

MtxVec v4

A programming guide to MtxVec for C++Builder

MTXVEC V4	2
A PROGRAMMING GUIDE TO MTXVEC FOR C++BUILDER	2
1 INTRODUCTION	4
1.1 WHY MTXVEC.....	4
1.2 HOW FAST IS MTXVEC.....	5
2 ORGANIZATION	5
2.1 COMPILER SUPPORT	5
3 QUICK START	6
4 MTXVEC PROGRAMMING INTERFACE	7
4.1 OBJECT HIERARCHY	7
4.2 SMART POINTERS	7
4.3 VECTOR, MATRIX, CVECTOR, CMATRIX, SPARSEMATRIX.....	8
4.4 METHOD CONVENTIONS OF TVEC AND TMTX	8
4.5 METHODS AND FUNCTIONS	8
4.6 REAL NUMBER DATA TYPE	9
4.7 COMPLEX NUMBER DATA TYPE	9
4.8 OPERATOR OVERLOADING	10
4.9 ELEMENT BY ELEMENT VECTOR/MATRIX EXPRESSION TYPES.....	11
4.10 LINEAR ALGEBRA VECTOR/MATRIX EXPRESSION TYPES.....	12
4.11 LINEAR ALGEBRA WITH TVEC AND TMTX TYPES	12
4.11.1 <i>Other types of operations</i>	12
4.12 INDEXES, RANGES AND SUBRANGES	13
4.13 SUBRANGES WITH OPERATOR OVERLOADING.....	14
4.14 RANGE CHECKING	15
4.15 OPEN ARRAY PARAMETERS AND SETIT METHOD.....	15
4.16 LOADING AND SAVING A MATRIX IN A TEXT FILE.	16
5 ADVANCED TOPICS	17
5.1 PREALLOCATED OBJECTS	17
5.2 WITHOUT REDUNDANT COPYING.....	17
5.3 MORE ABOUT SUBRANGES	18
5.4 MAKING USE OF THE TMTXVEC ABSTRACT CLASS	19
5.4.1 <i>Writing abstract class code</i>	20
5.5 FUNCTION PARAMETERS	20
5.6 POOL OF PREALLOCATED VECTORS AND MATRICES	21
6 MEMORY MANAGEMENT	21
6.1 INTRODUCTION	21
6.2 IN-PLACE/NOT-IN-PLACE OPERATIONS	22
7 WHY AND HOW NAN AND INF	23
7.1 KNOWN ISSUES	23
8 SERIALIZING AND STREAMING	24
8.1 STREAMING WITH TMTXCOMPONENT	24
8.2 STREAMING OF TVEC, TMTX AND TSPARSEMTX	24
8.3 READING AND WRITING RAW DATA.....	24
9 WORKING WITH TEXT	25
9.1 WRITE TMTX/MATRIX TO A TEXT FILE	25
9.2 READ TMTX/MATRIX FROM A TEXT FILE.....	25
9.3 FORMATTING FLOATING POINT VALUES – FLOATTOSTRING	25
10 PROGRAMMING STYLE	27
10.1 MIXING TVEC/TMTX AND MATRIX/VECTOR TYPES.	27

10.2	INLINING OF FUNCTIONS THAT USE VECTOR AND MATRIX TYPES	27
10.3	RAISING EXCEPTIONS	27
10.4	INVALID PARAMETER PASSED:	27
10.5	USE CREATEIT/FREEIT ONLY FOR DYNAMICALLY ALLOCATED OBJECTS WHOSE LIFETIME IS LIMITED ONLY TO THE PROCEDURE BEING EXECUTED.	27
11	GETTING UP TO SPEED	29
11.1	FLOATING POINT CODE VECTORIZATION	29
11.2	BLOCK BASED PROCESSING	30
11.3	COMMON PITFALLS	32
11.4	CODE VECTORIZATION METHODS	34
11.5	ENABLING THE EXPRESSIONS FOR MULTI-CORE CPU'S	35
11.6	INTEL COMPLEX NUMBER VML FUNCTIONS.....	35
11.7	MULTITHREADING FFT	36
11.8	MANAGING THE THREADS	36
12	DEBUGGING MTXVEC	36
12.1	DEBUGGER VISUALIZER.....	36
12.2	VIEWING THE VALUES OF VECTOR AND MATRIX IN THE DEBUGGER AS AN ARRAY	37
12.3	PRINTING CURRENT VALUES OF VARIABLES	38
12.4	DISPLAYING THE CONTENTS OF THE TVEC AND TMTX	38
12.4.1	<i>As a delimited text</i>	38
12.4.2	<i>Within a grid</i>	38
12.5	CHARTING AND DRAWING	39
12.6	MEMORY LEAKS.....	40
12.7	MEMORY OVERWRITES	40
13	GETTING READY TO DEPLOY.....	42
13.1	COMPACT MTXVEC	42
14	64BIT VERSION OF MTXVEC	42
15	USE UP TO 4GB OF MEMORY FOR 32BIT APPLICATION.....	43
16	MAJOR FUNCTION GROUPS	44
16.1	BASIC VECTOR MATH	44
16.2	STATISTICAL	46
16.3	COMPLEX NUMBER SPECIFIC	46
16.4	SIZE, STREAMING AND STORAGE	47
16.5	FFT'S.....	47
16.6	LINEAR ALGEBRA.....	47
16.7	MATRIX CONVERSIONS	48
16.8	MISCELLANEOUS MATRIX ROUTINES.....	48
17	COMPATIBILITY BREAKING CHANGES FROM VERSION 1.X, 2.X	49

1 Introduction

MtxVec is the most powerful and complete scientific software library available to Delphi, C++Builder and .NET users. It is an object oriented vectorized numerical library and adds the following capabilities to your development environment:

1. Comprehensive set of mathematical and statistical functions
2. Substantial performance improvements of floating point math by exploiting the SSE2/SSE3 and SSE4 instruction sets and multiple CPU cores.
3. Improved compactness and readability of code.
4. Significantly shorter development times by protecting the developer from a wide range of possible errors.
5. Automatic threading inside of the specific algorithms.

MtxVec makes extensive use of Lapack. Lapack is short for Linear Algebra Package and was originally called Linpack. Lapack is today de-facto standard for linear algebra and is free (www.netlib.org). Because Lapack is standard, different CPU makers provide performance optimized versions of Lapack to achieve maximum performance. Because linear algebra routines are the bottleneck of many frequently used algorithms, Lapack is a part of code that makes most sense to optimize. MtxVec uses the Lapack version optimized for latest CPU's provided by Intel with their Math Kernel library. MtxVec will also take advantage of all features of AMD CPU's.

MtxVec also makes extensive use of Intel Performance Primitives, which accelerate mostly not linear algebra based functions.

MtxVec will run on all Intel x86 compatible CPU's old and new, but will achieve highest performance on latest CPU generation.

1.1 Why MtxVec

- Natural math expression syntax for vectors and matrices with full support for operator overloading for vectors, matrices and complex numbers.
- Vector and Matrix are value classes (records with methods) and free the user from the need to manually free the memory that they allocate.
- Low level math functions are wrapped in to simply to use code primitives.
- All primitives have internal and **automatic memory management**. This frees the user from a wide range of possible errors like, allocating insufficient memory, forgetting to free the memory, keeping too much memory allocated at the same time and similar.
- Parameters are explicitly **range checked**, before they are passed to the dll routines. This ensures that all dll calls are safe to use.
- When calling Lapack routines MtxVec automatically compensates for the fact that in FORTRAN the matrices are stored by columns and in other languages by rows.
- Many **LAPACK** functions take many parameters. Most of them can be filled-in automatically by MtxVec, thus reducing the time to study each function extensively, before it can be used.
- Organized in to a set of "primitive" highly optimized functions covering all the basic math operations, which are used by all higher level algorithms, in a similar way as the BLAS is used by LAPACK.
- Although some compilers support native SSE2/SSE3/SSE4 and AVX1/2 instruction set, the resulting code can never be as optimal as a hand optimized version.
- Many routines are multi-threaded, including 1D FFT, sparse matrix solvers, matrix multiply, large parts of Lapack and all basic math functions like sin, cos, Ln, Exp,...
- All MtxVec functions must pass very strict automated tests. It is these tests, which give the library the highest possible level of reliability, accuracy and error protection.
- All low level code is abstracted away from the user. This allows a very easy transition to any future platform supported by MtxVec.

1.2 How fast is MtxVec

Typical performance improvements observed by most users are 2-3 times for vector functions, but speed ups up to 10 times are not rare. The matrix multiplication for example is faster up to 20 times.

2 Organization

MtxVec library is organized in to three levels - computational level, objects level and component level. The interface to the computational level is a set of functions, which are declared in nmkl.pas, ippspl.pas, and other files and implemented as external DLLs - MtxVec.Lapack2d.dll, MtxVec.Spld3.dll, MtxVec.Sparse2d.dll, MtxVec.FFT.dll, MtxVec.Random.dll, MtxVec.Vml2d.dll and MtxVec.Vmld.dll. The first level is not documented. The second level is written in Delphi. This level introduces vector and matrix objects, complex numbers and a number of utility functions declared in: MtxVec.pas, Polynoms.pas, Math387.pas, Probabilities.pas, SpecialFuncs.pas, Optimization.pas, Toeplitz.pas, Sparse.pas and MtxVecTee.pas. The second level is the “run-time” part of the MtxVec library. Third level is formed by a set of components which are build on the second level to offer a centralized and quick access to large parts of the library many times also offering ready to use user interface.

This users guide concentrates mostly on the second level, specifically the MtxVec.pas, Math387.pas and MtxExpr units and touches some features from MtxVecTee.pas. It gives a good overview on the concept and core features of MtxVec.

MtxExpr.h, Math387.h, Sparse.h, MtxVec.h etc... contain the C++ extension code to better support the C++ language. They are automatically included when MtxExpr.hpp, Math387.hpp are included in to the source file and should not be specified explicitly. Any required static lib files are also automatically linked against when this includes are specified. They enable features as operator overloading when working with vectors, matrices and complex numbers.

