntracks()
     << endl;
```

See the reference manual for a full list of access functions.

6.8 Mean distance and opening angle

To compute the mean distance of the tracks to the vertex and the rms opening angle, a special member function exists:

```
VERTEX::Vertex v(r1,r2);
v.push_back(r3);
if(v.findVertexVt()) {
  v.rmsDistAngle();
  cout << " rms distance: " << v.dist()
       << " rms angle: "    << v.angle()
       << endl;
}
```

6.9 Reconstructed Mass

To compute the reconstructed mass, one can choose to use the refitted momenta (in case the vertex fit was done using the Kalman filter) or to use the original track momenta. This behaviour can be toggled:

```
VERTEX::Vertex v(r1,r2);
if(v.findVertexVt()==false) return;
cout << "mass using orig. track momenta, particle rest mass=0: "
     << v.mass() << endl;

v.use_kalman(true); // turn on usage of refitted track params.

cout << "mass using refitted momenta, particle rest mass=0: "
     << v.mass() << endl;

v.use_kalman(false); // turn off refitted track params.
```

There are several methods to assign a rest mass to the tracks on the vertex. To assign a single rest mass to all tracks, do it like this:

```
VERTEX::Vertex v(r1,r2);
if(v.findVertexVt() == true) {
  cout << " pi-pi mass: " << v.mass(0.1396)
       << " K-K mass: "   << v.mass(0.494)
       << endl;
}
```

For vertices with up to 4 tracks, rest masses can be assigned to the individual tracks:

```
VERTEX::Vertex v(r1,r2);
if(v.findVertexVt()) {
  cout << " pi-K mass: " << v.mass(0.1396,0.494)
       << " K-pi mass: " << v.mass(0.494,0.1396)
       << endl;
}
v.push_back(r3);
if(v.findVertexVt()) { // fit 3-track vertex
  cout << "pi-K-pi mass: " << v.mass(0.1396,0.494,0.1396)
       << endl;
}
// wrong: 3-track vertex but only 2 arguments:
cout << " wrong mass assignment: " << v.mass(0.1396,0.494)
     << endl;
```

In case of a wrong mass assigment, the result is -1. The mass assignment can also be done in case of n tracks:

```
VERTEX::Vertex v(r1,r2);
v.push_back(r3);
v.push_back(r4);
v.push_back(r5);
v.push_back(r6);
v.push_back(r7);

// rest mass vector
vector<double> m0(7);
m0[0] = 0.1396*0.1396; // note: square of rest mass
m0[1] = ...
```

```
...
m0[6] = 0.494*0.494;

cout << "mass: " << v.mass(m0);
```

You can also assign a rest mass permanently to a track:

```
ArtePointer<RTRA> r = ...;
ArtePointer<TOPP> pi(101);

Track t(r);
Track u(r);

t.rm(0.140);      // assign rest mass by hand
u.rm(pi->mass);   // or via TOPP-table:

cout << "Track rest mass: " << u.rm() << endl;
```

In case you do not specify a rest mass, the permanent rest mass will be taken:

```
ArtePointer<TOPP> pi(101);

Track t1(r1);
Track t2(r2);
t1.rm(pi->mass);    // set pion rest mass
t2.rm(pi->mass);    // for both tracks
VERTEX::Vertex v(t1,t2);

if(v.findVertexVt() == true) {
  cout << "pi-pi mass: " << v.mass() << endl;
}
```

6.10 Using the Vertex-Iterator

To loop over the tracks assigned to a vertex and to access the Kalman filter information, the Relation iterator can be used in the following way:

```
using namespace VERTEX;
...

Vertex v(r1,r2);
if( v.findVertexVt() == false ) return;

for(const_iterator it = v.begin(); it != v.end(); ++it) {
  const Track& track = it->track;
  cout << "original track parameters:"
       << " x: " << track.x()
       << " y: " << track.y()
       << " p: " << track.p()
       << endl;
}
```

It is also possible to iterate over the tracks in reverse order:

```
using namespace VERTEX;
...