2.1 Compiler support

MtxVec v4 supports the following compilers:

	Versions
Delphi W32	Delphi versions 4 and 5 are supported with MtxVec 1.5. Delphi 6,7 and 2005 with versions prior to v4. MtxVec v4 support Delphi 2006, 2007, 2009, 2010, 2011 and 2012
Delphi VCL.NET	Delphi 2007
C++ Builder	Version 6 with v2.1, MtxVec v4 supports BCB 2006/2007/2009/2010/2011 and 2012.
Visual Studio C# (VB.NET, C++)	2005.NET, 2008.NET, 2010.NET

3 Quick start

```
//1. Include MtxExpr.hpp in the working cpp file
#include "MtxExpr.hpp"

//2. You are now ready to write the code:

{
    TCplx ac;
    sVector av;
    sMatrix am,bm;
    cVector cv;
    cMatrix cm;

    TDoubleArray arr;
    int i;

    ac = StrToCplx("1+2i"); //Convert from string to complex number

    am = RandGauss(5,5,true); //5x5 complex matrix with Gaussian noise
    av = Mtxexpr::Ramp(25); //real vector = [0, 1, 2,....., 23, 24]

    bm = Exp(am)*av + Cos(ac) + 2; // (*,/) treat Matrices as vectors
    // Same as: ./ and .* operators (not linear algebra)

    //To make linear algebra multiplication and division use functions

    bm = Divide(am,bm) + 2; //matrix division
    bm = Mul(am,bm) + 2; //matrix multiply

    //of course you can mix matrices and vectors

    av.Resize(5);
    av = Mul(av,bm) + 2; //vector from left and matrix multiply
    av.CopyToArray(arr); //copy data to array of double
    av.CopyFromArray(arr); //copy data from array to vector

    am.Complex = false; //5x5 complex matrix with Gaussian noise
    for (i = 0; i < am.Rows; i++) //standard loop example
    {
        //in case of complex values
    //    cv[i] = cm[i,0]*cv[i];
    //    cv.CValues(i) = cm.CValues(i,0)*cv.CValues(i); //same as previous line

        //in case of real values
        av[i] = am[i,0]*av[i];
        av.Values(i) = am.Values(i,0)*av.Values(i); //same as previous lines
    }
}
```

4 MtxVec programming interface

4.1 Object hierarchy

MtxVec organizes mathematical data structures and methods in to objects to simplify memory management and increase ease of use and features the following class hierarchy:

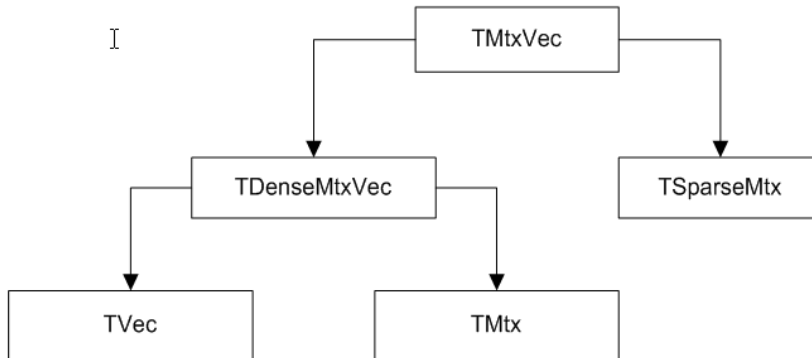


Figure 1 MtxVec class hierarchy

TVec and TMtx methods form the core of MtxVec library and offer a wide array of different routines. All methods and properties of TVec and TMtx are also accessible from sVector and sMatrix types which are declared as value classes in C++. Many TVec and TMtx methods are also available as stand-alone routines.

4.2 Smart pointers

By analogy with smart pointers in C++, MtxVec library defines shell classes in *MtxExpr.hpp* – sVector and sMatrix for TVec and TMtx classes respectively.

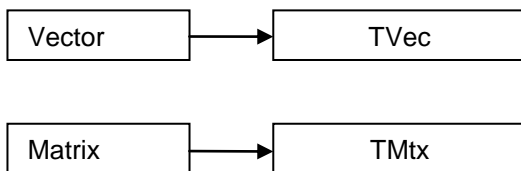


Figure 2 Mapping to value classes.

Pascal style forces to declare variables as below:

```

TVec* v;
CreateIt (v);
try {
    ...
    v->Sin();
    ...
} __finally {
    FreeIt (v);
}
  
```

C++ style looks much simpler:

```

sVector v;
...
v.Sin();
v->Sin();
  
```

...

Constructor creates actual object, destructor releases it and operator `->()` provides access to actual TVec object. Vector and Matrix classes use reference counting mechanism to optimize assignment operations and when working with subranges. sVector and sMatrix types are implicitly casted to TVec and TMtx respectively where required.

4.3 sVector, sMatrix, cVector, cMatrix, SparseMatrix

Classes sVector, sMatrix, SparseMatrix are smart shells for classes TVec, TMtx, TSpaseMtx respectively. Classes cVector and cMatrix are defined to allow a more comfortable access to complex elements, but are otherwise identical as sVector and sMatrix.

For C++ additional classes TVector and TMatrix are introduced. They are inherited from TVec and TMtx respectively to enable faster access to elements via Values and Values1D properties than TVec/TMtx classes can provide. Also the classes are used for backward compatibility with the previous versions of MtxVec library.

In practice, there is no need to use TVec, TMtx, TVector, TMatrix or TSpaseMtx classes. sVector, sMatrix and SparseMatrix should suffice.

Examples of definitions:

```
sVector v1(10); // real vector of ten elements
sVector v2(10,true); // complex vector of ten elements
cVector v3(10); // complex vector of ten elements

sMatrix m1(10,10); // real matrix 10x10
sMatrix m2(10,10,true); // complex matrix 10x10
cMatrix m3(10,10); // complex matrix 10x10
```

4.4 Method conventions of TVec and TMtx

TMtxVec classes and descendants (TVec, TMtx, TSpaseMtx) have a list of methods which may be called like this:

```
vector_object_a->Add(vector_object_b);
```

The object can hold either real or complex double precision data. If the method can put its result in one object, then the result is placed in the object on the left. In the following example the result is placed in the objects on the right:

```
a->CplxToReal(Real, Imag);
a->CartToPolar(Amplt, Phase);
```

It's useful to remember this when mixing TVec and TMtx types. A matrix operation which has TVec type as result will be a part of the TVec class and a vector operation which has a TMtx type result will be a part of TMtx class.

4.5 Methods and functions

Almost all methods can be replaced with functions. The next line:

```
v->Sin(x);
```

Is equivalent to:

```
y = Sin(x);
```

and to

```
y.Sin(x);
```

All standard function from $$ header file are also defined. So, the next syntax is allowed as well:

```
y = sin(x);
```

All those methods of TVec and TMtx are also available as functions in MtxExpr.hpp, where the result of the function is a single Vector (TVec) or a single Matrix (TMtx). These are equivalent to the methods of TMtxVec and its descendants.

4.6 Real number data type

```
{#ifndef TDOUBLE} type TSample = double; {#endif}
{#ifndef TSINGLE} type TSample = single; {#endif}
```

By using a DEFINE statement in bdsppdefs.inc file, the precision in which MtxVec runs can be switched between double and single.

4.7 Complex number data type

The complex data type is defined as:

```
struct TCplx
{
    double Re;
    double Im;
}
```

Both sVector (TVec) and sMatrix (TMtx) can hold real and complex data. Here is an example:

```
a.Length = 10;
a.Complex = true;
```

a.Length now becomes 5. Setting the complex property will simply halve or double the length property of the vector. The allocated memory will not change. There is a need however to view that memory as a real or as a complex array:

```
a.Values(0) = 1;
a.CValues(0) = Cplx(2,3);
```

a.Values(0) now becomes 2, because both Values and CValues arrays point to the same memory. The only difference between them is that one is of type double and the other is of type TCplx (struct TCplx { double Re,Im; }).

Real and complex Vectors and Matrices and TCplx variables can be mixed together in expressions:

```
sVector GetSomething()
{
    sVector av;
    TCplx ac;

    ac = StrToCplx("1+2i"); //Convert from string to complex number
    av.Size(10); //vector size
    return (av + ac + 2); //always by value operations
    // ./ and .* (not linear algebra)
}
```

4.8 Operator overloading

Classes Vector and Matrix overload many operators: +, -, *, /, =, etc. They enable the use of vectors and matrices in arithmetic expressions next to integers, doubles and complex numbers.

```
sVector v(10); // vector of ten elements;
v=1.23;      // assign 1.23 to each element of vector
v=v+v;      // double each element
v=v*2;      // double each element again
v*=2;       // and again
```

You can also mix TVec/TMtx and Vector/Matrix in the same expression. An expression will accept TVec and TMtx as a variable only, if the variable next to it is of Vector or Matrix type:

```
inline sVector test1(TVec *p)
{
    sVector bv;
    bv->Copy(p);

    return (bv*p); //same as: return Sqr(p);
//    return (p*p); //that would not work
}
```

To get access to elements of the vector we can use operator[] (int):

```
v[0] = v[1] + v[2];
```

The same, but longer:

```
v.Values(0) = v.Values(1) + v.Values(2);
```

However, CValues should be used to accessing to complex numbers:

```
v.CValues(0) = v.CValues(1) + v.CValues(2);
```

To access the contents as an array of integers, use the IValues method.

```
v.IValues(0) = v.IValues(1) + v.IValues(2);
```

The use of Delphi based properties Values[], CValues[], IValues[] is to be avoided. They can cause various side effects and have a large performance overhead. sMatrix also requires the long form to access the elements:

```
sMatrix m(3,3);

m.Values(0,0) = m.Values(1,1)+ m.Values(1,2);
m[0] = m[1] + m[2]; //work with matrix as with array
```

Comparative operators <, >, <=, >=, ==, != store the result of the comparison in a vector, each element of which is an integer value of "1" or "0". The default operators in Delphi support only a single Boolean value as a result of the comparison.

```
sVector a(3),b(3),r;
a[0]=7;a[1]=8;a[2]=9; // a = [ 4,5,6 ]
b[0]=3;b[1]=2;b[2]=1; // b = [ 6,5,4 ]
r = a < b;           // r = [ 0 ]
r = a == b;         // r = [ 1 ]
r = a >= b;         // r = [ 1,2 ]
```

To make the use of the values stored in r:

```
r = a >= b;
c->GatherByIndex(a,r); // c = [5,6]
```

Classes Vector and Matrix also have operators, which implicitly convert object to string:

```
AnsiString vs = v; // vector to string
std::string ms = m; // matrix to string
```

4.9 Element by element sVector/sMatrix expression types

All operators, which are defined for Vector and Matrix classes, work with elements. The next expression applies multiplication element by element instead of multiplying two matrices:

```
Matrix a(2,2),b(2,2),c;

// a is (1,2,3,
//       4,5,6,
//       7,8,9);
// b is (1,2,3,
//       4,5,6,
//       7,8,9);

c = a * b;

// c is (1,4,9,
//       16,25,36,
//       49,64,81);
```

To apply multiplication of two matrices it's necessary to call method `TMtx::Mul`:

```
c.Mul(a,b);

// c is (30,36,42,
//       66,81,96,
//       102,126,150);
```

or to use the dedication function, with the same meaning:

```
c = Mul(a,b);
```

To multiply two vectors element by element:

```
sVector a,bv,av;

a = av*bv;

//Both vectors must have the same length or an exception will be raised;
//either or both of the vectors can be complex.
```

To multiply vector and matrix element by element:

```
sVector av,a;
sMatrix bm;

a = av*bm;

//Both vector and the matrix must have the same number of elements or an
//exception will be raised;
//either or both of the av and bm can be complex.
```

The same principles are used for addition (+), subtraction (-) and division. It is of course possible to mix sMatrix and sVector types with integers, doubles, reals and complex numbers in the expressions.

4.10 Linear algebra Vector/Matrix expression types

Matrix, Matrix multiply: $C = A * B$, `C = Mul(A,B);`

The TMtx.Mul method also features parameters which make it possible to implicitly transpose or adjungate one or both matrices.

Matrix, Vector multiply: $b = A*x$, `b = Mul(A,x);`

Vector, Matrix multiply: $b = x*A$, `b = Mul(x,A);`

Vector, Vector multiply: $A = b*x$, `A = Mul(b,x);`

Vector, Matrix divide : $b = x/A$, `b = Divide(x,A);`

Matrix, Matrix divide : $B = C/A$, `B = Divide(C,A);`

Vector, Matrix ldivide : $b = A\backslash x$, `b = LDivide(A,x);`

Matrix, Matrix ldivide : $B = A\backslash C$, `B = LDivide(A,C);`

4.11 Linear algebra with TVec and TMtx types

When working with TVec vectors and TMtx matrices in most cases the type of the result defines the object which has that method. A few examples:

Matrix, Matrix multiply: $C = A * B$, `C->Mul(A,B);`

The TMtx.Mul method also features parameters which make it possible to implicitly transpose or adjungate one or both matrices.

Matrix, Vector multiply: $b = A * x$, `b->TensorProd(A,x);`

Vector, Matrix multiply: $b = x * A$, `b->TensorProd(x,A);`

Vector, Vector multiply: $A = b * x$, `A->TensorProd(b,x);`

Sparse matrix, vector multiply: $B = S * b$, `S->MulRight(b,B);`

The sparse matrix has that method and it returns the result in the second parameter. Sparse matrix is in this case an exception to the initial rule.

Vector, sparse matrix multiply: $B = x*S$, `S->MulLeft(x,B);`

Matrix, sparse matrix multiply: $B = A*S$, `S->MulLeft(A,B);`

Sparse matrix, matrix multiply: $B = S*A$, `S->MulRight(A,B);`

4.11.1 Other types of operations

Matrix addition or subtraction is straightforward with the Add method. Sparse matrix features specialized routines for this purpose also. To obtain a diagonal of a matrix there is a sVector.Diag method. To set a diagonal of a matrix there is a sMatrix.Diag method. To get a row/column of the matrix call sVector.GetRow (sVector.GetCol) and to set one: sMatrix.SetRow. (sMatrix.SetCol).

Many other methods follow the same pattern. The exceptions are usually methods which return multiple variables as a result and their overloads. A few examples: sMatrix.Eig(), sMatrix.SVD, sMatrix.LQR.

4.12 Indexes, ranges and subranges

Most TMtxVec methods support indexing. Here is a typical pattern that can be observed throughout the library:

```
TMtxVec->Exp();
TMtxVec->Exp(TMtxVec *X);
TMtxVec->Exp(int Index, int Len);
TMtxVec->Exp(TMtxVec *X, XIndex, int Index, int Len);
```

The first function version takes no parameters. The result overwrites the source data. The source data can be either real or complex and the method will apply the appropriate code to compute the result. The second version first checks the size of the source objects and tries to match the destination object to be of the same size. If the size operation is successful, the appropriate code is applied to compute the result. (See chapter 5.4 on how the size operation is performed).

The third version takes only Len values starting at Index. If the object is a 2D matrix and has 10 rows and 13 columns, its Length property is 130. The routine check's if Index and Len are within limits and applies the Exp function only to Len elements starting at position Index. To apply the Exp function to all elements within the matrix, the Index would be set to 0 and Len would be set to 130. Except for some exceptions, most indexed methods (methods that have Index and Len as a parameter) will raise an exception if the destination does not have a matching value of the Complex property.

One other important thing to mention about fourth version of the function is that the destination size is never changed. The only function version changing the size of the destination is the second. Both third and fourth function versions just perform error checking. An easy to remember rule: **All methods taking Index and Len parameters never change the size of the destination.** For vector this means Length, for the matrix rows and cols properties and for the sparse matrix rows, cols and nonZeros. There are some exceptions that allow changing the value of the complex property. **Add, Sub, Mul, Div, Offset and Scale methods allow mixing of real and complex data even for indexed methods.** (This was new in v2.0). If the result is to be complex, but the destination stores values of real type, all the destination values that are not to be overwritten will be converted to complex numbers with imaginary part set to zero. These automatic conversions are done in the most optimal way possible.

More examples:

```
a->Copy(b, 2, 0, 10);
```

'a.Copy' means 'copy' 10 elements of "b" from index 2 to "a" starting at index 0 of a. If there are no index parameters, the size of the target object will be set automatically. An alternative means for indexing is to use SetSubIndex or SetSubRange methods:

```
b->SetSubRange(2, 10);
a->Copy(b);
```

Which is the same as:

```
b->SetSubIndex(2, 11);
a->Copy(b);
```

The use of SetSubRange and SetSubIndex is recommended because it employs memory reuse, which takes advantage of the CPU cache, which in turn improves performance. SetSubRange can be called on object itself or it can obtain a view of memory from another object:

```
b_vec->SetSubRange(aMatrix, 2, 10);
a->Copy(b_vec);
```

This is the same as:

```
a->Size(10);
a->Copy(aMatrix, 0, 2, 10);
```

There are other types of indexing where there is a need to apply an operation to specific non-continuous indexes within a vector or matrix. This can result in heavy performance penalties (heavy means by a factor of 100-300) for some numerical algorithms. The entire CPU architecture is based on the assumption that memory is accessed by consecutive memory locations in about 90% of cases. It is therefore best to first gather the scattered data into one dense vector, perform math operations and then scatter the gathered data back to the original location:

```
a->Gather(b,nil,indIncrement ,2);
a->Log10();
a->Exp();
b->Scatter(a,nil,indIncrement,2);
```

This code will copy every second element from b to a, apply math and then scatter the result back to b without affecting other values in b. The Gather and Scatter methods can also accept an index or a mask vector. To access elements of an index vector via IValues array:

```
a->IValues[0] := 1;
```

IValues points to the same memory as Values and CValues and uses a simple integer array type pointer. Although TMtxVec can hold an array of integers, there are no functions that support operation on integers other than copy operations. There are more routines that can help with scattered data:

```
a->FindMask(b, "=", c);
```

The method will return ones for all indexes where b and c have a matching value and zeros elsewhere. FindAndGather can be used to find all indexes within b where values are different from NAN (not a number) and apply processing only to those values:

```
a->FindAndGather(b, "<>", NAN, Indexes);
a->Scale(2);
b->Offset(1);
a->Scatter(b, Indexes);
```

4.13 Subranges with operator overloading

MtxVec library allows the programmer to work with a selected range of the vector or matrix. For this purpose the following functions exist: SetSubrange / SetSubindex / SetFullRange. However, C++ syntax allows the use of more compact constructs. For example, operator()(int,int) of Vector represents a view of a part of the source vector:

```
sVector v(6);
v.Ramp(); // 0,1,2,3,4,5
v(0,1)=v(2,3)+v(4,5); // v[0]=v[2]+v[4], v[1]=[3]+v[4]
// v is 6,8,2,3,4,5
```

Another example is operator()(int) of Matrix, which obtains a vector view of a row of the matrix:

```
sMatrix m(3,3); // matrix 3x3
m.SetZero(); // set all elements of matrix to zero
m(1)->Ramp(1,1); // set second row to 1,2,3 (TVec method!)
// m is (0,0,0,
//       1,2,3,
//       0,0,0);
```

Operator()(void) maps the whole matrix to vector:

```
sMatrix m(3,3); // matrix 3x3
m()->Ramp(); // m is (0,1,2,
//                  3,4,5,
//                  6,7,8);
```

```

m() *= 2;      // double each element
              // m is (0,2,4,
              //      6,8,10,
              //      12,14,16);

```

Subrange operators return objects which refer to the memory of the original object. So any modification to the subranged object causes changes in the original object. This is valid for all subrange operators except one case: `sMatrix.Operator() (int,int,int,int)` which returns a submatrix which is not merely a view of the original, but a copy of selected elements.

```

sMatrix m(3,3); // matrix 3x3
m.SetZero();   // set all elements of matrix to zero
m(1,1,1,1)=1;  // does NOT change central element of m object

```

`Operator() (int,int,int,int)` copies a sub-matrix defined with four parameters: (RowIndexStart, ColumnIndexStart, RowIndexEnd, ColumnIndexEnd).

4.14 Range Checking

All Value access methods are range checked, when debugging is active. `MtxVec.h` holds the following definition:

```

#ifdef _DEBUG
#define MTXVEC_RANGE_CHECKING 1
#else
#undef MTXVEC_RANGE_CHECKING
#endif

```

Active range checking can have a noticeable effect on performance.

4.15 Open array parameters and SetIt method

One other important issue with C++Builder are open array parameters. Delphi allows passing arrays to routines without the need to first allocate them explicitly. Because many examples in `MtxVec` documentation use the `SetIt` routine an example below shows how the `SetIt` routine can be used in C++Builder:

```

//-----
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
#include "MtxExpr.hpp"
#include "MtxVec.hpp"
#include "MtxVecTee.hpp"
#include "MtxVecEdit.hpp"
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

//-----

void __fastcall TForm1::Button1Click(TObject *Sender)
{
    sVector x, y, spec;
    sMatrix a, b, c;

    // Setting a matrix to a vector

    double M[4][4] = { {1 , -3 , 5, -3},
                      {-1 , 12 , 0.3, 2.5},
                      {5 , 1.22, 2.33, -0.5},

```

```

                {2.4, -1 , 5, 3} };
int FinalIndex = 4*4-1; //array is zero based.
y.SetIt(false,&M[0][0],FinalIndex);

ViewValues(y,"Vector",true);

//    Setting a matrix to a matrix

a.SetIt(4,4,false,&M[0][0],FinalIndex);
ViewValues(a,"Matrix",true);

//    Setting a vector to a matrix

double Mv[16] = { 1 , -3 , 5, -3,
                 -1 , 12 , 0.3, 2.5,
                 5 , 1.22, 2.33, -0.5,
                 2.4, -1 , 5, 3};

a.SetIt(4,4,false,&Mv[0],FinalIndex);
ViewValues(a,"Matrix from Vector",true);

}
    
```

4.16 Loading and saving a matrix in a text file.

One more example:

```

//-----
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
#include "MtxExpr.hpp"
#include "MtxVec.hpp"
#include "MtxVecEdit.hpp"
//-----
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

//-----
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    sVector x, y, spec;
    sMatrix a, b, c;

    TStringList *list = new TStringList();
    try
    {
        //reading a text file

        list->LoadFromFile("C:\\test.txt");
        a.StringsToValues(list," ");
        ViewValues(a,"A matrix",true);

        //changing values of the matrix

        a.Values(0,0) = 1;
        ViewValues(a,"First element changed",true);
        a.Values(0,0) = a.Values(0,1);
        ViewValues(a,"First element changed again",true);

        //setting values of a vector
    
```

```

double Mv[10]={1 , -3 , 5, -3, -1 , 12 , 0.3, 2.5, 5 , 1.22};

int FinalIndex = 10-1; //array is zero based.
y.SetIt(false,&Mv[0],FinalIndex);
ViewValues(y,"Y vector",true);

a.Size(5,10,false);
for (int i = 0; i < 5; i++) a(i) = y; // a->SetRow(y,i)

ViewValues(a,"A matrix with 5 vectors in row",true);

double c1 = 1.5; //define scalar
b = a; //get some values to b

b = a*c1 + b; // calls lapack daxpy, adds A to B scaled by c1

ViewValues(b,"b = a*c1 + b", true);

//alternative executes equally fast as the expression:

b.Copy(a);
a.Scale(c1);
b.Add(a);

ViewValues(b,"b = a*c1 + b", true);

list->Clear();
b.ValuesToStrings(list," ", "", "");
list->SaveToFile("c:\\test1.txt");
}
__finally
{
    delete list;
}
}
    
```

While C++Builder 6 does correctly assign all default values to parameters, it does not display in the *code insight* that the routine does have default values assigned to some parameters. One other potential bug are also enumerated types. If variables of enumerated types are passed to the routine, the programmer must ensure that the variables are initialized. Out-of-range errors for enumerated types will not be caught in C++Builder.