Vertex v(r1,r2);
if( v.findVertexVt() == false ) return;

for(const_reverse_iterator it = v.rbegin(); it != v.rend();
    ++it) {
  const Track& track = it->track;
  cout << "original track parameters:"
       << " x: " << track.x()
       << " y: " << track.y()
       << " p: " << track.p()
       << endl;
}
```

If you want to have a direct access to the track or kalman information without using the iterator, the following can be done:

```
using namespace VERTEX;
...

Vertex v(r1,r2);
if(v.findVertexVt()) {
  const Track& track1 = v[0]->track;
  const Track& track2 = v[1]->track;
  cout << " Track 1: " << track1
       << " Track 2: " << track2
       << endl;
  cout << " p1: "      << track1.p()
       << " p2: "      << track2.p()
       << endl;
}
```

This method also works with tracks:

```
Track t1(r1);  // generate Track objects from
Track t2(r2);  // ArtePointers

Vertex v(t1,t2);
if(v.findVertexVt()) {
  const Vertex& vertex1 = t1[0]->vertex;
  const Vertex& vertex2 = t2[0]->vertex;
  // both vertex objects are the same in
  // this simple example...
}
```

But be careful: **direct access is only emulated and therefore less efficient!** The Relation iterators are STL[2] compatible, that means standard STL algorithms can be used:

[2]Standard Template Library, see e.g. `http://www.sgi.com/Technology/STL`.

```
Vertex v(r1,r2);
v.push_back(r3);
cout << "No of tracks in vertex: "
     << distance(v.begin(),v.end())
     << endl;
```

6.11 Mass constrained vertex fits

A mass constrained vertex fit can be used in case several tracks are known to originate from a narrow mass state. By this the precision of the parameters obtained from the vertex fit can be improved. In general several mass constraints with different combinations of tracks can be assigned to a vertex. In the most simple case, all tracks from a vertex participate in a single mass constraint:

```
VERTEX::Vertex v(r1,r2);

// TOPP entry of mother particle (K0)
ArtePointer<TOPP> K0(160);

// all tracks in vertex originate from a
// K0 mass state
v.MassConstr(K0->mass);

// mass constrained vertex fit
if( v.VtMass() == true ) {
  cout << "K0-vertex: " << v << endl;
}
```

Note: performing a mass constrained vertex fit always implies that the track momenta are used. Therefore `VtMass()` calls `use_momentum(true)` first. This will overwrite all `use_momentum` settings in the track objects.

The rest masses of the outgoing particles enter into the mass constraint, so in case you know in the above example that `r1` and `r2` are pions, you can set the pion rest mass before you perform the vertex fit:

```
VERTEX::Vertex v(r1,r2);

ArtePointer<TOPP> pi(101);
ArtePointer<TOPP> K0(160);

for(iterator it = v.begin(); it != v.end(); ++it) {
  it->track.rm(pi->mass);
}

v.MassConstr(K0->mass);

if( v.VtMass() == true ) {
  cout << "Vertex: " << v << endl;
}
```

In case you want to apply a mass constraint to a subset of tracks, you have to select for each mass constraint the tracks which belongs to it. The user has to manipulate a MassC object, which is generated by the Vertex object:

```
VERTEX::Vertex v(r1,r2);
v.push_back(r3);

ArtePointer<TOPP> Jpsi(295);
ArtePointer<TOPP> B0(246);

// get a mass constraint object
// attention: always use a reference!
MassC mC& = v.addMassConstr();

// set mass of mother particle
mC.mass(Jpsi->mass);

// loop over tracks on vertex
for(const_iterator it = v.begin(); it != v.end(); ++it) {
  const Track& t = it->track;

  // store relations to tracks which belongs
  // to the mass constraint in MassC object
  if( ... )
    mC.push_back(*it);
}

// apply a second mass constraint for all tracks
v.MassConstr(B0->mass);

if( v.VtMass() == true ) {
  cout << "B0 vertex: " << v << endl;
}
```

To remove all mass constraints from a vertex, use the `clearMassConstr()` member function (removing a single mass constraint object is not supported):

```
VERTEX::Vertex v(r1,r2);

v.MassConstr(0.496);
v.VtMass();
v.clearMassConstr(); // remove mass constraints
cout << "no of mass constraints: "
     << v.nMassConstr()
     << endl;
```

Note: In case a track is deleted from a vertex with mass constraints, make sure that the track was not part of a mass constraint. Otherwise you will have an invalid mass constraint object. If you want to delete a track which is part of a mass constraint, delete the mass constraints, too:

```
VERTEX::Vertex v(r1,r2);
v.push_back(r3);

ArtePointer<TOPP> B0(246);
v.MassConstr(B0->mass);
v.VtMass();

...