5 Advanced Topics

5.1 Preallocated objects

Vector and Matrix classes constructors have a default parameter named `preallocated` which is true by default. MtxVec library has a pool of preallocated vectors and matrices. The pool speeds up object creation but has a limited number of objects preallocated and is intended for objects with short life cycle. Long time living object should be created with the `preallocated` flag set to false. This saves pool slots, which finally improves execution times. Allocate object from the pool:

```
sVector a(10); //real vector
```

Allocate static object:

```
sVector a(10, false, false); //real vector
```

The first boolean flag is for real/complex and the second is for pool selection.

5.2 Without redundant copying

Classes sVector and sMatrix avoid extra memory copying in arithmetic expressions. Usually, temporary objects are used in arithmetic expressions as a result of an operator.

```
sVector a,b,c;
...
a=b+c; // "b+c" creates temporary object, value of
        // which will be assigned to "a" object
```

Assignment operator copies only reference to actual vector and does not copy vector contents. For large vectors this saves valuable time. Complicated arithmetic expressions also don't spend time for memory copying:

```
sVector a,b,c,d;
...
a=a+b*2+sin(c*d); // executes without redundant copying!!
```

This expression will be translated to the following MtxVec calls:

```
Temp.sin(c*d);
temp2.Mul(b,2);
temp3.Add(temp, temp2);
temp4.Add(a, temp3);
```

Things to notice:

- there are no copy operations
- nearly all TVec and TMtx methods are SSE2/SSE3/SSE4/AVX optimized and vectorized, including the sine function.
- no TVec/TMtx objects are created or destroyed and if the Length of the Vector is less than the amount of preallocated memory, no memory is freed and/or allocated either.

Result: Execution times which beat today's best optimizing compilers (specifically Intel) in a number of applications.

5.3 More about subranges

There are cases when assignment operator really does copy the memory. This happens for subranged objects.

```
sVector a;
...
a(0,1)=a(2,3)+a(4,5);
```

All subranged objects modify original data, if they are used as lvalue (left side of the assignment). Assignment operator copies subranged objects by value (not by refs). However, there is one case, which requires to be mentioned separately:

```
sVector a(4); a.Ramp(1,1); // a is 1,2,3,4
```

Case 1:

```
sVector b;
b = a(0,1); // makes copy, b is 1,2
            // a and b point to different vectors
b[0] = 5; // does not change vector "a", b is 5,2
          // a is 1,2,3,4
```

Case 2:

```
sVector b(a(0,1)); // does not make a copy
                  // a and b point to the same vectors
b[0] = 5; // changes vector "a", b is [5,2]
          // a is 5,2,3,4
```

Case 3:

```
Vector b = a(0,1); // does not make a copy
                  // a and b point to the same vectors
```

```
b[0] = 5;      // also changes vector "a", b is 5,2
              // a is 5,2,3,4
```

The third case call's copy constructor (not assignment operator) and vector "b" points to subranged object.

Recommendation: Do not explicitly call the copy constructor or assign objects in the same line where they are declared.

5.4 Making use of the TMtxVec abstract class

Many TVec and TMtx methods can accept any TMtxVec descendant class:

```
TVec->Copy(TMtxVec *Src);
```

When a parameter is of TMtxVec or TDenseMtxVec type, the function will accept TVec, TMtx, sVector and sMatrix types. This is one of the most powerful features of MtxVec:

- When the source is vector, the vector size and its data are simply transferred to the calling object.
- When the source is 2D matrix, the Rows and Cols information is lost and the entire matrix is copied as if it is a vector.
- When the source is a sparse matrix, only the non zero elements are copied, while the non-zero sparse pattern and the number of rows and columns is lost.

The following versions will also work flawlessly:

```
TMtx->Copy(TMtxVec* Src);
TSparseMtx->Copy(TMtxVec* Src);
```

but with one slight difference. If the source is of the same type as the destination, the method also sets the size of the destination object:

- When the source and destination are TMtx (2D matrix), the method sets Rows, Cols and Complex property of the destination and copies all data values from the source.
- When the source and destination are TSparseMtx (2D sparse matrix), the method sets Rows, Cols, non-zero sparse pattern and Complex property of the destination and copies all data values from the source.
- When the source and destination are TVec (1D vector), the method sets the Length and Complex properties.

If the source and destination are not of the same type, the data is copied as if the source is a vector. No exception is raised only, if the source and the destination have a matching Length and Complex properties.

If only the complex property is to be changed, but all the other properties describing the data preserved, the following method can be called:

```
TMtxVec->Size(TMtxVec *Src, bool aComplex);
```

Note: The size method does not preserve the data in the destination object. This is not needed because the destination is overwritten anyway.

To allow such level of abstraction, the TMtxVec class introduces several methods that allow working with the data of descendants as if it was a simple one dimensional array of values:

TMtxVec.Values1D - array property to access real values

```
aTMtxVecObject->Values1D[1] = 1; //sets real value at index 1 to 1
//aTMtxVecObject can be TVec, TMtx, Matrix or Vector
```

TMtxVec.CValues1D - array property to access complex values

```
aTMtxVecObject->CValues1D[1] = Cplx(1,0); //sets complex value at index 1 to 1
```

TMtxVec.PValues1D - function returns a pointer to real value

```
double *aPointer;
...
aPointer = aTMtxVec->PValues1D(1); //returns a pointer to value at index 1
```

TMtxVec.PCValues1D - function returns a pointer to complex value

```
TCplx *aCplxPointer;
aCplxPointer = aTMtxVec->PCValues1D(1); //returns a pointer to complex
```

5.4.1 Writing abstract class code

This is best examined by an example. The following method can accept TVec, TMtxVec, Vector or Matrix and Dst does not have to have the size preset:

```
void CustomExpj(TMtxVec *Dst, TMtxVec *SrcOmega)
{
    Dst->Size(SrcOmega, true);
    CustomExpjNoSize(Dst, SrcOmega);
}
```

The “abstract magic” is achieved by calling the Size method. This method is “virtual” and implements all the required behavior when setting the size of the destination. CustomExpjNoSize in the last line just fills the destination with the result. It is important to note the Size method allows the user to change the Complex property without knowing the actual object type and by preserving all other property values. The True flag passed to the Size method sets the Complex property to True and is optional (default is false).

Of course not all functions can accept abstract object types. For those that don't it is possible to narrow down the required type to either TVec, TMtx or TSparseMtx. If the function should accept only TVec and TMtx, but not TSparseMtx, request that the parameter should be of TDenseMtxVec type. Methods and properties that are to be used for abstract MtxVec code:

- Pointers: PValues1D, PCValues1D, PValues1D,
- Getting/settings values: Values1D, CValues1D, IValues1D
- Setting size: Complex, Length, Size(TMtxVec *Src, bool IsComplex);
- all methods of TMtxVec class.

By making use of the TVec.SetSubRange method, virtually any TVec method can be applied to the source data:

```
void CustomExpj(TMtxVec *Dst, TMtxVec *SrcOmega)
{
    Vector a;

    Dst->Size(SrcOmega, true);
    a->SetSubRange(Dst);
    a->Expj(SrcOmega); //call a TVec only method here
}
```

5.5 Function parameters

It is recommended to declare function parameters as TMtxVec, TDenseMtxVec, TVec, or TMtx and not of sVector or sMatrix type. Regardless if the Vector or Matrix type are passed with a var parameter or not, they will always behave as if passing an object; by reference.

5.6 Pool of preallocated vectors and matrices

For Vector and Matrix classes, this is done automatically. TVec and TMtx objects however can be created and destroyed in the standard way:

```
TVec *a = new TVec();
TMtx *b = new TMtx();
try
{
    ....
}
finally
{
    delete a;
    delete b;
}
```

Or in a fast way, by using object cache:

```
CreateIt(a);
CreateIt(b);
try
{....
}
finally
{
    FreeIt(a);
    FreeIt(b);
}
```

Object cache is a set of objects, which are created when the application is started. When a call to CreateIt is made, no object actually gets created. The CreateIt procedure simply assigns a pointer to an already created object to the parameter. That is not all since the already created object has some memory allocated and there is no new memory allocated until some default size is exceeded. This type of memory allocation (call it preallocation) is speedier by a factor of 2 and in some cases even more. It is difficult to predict the actual effect on the entire application, where the gains could be significant. The use of object cache is not significant only because the calls to Create/Destroy and GetMem/FreeMem are never made, but also because it increases memory reuse.

6 Memory management

6.1 Introduction

The memory for the Matrix is allocated by setting the Rows and Cols properties:

```
sMatrix a;

a.Rows = 4; {allocates nothing}
a.Cols = 4; {a now holds 16 elements}

a.Rows = 0; {deallocates memory}

a.Size(4,4,false); //same as: a.Complex := False; a.Rows := 4; a.Cols := 4;
```

The complex property should be set before setting the Cols property. All arrays are zero based. (The first element is always at index 0).

Dynamic memory allocation of two dimensional matrices in Delphi is not done in one single block as expected by fortran routines. For that purpose, the Matrix uses it's own memory allocation to

dynamically allocate two dimensional arrays in a single continuous block of memory. Therefore, there are two more properties available from Tmtx and Matrix:

```
a->Values1D[i]
a->CValues1D[i]
```

Pointers behind these two properties point to the same memory location as Values and CValues pointers. But instead of accessing the elements by rows and columns, they see the whole matrix as a one-dimensional array. To access matrix elements:

```
a1 = a->Values1D[i*Cols+j];
```

This will access the same matrix element as:

```
a1 = a->Values[i][j];
```

or

```
a1 = a[i,j];
```

The preferred method for memory allocation, is by using the Size method:

```
a->Size(4,4,false,false);
```

Size method will ensure that no more memory is allocated than necessary when resizing. Imagine a 5x10000 matrix, being resized to 10000x5, but the rows are set to 10000 first creating a matrix with 10000x10000 elements, possibly causing an out of memory message.

Matrix data is stored in row-major ordering of C and PASCAL and not in column major ordering of the FORTRAN language. All appropriate mappings are handled internally.

6.2 In-place/not-in-place operations

In case of very large matrices the memory requirements would become a problem (10 000 x 10 000 matrix requires 800MB storage). In such cases the user can use LAPACK routines directly by adding nmkl to the uses clause. Whenever possible LAPACK performs matrix operations in-place. Often the matrix size can be greatly reduced by using banded matrix format or sparse matrices.

7 Why and how NAN and INF

NAN is short for Not a Number and INF is short for infinity. By default Delphi will raise an exception when a division by zero occurs or an invalid floating operation is performed. By including Math387 unit in the uses clause, the floating point exceptions are automatically disabled. This allows code in loops without the checks, if the function input parameters are within the definition area of that function. This alone speeds up the code, because most of the try-except and if-then clauses used for that purpose can be left out. Instead, you can concentrate on the code itself and let the CPU work out the details. If a division by zero occurs and floating point exceptions are off, then the FPU (floating point unit) will return INF (for infinity). If divide zero by zero is attempted, the FPU will return a NAN (not a number). When working with arrays, this can be very helpful, because the code will not break when the algorithm encounters an invalid parameter combination. It is not until the results are displayed in the table or drawn on the chart that the user will notice that there were some invalid floating point combinations. It might also happen that INF values will be passed to a formula like this: number/INF (= 0) and the final result will be a valid number.

MtxVec offers specialized routines for string to number and number to string conversions in Math387 unit (StrToVal, StrToCplx, FormatCplx, FormatSample, StrToSample, SampleToStr) and drawing routines in MtxVecTee unit (DrawValues, DrawIt) capable of handling NAN and INF values. By using those routines, the user will avoid most of the problems when working with NAN and INF values. StrToSample for example will convert a NAN or INF string to its floating point presentation:

```
double a;  
a = StrToSample("NAN");  
if (IsNan(a)) throw "a = NAN";
```

StrToFloat routine would raise an exception on its own. To test for a NAN and INF value, the first attempt would look like this:

```
if (a == NAN) then ...
```

This however will not work. NAN and INF are not values which are defined with all the bits of a floating point variable. There are just a few bits that need to be set within a floating point variable which will make it a NAN or an INF. The proper way to test for a NAN and INF are therefore these:

```
if (IsNan(a)) ...  
if (IsInf(a)) ...  
if (IsNanInf(a)) ...
```

MtxVec methods and routines correctly handle NAN and INF. It is therefore acceptable to write something like this:

```
if (a->Find(Nan) > 0) ..//test if "a" vector holds a NAN value
```

7.1 Known issues

C++Builder versions 2009 and 2010 are known to modify the CPU exception mask when an exception is triggered and then handled inside of the catch clause. The exception mask is modified in such a way that an "invalid floating point operation" and "division by zero" exceptions will be raised when the code yields: 1/0 or 0/0. This behavior may cause some algorithms to break. The bug has been fixed in C++Builder 2011 compiler. The workaround for the older compilers is to always disable the floating point exception mask after the exception catch clause. There may be other situations where this mask is changed but that has not been researched in great depths.

8 Serializing and streaming

8.1 Streaming with TMtxComponent

All components should be derived from a common ancestor: TMtxComponent. (Declared in MtxBaseComp.pas). This component features the following methods:

- SaveToStream
- LoadFromStream
- SaveToFile
- LoadFromFile
- Assign
- AssignTemplate
- LoadTemplateFromStream
- SaveTemplateToStream
- LoadTemplateFromFile
- SaveTemplateToFile

Most C++Builder developers know the first five routines. What is interesting about TMtxComponent is that all components derived from it have all their published properties streamed, without the need to make any changes to the five routines. Therefore, all components derived from TMtxComponent have the capability to store their states (properties) to the stream, file or to assign from another object of the same type. The default Delphi component streaming mechanism has a bug when it comes to streaming properties with a declared default value. This has been fixed for TMtxComponent.

The "template" routines are a bit more advanced. They set a special protected property named BlockAssign to True. Property setter routines can then prevent properties to be changed. This is very useful when there is a need to save only "parameters" and not the "data" of the component. The parameters will remain the same, while the "data" will be different next time and there is no point in wasting disk space by saving it.

8.2 Streaming of TVec, TMtx and TSparseMtx

TVec and TMtx also have their own methods for streaming:

```
SaveToStream  
LoadFromStream  
SaveToFile  
LoadFromFile  
Assign
```

These routines will process all the published properties and data of the TVec and TMtx objects. These methods only save and load data in the binary format. (MtxVec also supports Matrix Market text file format.)

8.3 Reading and writing raw data

TVec and TMtx have the capability to save their state via SaveToStream and LoadFromStream. The downside on using these two routines is that it is not possible to save the raw data only. The values of all the properties are always included as the header of the saved data block. Saving raw data only can be achieved by using two other methods: *TVec.WriteValues* and *TVec.ReadValues*. These two methods will read and write to and from TStream descendants only the contents of Values array itself (raw data). When writing to stream, it is also possible to define the precision and consequently the size of the disk space to occupy. Supported precisions include:

```
TPrecision = (prDouble, prSingle, prInteger, prCardinal, prSmallInt, prWord,  
prShortInt, prByte, prMuLaw, prALaw, prInt24);
```

MuLaw and ALaw are audio compression standards for compressing 16 bit data to 8 bits. prInt24 is a 24 bit signed integer useful for 24bit digital audio.

When data is being read with ReadValues, the type of the data must be explicitly specified. All data types are converted to TSample.

9 Working with text

Sometimes it is necessary to read or write to and from a text file. Here is how this can be done:

9.1 Write Tmtx/Matrix to a text file

```
sMatrix AMtx;

AMtx = RandUniform(20,20,true,-1,2); //20x20 complex matrix
TStringList *StringList = new TStringList();
try
{
    AMtx->ValuesToStrings(StringList, (char) 9); // { use tab = as delimiter }
    StringList->SaveToFile("C:\\\\ASCIImtx.txt"); //{ Save matrix to txt file }
}
__finally
{
    delete StringList;
}
```

9.2 Read Tmtx/Matrix from a text file

```
sMatrix AMtx;

TStringList *StringList = new TStringList();
try
{
    StringList->LoadFromFile("C:\\\\ASCIImtx.txt"); // Load matrix from txt file
    AMtx->StringsToValues(StringList, (char) 9); // use tab as delimiter
    ViewValues(AMtx,"Data",true); // Display loaded values
}
__finally
{
    delete StringList;
}
```

9.3 Formatting floating point values – FloatToString

Math387 unit declares the following routines:

```
AnsiString FormatCplx(const TCplx &Z, const AnsiString ReFormat = " 0.###;-0.###",
                    const AnsiString ImFormat = "+0.###i;-0.###i")
AnsiString FormatSample(double X, AnsiString Format = "0.###")
```

Both are similar to FormatFloat and return a string representing the floating point number passed as the first parameter. If the Format parameter is an empty string, the routines will call the SampleToStr and CplxToStr routines. They are declared like this:

```
AnsiString CplxToStr(const TCplx &Z, const int Digits = 0x0, int Precision = 0xf)
AnsiString SampleToStr(const double X, const int Digits = 0x0, int Precision = 0xf)
```

and are similar to FloatToStr. The Digits parameter specifies the minimum number of digits in the exponent. The Precision defines the number of digits to represent the number. Example:

```
double a = 12.123456;
AnsiString myString = SampleToStr(a, 0, 3); // myString = '12.1';
```

```
double a = 12.123456;  
AnsiString myString = SampleToStr(a); // myString = '12.123456';
```

To convert from string to a floating point number, the following routines can be used:

```
double StrToSample(const AnsiString Source)  
TCplx StrToCplx(const AnsiString Source)
```

They are similar to StrToFloat with a few exceptions when it comes to handling NAN and INF values. (See the chapter: "Why and how NAN and INF".)

In general these routines improve the default string-to-number and number-to-string conversions by adding sensitivity to single/double precision and better support for NAN and INF values.

10 Programming style

Every programmer has a preferred style of programming: different indentation, different variable naming, different coding style (use of exceptions, for loops, dynamic memory allocation etc.). This section lists the recommended coding style for programming with MtxVec.

10.1 Mixing TVec/TMtx and Matrix/Vector types.

Handle Vector/Matrix as objects even though they are records. When writing new functions and methods, continue to declare their parameters as TVec or TMtx, because sVector and sMatrix will be implicitly converted (dereferenced) to TVec and TMtx anyway. This will reduce the total count of temporary variables which allocate objects from object cache. TVec and TMtx objects encapsulated by Vector/Matrix are obtained from object cache, which has limited size. This means that all the limitations and advantages of CreateIt/Freelt usage still apply.

10.2 Inlining of functions that use Vector and Matrix types

Functions that return Vector/Matrix or internally declare these types should be inlined for increased performance. This will reduce the count of temporary variables allocated by the compiler that obtain objects from object cache.

10.3 Raising exceptions

All code should be protected with try-finally blocks, exceptions can be raised safely to indicate an invalid condition. When the user tries to perform calculation with an MtxVec application, this is what will happen:

1. Allocate memory for the calculation.
2. Start calculation
3. Display results.
4. Free allocated memory and resources.

If during the calculation an error condition is encountered, because the data is not valid, raising an exception will first Free allocated memory and resources and then display a message box stating what the error was. It is important that memory was freed, because now the user can retry the calculation with new data or parameters, without the need to restart the application to reclaim the lost memory.

10.4 Invalid parameter passed:

```
if (a == null) throw "a= null";  
...
```

This will pass an exception to the higher-level procedures, which will free any allocated memory and exit. Once the exception reaches the highest-level routine a message box will be displayed with text "a = nil" and an OK button.

10.5 Use CreateIt/Freelt only for dynamically allocated objects whose lifetime is limited only to the procedure being executed.

All objects created within a routine should be destroyed within that same routine. If TVec or TMtx are global objects, make them a part of an object or component. Global objects are those, which are not created and destroy very often and might persist in memory throughout the life of an application. This rule should be followed in order not to waste the object cache. The purpose of object cache is to allow speedy memory allocation and deallocation. Where this is not needed, it should not be used, because that could slow down other routines using it:

- Object cache might run out of precreated objects and calls to Create/Free would result in direct calls to Create/Free.
- Object cache size would have to be increased to prevent (1) and the entire application would require more memory.