// remove track with biggest chi2 contribution
v.remove_worst();
// remove mass constraints if defined
if(v.nMassConstr() > 0)
  v.clearMassConstr();
```

6.12 Refitted Track parameters

After a successful Kalman filter based vertex fit, the track parameters are refitted with the constraint that the tracks originate in the vertex. You can get the refitted parameters like in the following example:

```
using namespace VERTEX;
...

Vertex v(r1,r2);
if( v.findVertexVt() == true ) {

  cout << "Refitted track parameters: "
       << " x: " << v.vx()  // same
       << " y: " << v.vy()  // for all
       << " z: " << v.vz()  // tracks
       << endl;
  for(iterator it = v.begin(); it != v.end(); ++it) {
    const Kalman& kalman = it->kalman;
    cout << " tx: " << kalman.tx()
         << " ty: " << kalman.ty()
         << " p: "  << kalman.p()
         << endl;
  }

  // or:
  cout << "refitted parameters of 1st track:"
       << " x: "  << v.vx()
       << " y: "  << v.vy()
       << " tx: " << v[0]->kalman.tx()
       << " ty: " << v[0]->kalman.ty()
       << endl;
}
```

6.13 Store Vertex in RVER table

A Vertex object can be stored as an RVER object in the following way:

```
VERTEX::Vertex v(r1,r2);

if(v.findVertexVt() && v.chi2() < 4.) {
  cout << "Created new RVER entry: "
       << v.createRVER()
       << endl;
}
```

The `createRVER()` member function returns an ArtePointer to the RVER object. If the Vertex object is not generated by RTRA tracks, the RREL relations are not set and a warning is printed to stdout. **Note:** in case of RSEGs, no transformation into the global HERA-B coordinate system is done.

6.14 Using an external Track list

For more complicated operations it is better to have a global track list to work with. The track list can be e.g. generated by a track selection routine:

```
using namespace VERTEX;
...
vector<Track> trvec;   // Track container

for(ArteTable<RTRA>::iterator it=ArteTable<RTRA>::begin();
    it!=ArteTable<RTRA>::end(); ++it) {
    ArtePointer<RTRA> rtra = *it;

    if(rtra->hsi > 6 && rtra->hou > 20) // track selection
       trvec.push_back(Track(rtra));    // store track
}
```

Now one can combine the selected tracks to fit e.g. two-track vertices:

```
vector<Vertex*> vtx;   // vertex container

for(unsigned int i=0; i<trvec.size(); ++i) {
  for(unsigned int j=i+1; j<trvec.size(); ++j) {
    Vertex* v = new Vertex(trvec[i],trvec[j]);

    if(v->findVertexVt())
      vtx.push_back(v);
    else
      delete v;
  }
}
```

The construction of a vertex generates track-vertex relations. For sake of execution speed, one has to generate the Vertex-objects on the heap and store the Vertex-pointer in order to avoid useless copying. Despite of this fact it is always a bad idea to store Vertex objects in STL containers. For an explanation see [4]. If the Vertex object is constructed on the stack memory, the relations are lost[3]:

```
vector<Vertex> vtx;   // store found vertices

for(unsigned int i=0; i<trvec.size(); ++i) {
  for(unsigned int j=i+1; j<trvec.size(); ++j) {
    Vertex v(trvec[i],trvec[j]);

    if(v.findVertexVt())
      vtx.push_back(v); // inefficient copying!
  }
}
// !! The Track-Vertex relations are lost now!!
```

Once the vertices are generated, one can exploit the relations. Looping over all vertices a track belongs to and plot the χ^2 is done in the following way:

```
// loop over tracks from global track list
for(unsigned int i=0; i<trvec.size(); ++i) {
  const Track& track = trvec[i];