Vector and Matrix types allocate their internal TVec/TMtx objects via Create/Free by default. It is therefore recommended to use these types mostly for local variables within functions and procedures. To declare a global variable of this type, explicitly calling its constructors will avoid object cache:

```
Vector a(10, false, false); //real vector
```

11 Getting up to speed

11.1 Floating point code vectorization

MtxVec also allows the programmer to write high level object code that gives the benefits of the most optimized assembler version of the code supporting latest CPU instructions from within your current development environment. This is best examined on an example. Simply trying to use a faster Power function in the following loop will bring no major gains:

```
for (i = 0; i < 1000000; i++)
{
    Y[i] = (c1*Ax[i]+c2)/pow(1.0 + pow(Bx[i],eA), eB);
}
```

But if the above loop is rewritten like below things change a lot.

```
a->Length = 2000;
b->Length = 2000;
for (i = 0; i < 499; i++)
{
    YourFunc(a,b,c,c1,c2,ea,eb);
}

//By using expressions:
inline Vector YourFunc(TVec *a, TVec *b, double c1, double c2,
                      double ea, double eb)
{
    return (c1*A+c2)/Power(1.0 + Power(B,ea), eB);
}

//Using TVec:
inline void YourFunc(TVec *a, TVec *b, TVec *Result, double c1, double c2,
                    double ea, double eb)
{
    TVec *a1, *b1, *Result;

    if (a->Length <> b->Length) throw "a.Length <> b.Length";
    CreateIt(a1,b1); //work vectors
    try
    {
        a1->Copy(a);
        a1->Scale(c1);
        a1->Offset(c2);
        b1->Power(b,ea);
        b1->Offset(1);
        Result->Power(b1,-eb);
        Result->Mul(a1);
    }
    __finally
    {
        FreeIt(a1,b1);
    }
}
```

We can note that we wrote more lines and that we create and destroy objects within a loop. The objects created and destroyed within the function are not really created and not really destroyed. The CreateIt and FreeIt functions access a pool of precreated objects called object cache. The objects from the object cache have some memory pre-allocated. But how could so many loops, instead of only one, be faster? We have 7 loops (Copy, Scale, Offset, Power, Offset, Power, Mul) in the second case and only one in the first. This makes it impossible for any compiler to perform loop optimization, store local variables in the CPU/FPU, precompute constants. The secret is called SIMD or Single Instruction Multiple Data. Intel's and AMD CPU's support a special instruction set. It has been very difficult for any compiler vendor to try to make efficient use of those instructions and even today most compilers run

without support for SIMD with two major exceptions: Intel C++ and Intel Fortran compilers. SIMD supporting compilers convert the first loop of our case in to the second loop of our case. The transformation is not always as clean and the gains are not as nearly as large, as if the same principle is employed by hand. Sometimes it is difficult for the compiler to effectively brake down one single loop in to a list of more effective ones.

What is so special about SIMD and why are more loops required? The SIMD instructions work similar to this:

- load up to 4 array elements from memory (ideally takes 1 CPU cycle)
- execute the mathematical operation (ideally takes 1 CPU cycle)
- save the result back to memory(ideally takes 1 CPU cycle)

Total CPU cycle count is 3. The normal loop would require 1 cycle for each element to load, store and apply function (in best case). In total that would be 12 CPU cycles. Of course the compiler does some optimization in the loop, stores some variables in to FPU registers and the loop does not need full 12 cycles. Therefore typical speed ups for SIMD are not 4x but about 2-3x. However there are some implicit optimizations in our second loop too. Because we know that the exponent is fixed, the vectorized Power function can take advantage of that, so the gap is increased again. Of course, the first loop could also be optimized for that, but you would have to think of it.

11.2 Block based processing

When working with vectors it is absolutely critical to also consider the size of the CPU cache. If the arrays will not fit in the available CPU cache, a large (sometimes up to 3x) performance penalty will be imposed upon the algorithm. This means that vector arithmetic's should not be applied to vectors whose size exceed certain maximum length. Typically the maximum number of double precision elements ranges from 800 to 2000 per array. Longer vectors have to be split in pieces and processed in parts. MtxVec provides tools that allow you to achieve that easily. The following listing shows three versions of the same function.

Plain function:

```
double MaxwellPDF(double x, double a)
{
    double xx;

    if ((x >= 0) && (a > 0))
    {
        xx = Sqr(x);
        return Sqrt(4*a*INVTWOPI)*a*xx*Exp(-0.5*a*xx);
    }
    else return NAN;
}
```

Vectorized with expressions:

```
inline Vector MaxwellPDF(sVector x, sVector a)
{
    sVector tmp;
    tmp = Sqr(x);
    return (Sqrt(4*a*INVTWOPI)*a*tmp*Exp(-0.5*a*tmp));
}
```

Vectorized function:

```
void MaxwellPDF(TVec *X, double a, TVec *Res)
{
    TVec *Res1;

    CreateIt(Res1);
    try
```

```

    {
        Res1->Sqr(X);
        Res->Copy(Res1);
        Res->Scale(-0.5*a);
        Res->Exp();
        Res->Mul(Res1);
        Res->Scale(Sqrt(4*a*INVTWOPI)*a);
    }
    finally
    {
        FreeIt(Res1);
    }
}

```

Block vectorized function:

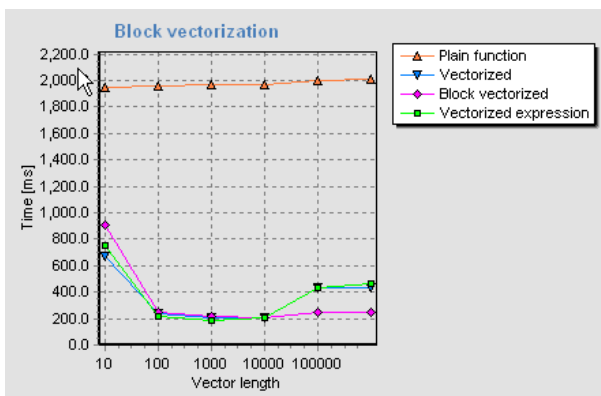
```

void MaxwellPDF(TVec *X, double a, TVec *Res)
{
    TVec *Res1;

    CreateIt(Res1);
    try
    {
        Res->Size(X);
        Res->BlockInit();
        X->BlockInit();
        while (!X.BlockEnd)
        {
            Res1->Sqr(X);
            Res->Copy(Res1);
            Res->Scale(-0.5*a);
            Res->Exp();
            Res->Mul(Res1);
            Res->Scale(Sqrt(4*a*INVTWOPI)*a);
            Res->BlockNext();
            X->BlockNext();
        }
    }
    finally
    {
        FreeIt(Res1);
    }
}

```

On P4 2.4 GHz CPU the vectorized function is about 9.5x faster than the plain function when using Delphi 2007. The block vectorized version of the function is a little slower for short vectors but maintains its high performance even for vectors exceeding 10 000 double precision elements. (For a CPU with 512kB CPU cache the limit is about 10 000 elements and if CPU with 128kB cache is used the limit is about 2000 elements.)



The block vectorized function is only marginally faster than vectorized version due to the use of SSE2/SSE3 instructions. If the CPU does not support SSE, then the gain of the block vectorized version will be much more significant (typical gains are about 6 times). For example, when using older CPU's the speed of the plain function for vectors with length larger than the size of the CPU cache will be higher than that of its vectorized version. The vectorized version has to access memory multiple times, while the plain function version can cache some intermediate results in to FPU registers or CPU

cache. The block vectorized version will ensure that the chunk of the vector being processed can fit in to the CPU cache and will thus give optimal performance for long vectors even in that case.

It is also worth noting that vectorized expression, if inlined, is nearly as fast as function vectorized with TVec objects even for vectors of only 10 elements.

11.3 Common pitfalls

1. When using block vectorization, make sure that the temporaries are “not” block vectorized. Only the input vector and the output vector are block vectorized. In the example with MaxwellPDF, it would be unnecessary to call BlockInit on Res1 TVec object. Typically block vectorization would be the last optimization to perform for the application and it would be applied to the top level function only. This allows the programmer to control the size of the blocks that are being processed throughout the algorithm from one central point. This is why no functions from TVec, TMtx, TDenseMtxVec and TMtxVec have been block vectorized.
2. Vectorization also increases the use of memory. Keeping the vectors short, will keep the memory usage low.
3. The default size of the block for vectors storing complex numbers should be less than 512, or it should not exceed the size of vector memory preallocated by object cache.
4. About 98% percent of functions available are SSE2/SSE3 vectorized but not all. It makes sense to have an algorithm available even if it is not executing with the highest performance. Specifically the following functions are not vectorized: all variants of Find (including FindIndexes, FindMask etc...), and some complex number versions of certain functions. The user is best advised to check the source code if in doubt. Future versions will include more vectorized functions. One way to get better performance with these functions is to make sure that block processing rules are always observed.
5. The trigonometric functions are extensively used in complex number math. It is important to be aware of some limitations of real valued sine and cosine functions. Their performance depends upon the size of the argument: $\sin(1)$ will be computed faster than $\sin(100000)$. This is true for standard FPU instructions and for the SSE versions. Together with the speed, the accuracy of the sine/cosine will be reduced also. If the number has 20 digits, only the last 10 numbers after the decimal point will remain valid. Math387 unit includes a utility function called FixAngle which should be called on the argument before it is passed to the sine function, but only if a “large” argument is to be expected. This will fix the accuracy and the speed problem. Of course, if the argument is not large, the speed will decrease and the accuracy will not be improved.
6. There are penalties on processing NAN and INF. Make sure those are fished out from the vectors soon after they might occur, to lower the performance cost on the follow up code.
7. For CPU's without SSE, the only way to improve the performance is to strictly follow the rules of block processing. (CPU cache size).
8. When running “quick” benchmark tests make sure to pass “valid” parameters to functions. If the function is not defined in a region, the result of the function will be a NAN or INF. All subsequent functions that will receive NAN or INF at the input could run much slower. This is the limitation of the SSE instructions. The program must be guarded against such cases explicitly. (Either by checking the user input or by inserting checks in the code that will abort the algorithm sooner if a NAN or INF is detected.). This can be important when running algorithms that already take a long time to compute with valid data.
9. Intel also warns about denormals. They are another cause for slowdown. Denormals are numbers which get truncated due to limited floating point number range. So again, the input to the algorithm when testing it, should be valid data.

Vectorized expressions specifics:

10. Do not forget to inline functions which use vectorized expressions. When using vectorized expressions it is recommended to inline most computationally intensive methods either manually or with the help of “inline” keyword. This will substantially reduce the number of temporary variables allocated by MtxVec and thus help keep the working set of the variables inside the CPU cache.
11. Delphi compiler does not support a specific optimization called: collection on common sub expression. If your expression contains the same expression multiple times, be sure to assign

its result to a temporary variable to prevent the expression from being evaluated multiple times.

12. Use parenthesis to indicate which part of the expression should be evaluated first. There is no optimization analysis that would be based on precedence of the same operator. Namely, multiplying two vectors or multiplying two real numbers is very different in terms of clock cycles used. But the compiler does not know that.

11.4 Code vectorization methods

Vectorizing the code means writing the code with the help of vector and matrix variables. Only when the code is written in such form has the compiler or the underlying library a chance to exploit the SIMD instruction set. With the evolution of CPU's such design will be bringing increasingly bigger gains over traditional code design. Vectorizing the code will provide by far the greatest performance boost keeping multi-core gains far behind in the shade. Therefore, if there is any chance to vectorize the code, it should by all means be attempted.

When vectorizing the code we have to rewrite all our functions so that they take input data in vector form and work around if-then sentences. If-then sentences are in-fact the biggest party breaker when it comes to code vectorization and it also makes sense to show an example of a method called "Vector patching", which allows very effective vectorization of a great deal of additional code.

Many times the code vectorization works great, except for a few special values which have to be handled separately. If those can not be moved out of the loop, make another loop following the first in which you only check for the special values using standard if-then sentences. Because the first computationally intensive loop has been vectorized, the extra loop patching up the vector is a really cheap way out. Below is an example of the Power function with vector patching. Notice that the vectorized part is followed by a separate loop patching up the result.

```
function TMtxVec.Power(Base: TMtxVec; Exponent: TCplx): TMtxVec; { X^Y, X >= 0 }
var a,b: TVec;
    i: integer;
begin
    Result := Self;
    if Self = Base then raise EMtxVecInvalidArgument.Create ('Self = Base');
    Size(Base, TRUE);

    if Math387.Equal(Exponent,0) then
    begin
        Result.SetVal(C_ONE);
        Exit;
    end;

    if fLength = 0 then Exit;

    if Base.Complex then
    begin
        vzPowx(Base.PCValues1D(0),Exponent,PCValues1D(0),Base.Length);

        for i := 0 to fLength-1 do
        begin //Resolve the special cases
            if Math387.Equal(Base.CValues[i],0) then
            begin
                if Exponent.Im = 0 then
                begin
                    if Exponent.Re > 0 then CValues[i] := C_ZERO else
                    if Exponent.Re < 0 then CValues[i] := Cplx(Inf) else
                    CValues[i] := C_ONE;
                end else CValues[i] := CNAN;
            end;
        end;
    end else
    begin
        CreateIt(a,b);
        try
            a.Ln(Base);
            b.Mul(a,Exponent);
            Exp(b);

            for i := 0 to fLength-1 do
            begin //Resolve the special cases
                if Base.Values[i] < 0 then
                begin
                    CValues[i] := Math387.Power(Cplx(Base.Values[i]),Exponent);
                end else
                if Base.Values[i] = 0 then
```

```

begin
  if Exponent.Im = 0 then
  begin
    if Exponent.Re > 0 then CValues[i] := C_ZERO else
    if Exponent.Re < 0 then CValues[i] := Cplx(Inf) else
      CValues[i] := C_ONE;
    end else CValues[i] := CNAN;
  end;
end;
finally
  FreeIt(a,b);
end;
end;

GCOperation(Base);
end;

```

This will work only if the special values are “rare”. If you have to distribute the processing down multiple paths nearly equally, things get complicated. In this case, one way out is to split the vector in to subvectors and store the indices of individual values within the vector separately. Once the processing of each separate part has completed, merge the individual parts together again. Good starting point for this approach are TVec.Gather and TVec.Scatter methods.

11.5 Enabling the expressions for Multi-core CPU's

Functions using Vector/Matrix types can also expect their expressions to execute on multiple CPU cores without any additional work. For example:

```

inline sVector MaxwellPDF(sVector x, sVector a)
{
  sVector tmp;
  tmp = Sqr(x);
  return (Sqrt(4*a*INVTWOPI)*a*tmp*Exp(-0.5*a*tmp));
}

```

The Exp function and the Sqrt functions called in this example will be automatically threaded in two threads, when the x Vector length exceeds about 5000 elements. However, there are some tricks here. Vector length required for effective threading in this case exceeds the size of the CPU cache for most CPU's. This means that we would lose more speed by not being inside the CPU cache than gain by having the two functions execute on two cores. Future hardware designs will bring bigger L1 and L2 cache sizes and shorter thread switching times.

On high-end CPU's with large L1 and L2 cache sizes, the typical working set of variables would not exceed the CPU cache size and performance gains would be noticeable. The following vector functions are threaded: Inv, Divide, Sqrt, InvSqrt, Cbrt, InvCbrt, Power, Exp, Ln, Log10, Cos, Sin, SinCos, Tan, ArcCos, ArcSin, ArcTan, ArcTan2, Cosh, Sinh, Tanh, ArcCosh, ArcSinh, ArcTanh, Erf, Erfc, ErfInv, Trunc, Round, Ceil, Frac.

On the LAPACK side entire BLAS3 is threaded, all FFT's including 1D, random generators and sparse matrix solvers. There is going to be increasing number of threaded functions in the future.

11.6 Intel complex number VML functions.

The complex versions of Intel VML functions, are several times slower from the ones implemented in MtxVec and even Math387. However when running on machines with 4 or more CPU cores, it is possible to use those functions, by commenting in {.\$DEFINE VML_COMPLEX} in bdsppdefs.inc. Those functions are namely multithreaded and when vectors are about 1000 elements long, they can already execute on up to 4 cores. On single and dual core CPU's MtxVec complex number vectorized functions will be faster. The benchmarks for any given CPU can be run with the help of the MtxVec Demo if needed: Vector Operations -> Benchmarks.

11.7 Multithreading FFT

FFTs are internally multithreaded. Threading can also be implemented in the users code. The following is to be considered when implementing custom threading:

- 1.) Allocate memory using Vector/Matrix or CreateIt/Freelit. Do not create new objects of type TVec or TMtx inside the loop which is threaded.
- 2.) Use not-in-place versions of the FFT functions. Sometimes the in-place versions require that internally the memory has to be copied.
- 3.) Make use of FFTFromReal and IFFTToReal where possible. Real to complex functions can be up to twice as fast as complex to complex.
- 4.) Stick to the FFT's which are power of two. FFT's which are not power of two can be 3..4x slower than with the size of the next larger power of two.
- 5.) When implementing custom threading, disable the FFT internal threading via the MtxVec.Controller variable.
- 6.) Limit the total number of different FFT sizes that you compute. This will reduce the memory requirements.
- 7.) Modern CPU's implement "turbo" mode, which makes one core run up to 50% faster (on mobile CPUs) than running all cores concurrently. Take this into account when comparing threading efficiency.
- 8.) A well optimized usage of an FFT running on one core can easily be faster than those running on multiple.

11.8 Managing the threads

The threads launched by MtxVec will default to the maximum core count in the system. By default the library decides on its own if, when and how many threads it will launch to speed up the processing. Some parameters however can be customized.

MtxVec declares a global variable inside of MtxVec unit called controller. This controller object has properties to control the object cache and threading. Specifically it is possible to set and read thread count for all subsystems like FFT, BLAS, VML and IPP. Additionally, it is also possible to enable or disable threading by setting ThreadingMode. Other information provided includes CPU cache size, CPU core count and CPU frequency.

Additional two properties are DenormalsAreZero and ThreadWaitBeforeSleep. Denormals are numbers which are smaller than the floating point range and thus clipped in precision. This property controls the CPU flag and affects all processing. This is similar to floating point exception flag, which is disabled by default. Forcing denormals to zero will substantially improve SSE performance, according to Intel by up to 50x, if your algorithms occasionally produce denormal values. The downside of this speed up is a small loss of precision.

ThreadWaitBeforeSleep is set to 0 [ms] by default. It affects all threads of all threading subsystems (FFT, BLAS, VML and IPP). Sometimes it is desirable to further decrease thread context switch time at the expense of CPU usage. By setting this value to greater than zero, the threads will not enter sleep once completing a job, but will actively wait for the next job consuming 100% of the CPU until timeout occurs. This makes it possible to speed up short jobs, which could otherwise not be threaded. The downside is a higher CPU usage than otherwise spent mostly on thread waiting and use of system resources which cannot be spent by other applications.

When executing multiple threads on CPU's which contain multiple multi-core CPU's, like for example Intel Q6600, it becomes important to group threads working on the same data on the same pair of cores. By default MtxVec sets the thread affinity to true internally, to ensure that. On true quad or more core CPU's this setting is obsolete.

12 Debugging MtxVec

12.1 Debugger Visualizer

A set of extensions have been added to the Delphi debugger with the release of MtxVec v3.5. The extensions are called MtxVec debugger visualizer and are installed in to the Delphi IDE as a package. This package adds two new menu items to the Run menu: View Values and Draw values. The corresponding shortcuts are CTRL+F6 and CTRL+ALT+F6. By positioning the cursor on the variable while debugging you can obtain the view of the variable either formatted in a table or drawn on the chart. To cancel the displayed windows, press the Escape key. The debugger visualizer will work for the following types:

- sVector/cVector and TVec
- sMatrix/cMatrix and TMtx
- 1D static arrays of integers, smallint, bytes, singles, doubles and TCplx.
- 1D dynamic arrays of integers, smallint, bytes, singles, doubles and TCplx.
- TSignal type from DSP Master.

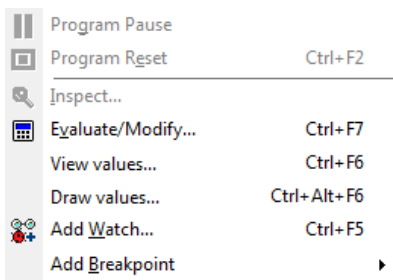


Figure 3 Two new commands in the Run menu.

The expressions passed to the visualizer will include the current word and any dot separated names to the left of it. By holding down Shift you also get a chance to modify the expression. If the expression is not recognized, the windows will not be displayed. Example:

```
tmp = Sqr(x);
```

If tmp or x are supported types, position the cursor just before, after or inside the variable name.

```
Test1.test2.tmp = Sqr(test3.x);
```

Same here, except that entire dot separated expression will be passed to the evaluator. This however will not work:

```
(Test1.test2).tmp
```

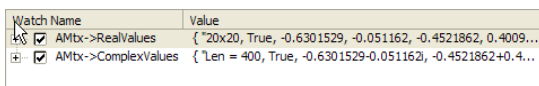
and requires manual intervention.

12.2 Viewing the values of Vector and Matrix in the debugger as an array

The following watches can be specified for variables of Vector or Matrix type:

- aVector->RealValues
- aVector->ComplexValues
- aMatrix->RealValues
- aMatrix->ComplexValues

Picture below shows how the watches look:



The `RealValues` and `ComplexValues` are functions with the sole purpose to return string in the watches. 20x20 shows the matrix size and True means that it is Complex. `RealValues` and `ComplexValues` return only the first 1000 elements converted to string, but usually that is sufficient. The debugger namely does not offer an option to examine the contents of `Vector` and `Matrix` objects quickly. When the provided debugger watches are no longer sufficient, use the `MtxVecTee.DrawIt` or `MtxVecEdit.ViewValues` methods.

12.3 Printing current values of variables

The most common way to debug an algorithm in the “old days” was to print the values of the variables to the screen or to the file. Today it is more common to use watches and tooltips to examine the values of variables. In some cases these two approaches do not work because of multithreading. Sometimes it is also desirable to have an algorithm return report on its convergence, like it is the case with optimization algorithms. For cases like this there is a global variable called `Report` declared in `MtxVec.pas` unit. `Report` is of a `TMtxVecReport` type and is derived from `TStringStream`. It has been extended with several new methods to support:

1. Saving the stream to `TStream` or a file.
2. Write the contents of `TVec` and `TMtx` objects to the stream. (as text)
3. Specify the text formatting of the floating point values.