  // loop over vertices a track belongs to
  for(const_iterator it = track.begin();
      it != track.end(); ++it)
      cout << " vertex-chi2 "
           << it->vertex.chi2()
           << " refitted track momentum: "
           << it->kalman.p()
           << endl;
}
```

Looping over all tracks of a vertex is done like this:

```
// loop over all vertices
for(unsigned int i=0; i<vtx.size(); ++i) {
  const Vertex& vertex = *vtx[i];

  // loop over tracks on a vertex
  for(const_iterator it = vertex.begin();
      it != vertex.end(); ++it)
    cout << " track-momentum: "
         << it->track.p()
         << " refitted track-momentum: "
         << it->kalman.p()
         << endl;
}
```

[3] Because they are deleted by the destructor of the temporary vertex object which owns the relations.

If you delete a Track from the global Track list, the track gets automatically removed from all vertices it belongs to. Those vertices which lost the track are flagged invalid and can be refittet:

```
trvec.pop_back(); // delete the last track
                  // from the global track list

// do a garbage collection
for(unsigned int i=0; i<vtx.size(); ++i) {
  Vertex* vertex = vtx[i];

  if(vertex->size() < 2) {
      delete vertex;
      vtx[i] = 0; // set NULL pointer in vector
      continue;
  }

  // refit vertices which lost a track
  if(vertex->valid() == false)
     vertex->findVertexVt();
}
```

6.15 Reconstruct the incoming track

Once a vertex has been found, the track parameters of the incoming track can be reconstructed using the vertex space-point and the sum of the momenta of the outgoing tracks. The user can choose when the incoming track should be reconstructed by calling the `calc_mother()` member function:

```
using namespace VERTEX;
...
Vertex v(r1,r2);
if(v.findVertexVt() == true) { // vertex found
  v.calc_mother();
  cout << v;
  // or using member functions:
  cout << "Reconstructed Track:"
       << " x: "  << v.x()
       << " y: "  << v.y()
       << " z: "  << v.z()
       << " tx: " << v.tx()
       << " ty: " << v.ty()
       << " p: "  << v.p()
}
```

The covariance matrix of the incoming track can be computed seperately. Once the covariance matrix is available, the vertex object can be used again for a Kalman-filter vertex fit:

```
using namespace VERTEX;
...
```

```
Vertex v(r1,r2);
if(v.findVertexVt() == true) {
  // track parameters of incoming track
  v.calc_mother();
  // covariance matrix of incoming track
  v.calc_mother_cov();
  cout << "Cov. matrix of incoming track: "
       << v.V();

  // vertex incoming track with an RTRA track r3:
  Track t(r3);
  Vertex vv(t,v);
  if(vv.findVertexVt() == true) {
    cout << " Vertex found: " << vv << endl;
  }
}
```

6.16 Removing tracks from a Vertex

There are several ways to remove a track from a vertex. Special member functions are provided in order to remove the last track from the track list and the track with the worst χ^2 contribution:

```
using namespace VERTEX;
...
Vertex v(r1,r2);
v.push_back(r3);
v.findVertexVt();
// remove r3 again using the inverse Kalman filter
if(v.remove_last() == true)
  cout << "last track removed" << endl;

v.push_back(r3);
v.push_back(r4);
v.findVertexVt();
// remove track with biggest chi2 contribution using
// the inverse Kalman filter:
if(v.remove_worst() == true)
  cout << "track with largest chi2 removed" << endl;
```

As `remove_worst()`, `remove_last()` and `VtRemoveTrack()` use the inverse Kalman filter, make sure that the vertex is fitted by `findVertexVt()` before you call these functions. You cannot use these functions in case of a 2-track vertex.

The most flexible way to remove one or several tracks from a vertex is to use the `erase()` member function. This allows to use the STL way of removing an element from the relation container. The following example shows how to remove all tracks with a χ^2-contribution larger than 1 from the vertex:

```
using namespace VERTEX;
```

```
...
// take a primary vertex from RVER
ArtePointer<RVER> prim_vtx = ...;
Vertex v(prim_vtx);
// get full kalman information for this vertex
v.findVertexVt();

double prchi2 = 0;
// STL way to use erase()
for(iterator it=v.begin(); it!=v.end();) {
  const Kalman& track = it->kalman;
  double chi2 = track->chi2();

  if( (chi2 - prchi2) > 1.) {
    it = v.erase(it);
  } else {
    ++it;
  }
  prchi2 = chi2;
}

// refit the vertex in case tracks were removed
if(v.valid() == false)
  v.findVertexVt();
```

The `erase()` member function removes the relations without performing the inverse Kalman filter. It is under user control when the vertex should be again refitted.

6.17 Distance functions

A couple of distance functions do exist. They are all defined in the VERTEX namespace although not all distance functions deal with VERTEX classes:

```
using namespace VERTEX;
...
ArtePointer<RVER> rver(4);
ArtePointer<MVER> mver(3);
ArtePointer<GTAR> gtar(2);

// no clash with STL possible
cout << "spatial rver-mver distance: " << distance(rver,mver)
     << endl
     << "spatial rver-gtar distance: " << distance(rver,gtar)
     << endl;
```

Some distance functions are implemented as class members and can be called in two ways:

```
using namespace VERTEX;
...
```

```
Vertex v1(r1,r2);
Vertex v2(r3,r4);

cout << "chi2 vertex-vertex distance: "
     << v1.distance(v2)
     << " or: "
     << distanceChi2(v1,v2)
     << endl;

cout << "chi2 distance vertex-space point: "
     << v1.distance(0.1,0.1,-4.9)
     << " or: "
     << distanceChi2(v1, 0.1, 0.1, -4.9)
     << endl;
```

More special distance functions are named differently. E.g. the function `SGNdistance(const Track& t, const Vertex& v)` returns the signed spatial distance between Track and Vertex. The sign tells you if the point of the track closest to the vertex is upstream ($sgn = -1$) or downstream ($sgn = +1$) of the vertex. This information can be used for background suppression.

6.18 Pull - distributions

The pull-distributions of the reconstructed vertices in x, y and z are shown in figure 1. A pull of a variable x_i is defined like this:

$$P(x_i) = \frac{x_i^{\mathrm{REC}} - x_i^{\mathrm{MC}}}{\sqrt{C_{ii}}} \tag{1}$$

MC reconstruction was done with Geometry 99.0813 and ARTE-version 03-07-r1. VDS track segments reconstructed by CATS were used for the vertexing. One can see that apart from non-gaussian tails the width of the pull-distributions is about 1 as one would expect in case of a correct covariance matrix. Using RTRA tracks for vertexing broadens the sigma of the pull-distributions by about 30%, which means that the errors in the RTRA covariance matrix are underestimated.

7 MATRIX Examples

The vector and matrix objects which can be constructed use only `double` precision numbers.

7.1 VtVector Objects

A default VtVector object can be constructed by giving the dimension as an argument. The components will be initialized to 0.:

```
MATRIX::VtVector v(10);  // 10-dim vector
cout << v << endl;
```

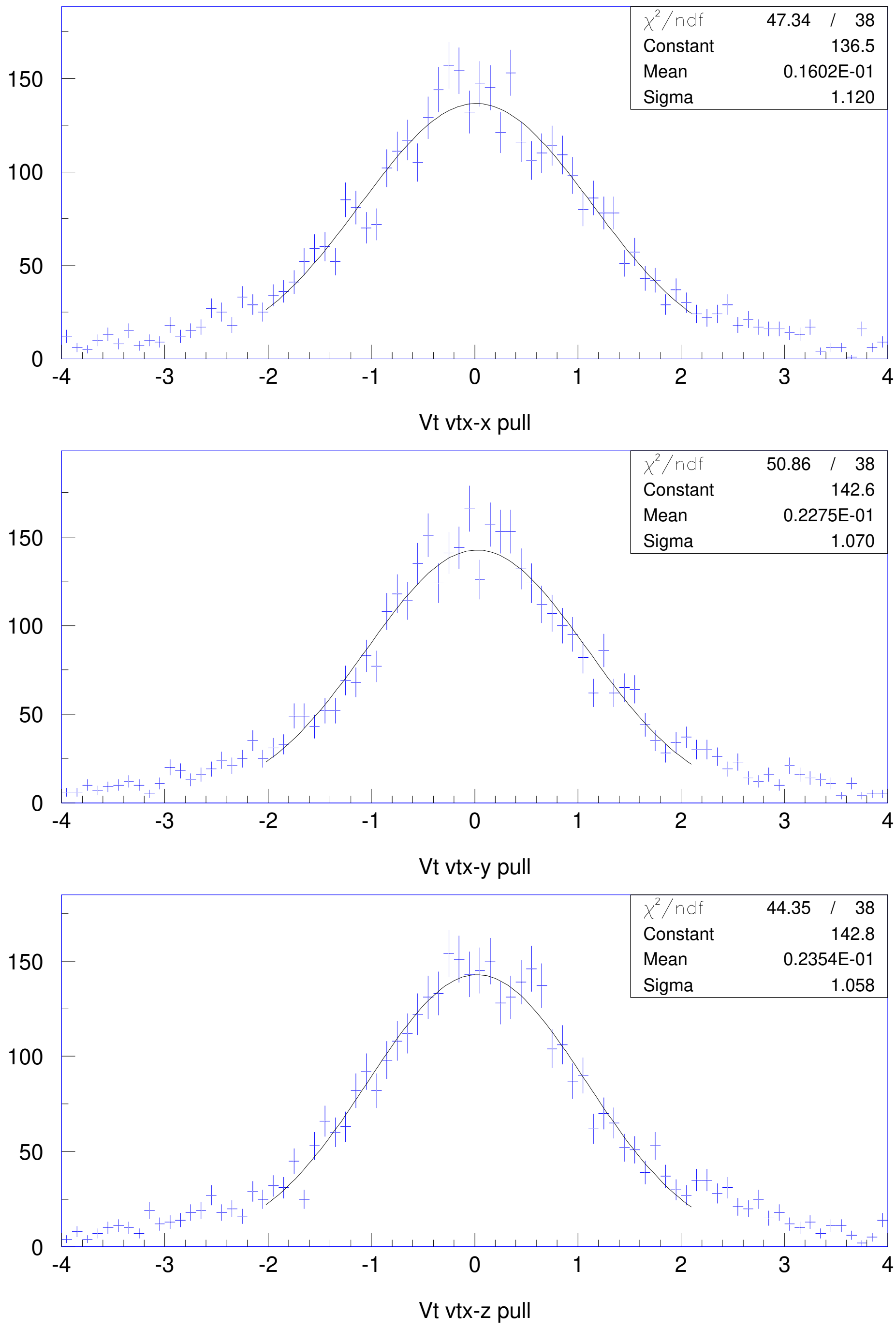

Figure 1: Vt++ vertex pull-distributions for x, y and z coordinate

Another alternative is to define the vector by specifying the components already in the constructor call:

```
MATRIX::VtVector v(1.0,2.0);  // 2-dim vector
MATRIX::VtVector w(-2.4,3.14,2.714,4.55,17.77);
```

Up to 6 arguments are supported.

To access the vector components both `operator()()` and `operator[]()` can be used:

```
MATRIX::VtVector v(4);

v(0) = 5.1;  // or: v[0] = 5.1
v(1) = -4.9; // or: v[1] = -4.9

cout << "1st component: " << v[0]
     << "2nd component: " << v[1]
     << endl;
```

Both access methods are equivalent.

Mathematics with VtVector objects is intuitive:

```
MATRIX::VtVector v(1.0,2.0);
MATRIX::VtVector w(3.0,4.0);

// create vector via copy-constructor
MATRIX::VtVector u = v + w;

// use a temporary vector object
cout << "v - w: " << v - w << endl;

v += 3.;  // add 3 to each component
double pi = 3.14159265;
w *= pi;

// skalar product is supported:
double vw = v * w;
```

Note: The dimensions of the VtVector objects must be the same. If they are not the same, the program will exit with an `assert`-statement in case the code is compiled without the `-DVtFAST` option. In case the `-DVtFAST` compile option was used, the unequal vector dimensions are ignored, which might lead to unexpected results.

The following special functions exist:

```
MATRIX::VtVector v(-2.5,4.77);

cout << "dimension of v: " << v.size() // or v.nrow()
     << "vector length : " << v.mag()
     << "squared length: " << v.mag2()
     << endl;
```

7.2 Constructing Matrix objects

Four different kinds of matrices are supported: $n \times m$ matrices, $n \times n$ matrices, symmetric $n \times n$ matrices and symmetric, positive definite 5×5 HERA-B Track covariance matrices. For each of the above matrix types a class exists: `VtMatrix`, `VtSqMatrix`, `VtSymMatrix`, `CMatrix`. The constructors are different:

```
using namespace MATRIX;
...
ArtePointer<RTRA> r(4);

VtMatrix    A(2,3);  // construct a 2x3 matrix
VtSqMatrix  B(2);    // construct a 2x2 matrix
VtSymMatrix C(2);    // construct a 2x2 sym. matrix
CMatrix     D(r);    // cov. matrix of 4th RTRA
```

For a full list of constructors see the reference manual.

To print a matrix object, just use `cout`:

```
MATRIX::VtSqMatrix A(2);
A[0][0] = 0.5;

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

7.3 Accessing Matrix elements

Matrix elements can be accessed in two ways: either using the `operator()()` or the `operator[]()` member functions:

```
using namespace MATRIX;
...
VtMatrix A(2,3);

A(0,0)  = 1.;
A(0,1)  = 2.;
A(1,0)  = 5.;
...
// or:
A[0][0] = 1.;
A[0][1] = 2.;
A[1][0] = 5.;
...
```

The `operator()()` access method is slightly more efficient, on the other hand the [][]-method is well known from C and easier to recognize.

7.4 Choosing the right Matrix class

The special properties of the classes are exploited wherever possible, therefore it is a matter of execution speed to choose the right matrix class.

Example: One can use the `VtMatrix`, `VtSqMatrix` and `VtSymMatrix` class in order to represent a symmetric matrix, but only the latter class is able to exploit the symmetry and produces faster code. For the user the code looks in any case the same:

```
using namespace MATRIX:
...

// inefficient symmetric matrix:
VtMatrix A(2,2);
A[0][0]           = 1.;
A[0][1] = A[1][0] = 2.;
A[1][1]           = 3.;

VtMatrix    B = A;
B *= 5.4;
VtMatrix    C = B + A;

// efficient symmetric matrix:
VtSymMatrix a(2);
a[0][0]           = 1.;
a[0][1] = a[1][0] = 2.;
a[1][1]           = 3.;

VtSymMatrix b = a;
b *= 2.7;
VtSymMatrix c = b + a;
```

7.5 Matrix operations

Using the matrix classes for mathematical operations is more or less self explaining:

```
VtMatrix A(2,3);

// fill matrix A
...

A *= 4; A += 3.14;

VtMatrix B = A * 5.;
VtMatrix C = -B + A;

C = A.T();  // C is transpose of A
C.VtT();    // transform C into it's transpose

VtVector a(-1.0, 2.4, 3.5);
VtVector b = A * a;
cout << "scalar product: " << a * b << endl;

VtSymMatrix D(3);  // 3x3 matrix

// fill matrix D
```

```
...

VtSymMatrix E = D.dsinv(); // return inverse
D.VtDsinv(); // transform D into its inverse
```

For a full list of supported operations see the reference manual.

Special member functions exist to calculate the product $v^t * A * v$ and $B^t * A * B$ (A, B are matrices, v is a vector):

```
using namespace MATRIX;
...
VtVector v(1.0,2.0,3.0);
VtSymMatrix A(3);
VtMatrix    B(3,2);

// fill matrices A, B
...

// calculate v^t*A*v:
double vAv = A.product(v);

// calculate B^t*A*B:
VtSymMatrix C(2);
C = A.product(B);
```

7.6 Special member functions

With the `clear()` member function you can set the content of a matrix or vector object to zero:

```
using namespace MATRIX;
...
VtVector a(2.5,1.0,0.9);
a.clear();
VtMatrix A(3,4);
A = ...;
A.clear();
```

To copy one matrix into another, two possibilities exist. In case the dimensions of both matrices are equal, `operator=()` can be used:

```
MATRIX::VtSymMatrix A(3);
A = ...;
MATRIX::VtSymMatrix B(3);
B = A;  // operator=() is used
```

If the dimensions of the matrices are not equal, the `copy()` member function can be used:

```
MATRIX::VtMatrix A(2,4);
A = ...;
MATRIX::VtMatrix B(2,3);
B.copy(A);
```

In this case only that part of `A` which fits into `B` is copied. Example:

```
A:   1.0  2.0  3.0  4.0    =>    B:   1.0  2.0  3.0
     5.0  6.0  7.0  8.0               5.0  6.0  7.0
```

With the `place_at()` member function you can copy a smaller matrix at a certain position in a bigger matrix. This is useful in case you have to construct a big matrix out of several small matrices.

```
using namespace MATRIX;
...
VtMatrix A(2,2);
A = ...;
VtMatrix B(5,5);
B.place_at(A,2,1);
```

Example:

```
A:   1.  2.    => B:   0.  0.  0.  0.  0.
     3.  4.            0.  0.  0.  0.  0.
                       0.  1.  2.  0.  0.
                       0.  3.  4.  0.  0.
                       0.  0.  0.  0.  0.
```

8 Vt++ running in ROOT/CINT

It is possible to make use of Vt++ classes in ROOT[4]. An indirect way is to precompile user functions which contain calls to Vt++ classes. But it is also possible to use Vt++ classes directly in CINT. To do so, it is neccessary to generate a shared library containing the Vt++ classes. This can be done by modifying the `Makefile.BEE` makefile which is contained in the Vt++ package. In future a Vt++ shared library will be provided with the BEE analysis software[5]. For the following examples ROOT version 2.25.03 has been used.

Once the shared library (including the cint dictionary) is available, it can be loaded into ROOT:

```
*******************************************
*                                         *
*        W E L C O M E  to  R O O T       *
*                                         *
*   Version   2.25/03    27 October 2000  *
*                                         *
*  You are welcome to visit our Web site  *
*          http://root.cern.ch            *
*                                         *
*******************************************

CINT/ROOT C/C++ Interpreter version 5.14.50, Sep 2 2000
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
```

[4] see `http://root.cern.ch`

[5] see `http://www.mpi-hd.mpg.de/herab/clue`

```
root [0] gSystem->Load("libvt++.so");
vt++ version /1-2 loaded (Mit Okt 25 18:29:27 MEST 2000).
```

The version information will be printed automatically when the shared library is loaded.

Now you can e.g. get the version information interactively:

```
root [1] using namespace VERTEX
root [2] VtIni a
root [3] cout << a
vt++ version /1-2 loaded (Mit Okt 25 18:29:27 MEST 2000).
(class ostream)1076664544
```

In order to use the vertexing capabilities of Vt++, track information is needed. This will be provided within the BEE package. Outside BEE, the MATRIX classes can still be used:

```
root [4] using namespace MATRIX
root [5] VtVector v(1,2,3)
root [6] cout << v

            1
            2
            3
(class ostream)1076664544
```

The classes can be used in the expected way:

```
root [7] VtVector b(5,6,7)
root [8] cout << "scalar product: " << v * b << endl;
scalar product: 38
(class ostream)1076664544
```

A problem is the `operator[]()` in the matrix classes. `operator()()` can be used instead:

```
root [9] VtMatrix z(3,3)
root [10] z(0,1) = 1.
(const double)1.00000000000000000e+00
root [11] cout << z

            0             1             0
            0             0             0
            0             0             0
(class ostream)1076664544
```

References

[1] T. Lohse, Vertex Reconstruction And Fitting, HERA-B Note 95-013

[2] T. Glebe, Vt++ Version 1.0, HERA-B Note 00-175

[3] Todd L. Veldhuizen et al., Will C++ be faster than Fortran? Proceedings of the 1st International Scientific Computing in Object Oriented Parallel Environments (ISCOPE'97), 1997 Springer.

[4] D. Kühl, STL and OO Don't Easily Mix, Proceedings of the 2000 Workshop on C++ Template Programming.

[5] T. Veldhuizen, Expression Templates, C++ Report, Vol. 7 No. 5 June 1995.