Typically the `Report` object can be used like this:

```
sVector a,b;
a = Ramp(5); // a = [0,1,2,3,4]
b = Ramp(5,5,1); //b = [5,6,7,8,9]

Report->Print(OPENARRAY(TMtxVec*,(a,b)),
             OPENARRAY(AnsiString,("Vector a","Vector b")));
Report->NewLine();
Report->SaveToFile("C:\\test.txt");
Report->Clear();
```

The last parameter in the print command defines the variable name. Vectors and matrices can be mixed within the same `Print` method call. Other useful methods declared next to those already defined in `TStringStream` are:

- `report->PrintVec`
- `report->PrintMtx`
- `report->PrintSample`
- `report->PrintCplx`
- `report->PrintSampleArray`
- `report->PrintCplxArray`

12.4 Displaying the contents of the `TVec` and `TMtx`

12.4.1 As a delimited text

`TVec` and `TMtx` have two methods for getting and setting their values to and from a text file. They are called `StringsToValues` and `ValuesToStrings` and feature:

1. handling of complex and real values.
2. accept NAN and INF values
3. can get/set only a sub vector or a sub matrix.
4. Control the displayed precision

12.4.2 Within a grid.

The best way to display the contents of a matrix is within a grid. TVec and TMtx feature methods declared in MtxDialogs.pas that support getting and setting the contents of the TStringGrid object. These methods are: GridToValues and ValuesToGrid. They feature:

1. handling complex and real values
2. accept NAN and INF values
3. can display column/row headers or not
4. can get/set only a sub vector or a sub matrix.
5. Control the displayed precision

These methods are also used by the ViewValues method, which is declared in MtxVecEdit.pas.

12.5 Charting and drawing

MtxVec provides two units: MtxVecEdit and MtxVecTee to support the display and charting of the contents of TVec and TMtx. By calling the ViewValues routine a simple editor will be displayed allowing the user to examine the values of a TVec or TMtx object. See Figure 4. This editor can be displayed modally or not. If it is displayed modally, the values can be changed and the contents of the object will also change. If the editor is not displayed modally, the changes will be discarded. If the changes are not to be saved, then the user can freely select the number formatting. If the changes are to be saved, the number formatting must be full precision or otherwise the values will be truncated to the displayed precision. The values can not be edited unless Editable flag from the Options menu is checked. From the editor Chart menu "Series in rows" or "Series in cols" can be selected to draw the displayed values as a chart. Vector or matrix values can also be drawn directly on the chart by calling the DrawIt routine. This routine is located in the MtxVecTee unit. MtxVecTee routine contains a large set of DrawValues routines. This routines copy data from TVec or TMtx to the defined TChartSeries. Adding of new values is optimized for the TeeChart version used and charting can be considerably faster, if DrawValues is used. DrawValues routines also take care of any NAN's and INF's.

```
sVector a;

a->LoadFromFile("c:\\test.vec");
ViewValues(a); //display a window showing values in "a"
DrawIt(a); //display a chart of values in "a"
...
```

20x20	0-Re	0-Im	1-Re	1-Im	2-Re
0	0.4260	-0.8290	0.1630	0.2030	1.4050
1	1.2040	0.5370	-0.8320	-0.7470	1.9240
2	0.7360	0.5290	1.1640	1.2290	1.3710
3	1.1090	0.4930	0.2380	0.3140	1.3720
4	1.5070	-0.3620	0.4990	-0.0810	0.0880
5	-0.5300	0.8290	0.0600	1.7950	0.6250
6	-0.8650	1.1710	0.9170	-0.5980	-0.7720
7	-0.1710	-0.7680	0.0460	0.9950	-0.2570
8	1.6820	0.2630	-0.3120	0.1320	-0.0260

Figure 4

In the top left corner on Figure 4 is the size of the matrix (in this case 20x20). In the left most column are rows indexed starting with 0. The top row shows column labels. "0-Re" means that this is the first column of a complex matrix and shows the Real part of the complex number, "0-Im" column shows the imaginary component of the complex number stored in the first column of the matrix. On Figure 5 the

magnitudes of the values stored in the complex matrix can be seen. The layout of the values is the same as in the matrix editor (the left axis labels should be inverted).

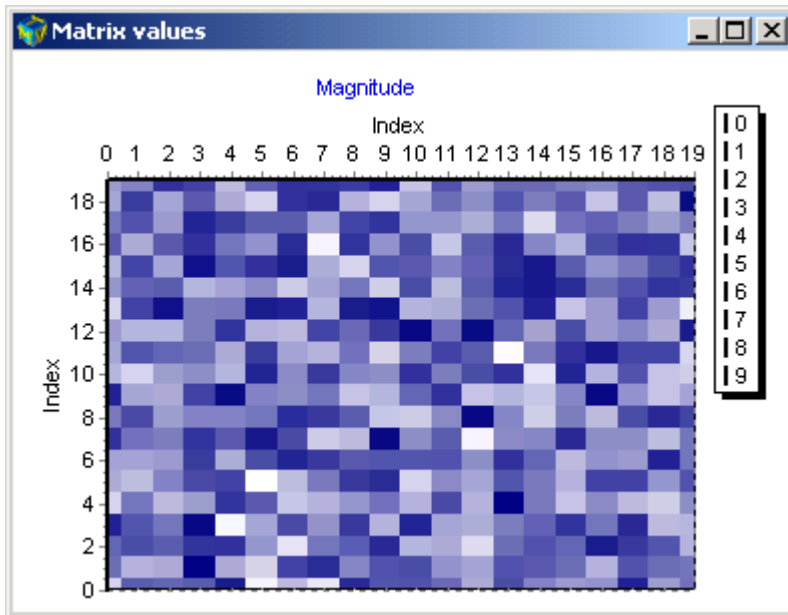


Figure 5

12.6 Memory leaks

When using Vector and Matrix classes, all the memory is managed automatically for the programmer giving him a free ride.

When using TVec and TMtx, the programmer should make sure to always match the Create and Free methods of objects and CreateIt/Freelt pairs, as already emphasized. MtxVec unit holds two global variables: Controller.MtxCacheUsed and Controller.VecCacheUsed. Their value will show the number of unfreed objects in the object pool. After the application has finished using MtxVec routines, these two variables should have a value of 0. This means that all objects for which CreateIt was called, were also passed to the Freelt routine.

All TVec and TMtx objects instances are also counted. If the application exits without first freeing all objects TVec/TMtx objects, an exception will be raised and a message dialog will be displayed.

12.7 Memory overwrites

When using TVec and TMtx or Vector and Matrix memory overwrites should be a thing of the past. But if you chose to work directly with memory here are some hints. Memory overwrite errors may not pop up immediately. The reason for this is that TVec and TMtx objects residing in the object cache have preallocated a specified number of elements. Such pre-allocation speeds memory allocation for small vectors and matrices considerably and also makes reallocations faster. MtxVec explicitly checks all parameters passed to TVec and TMtx routines for range-check errors.

The memory preallocation is disabled by calling:

```
Controller->SetVecCacheSize(0, 0);
Controller->SetMtxCacheSize(0, 0);
```

The first parameter defines the number of TVec/TMtx objects created in advance and the second parameter defines the number of array elements for which to preallocate the memory. By setting

memory preallocation to zero Access Violation Exceptions will be raised much closer to the actual cause of the problem.

13 Getting ready to deploy

Once the app has been debugged and is ready to be deployed, files required by MtxVec have to be included in the distribution package. These files are located in the windows\system or Windows\System32\ on 32bit OS. On 64bit OS the 32bit dlls are located in windows\SysWOW64\ and the 64bit versions of dlls are located in the Windows\system32 directory:

- MtxVec.lapack4d.dll. This library is mandatory and contains lapack functions.
- Libiomp5md.dll. This library is mandatory and contains OPENMP runtime used by other multithreaded dlls.

13.1 Compact MtxVec

Compact MtxVec is an initiative to allow the customer to provide additional processing muscles only where needed, and at the same time keep the distribution size as low as possible. Which dlls are linked in can be most easily specified from the MtxVec recompile tool. Alternatively this can also be controlled with the defines in bdsppdefs.inc:

- MtxVec.spl4d.dll. The library is optional and contains many vectorized and threaded functions from Intel IPP (excluding sin, cos, exp...). It is linked in when IPPDLL is defined in the bdsppdefs.inc. When not linked in, the functions declared in ippspl_core.pas will be called instead.
- MtxVec.vml4d.dll. This library is optional and contains many vectorized (but not threaded) math functions from Intel IPP (sin, cos, exp, ..) It is linked in when VMLDLL and IPPVML are defined in bdsppdefs.inc. When VMLDLL is not defined the Delphi based Math387 functions written in assembler will be called instead.
- MtxVec.Vmld.dll. This library is optional and contains vectorized and threaded math functions from Intel MKL (sin, cos, exp, ...). It is linked in when VMLDLL is defined and IPPVML is not defined in bdsppdefs.inc. When VMLDLL is not defined the Delphi based Math387 functions written in assembler will be called instead.
- MtxVec.sparse4d.dll. If the application uses sparse.pas, this library is required.
- MtxVec.Random.dll is used by the random generators located in RndGenerators.pas unit. The random generators are threaded and vectorized. It can be left out by commenting out {\$DEFINE RNDDLL} in bdsppdefs.inc, or by not linking against the unit. The fall back code is shipped with Dew Stats Master only.
- MtxVec.FFT.dll. This library contains multithreaded 1D, 2D and 3D DFT and FFT functions. It can be left out, by commenting out {\$DEFINE FFTDLL} define in bdsppdefs.inc. When not linked in, the fallback code declared in Lapack_dfti_core will be used instead.

If distribution size is a problem, we can make a build of custom size dll's for your specific application. The provided dll's have specialized code of each function for a general purpose Pentium compatible CPU, and separate code paths for CPU's with SSE2, SSE3 and SSE4 instruction sets. The appropriate code version is selected automatically, when the libraries are loaded

14 64bit version of MtxVec

This feature is currently available only to users of .NET. It is possible to run Delphi.NET application as either 32bit or 64bit, but it is not possible to debug 64bit applications with the Delphi IDE. The debugging must be done in 32bit, before the application is built as a 64bit app. To run MtxVec from a 64bit application, you need to copy 64bit versions of the required dll's to System32 dir on a 64bit OS. 32bit dll's are located inside SysWOW64 dir and have the same name as their 64bit counterparts.

The 64bit mode used uses 32bit integers and 64bit pointers. This means that you can allocate at most 16GB large arrays for double precision and 8GB large arrays for single precision.

15 Use up to 4GB of memory for 32bit application

MtxVec is large pointer aware and supports 32bit applications, which can address up to 4GB of memory on 64bit OS and up to 3GB on 32bit OS. To enable this feature for your application manually paste:

```
{ $SetPEFlags IMAGE_FILE_LARGE_ADDRESS_AWARE }
```

somewhere into the .dpr file source. This will function on all projects which use FastMM as the memory manager (Delphi 2006 and later). According to Pierre Le Riche, the author of the FastMM memory manager:

"I've been using a 4GB address space in all my applications since Delphi 2006 and have experienced no issues with the compiler or RTL. I also use a plethora of 3rd party components and have not experienced any problems with them either."

Due to address space fragmentation there are no contiguous blocks greater than 2GB with a 4GB address space. Largest single block is therefore limited to 2GB. Even if the compiler would allow it, it wouldn't be able to actually allocate it. On 64bit Windows 7 OS, it is possible for MtxVec to allocate a maximum of 3.5GB of memory. For 32-bit Windows XP/2003 it is necessary to specify the /3GB option in the boot.ini file.

16 Major function groups

The following function groups do not contain all the functions, but they do allow a faster navigation when searching for most common routines when writing custom functions. The “features” column in the tables can contain the following keywords:

SSE2/SSE3 – supports SSE2/SSE3/SSE4 and AVX instruction sets

SMP – symmetric multiprocessing (support for multiple CPU’s)

RCX – allows mixing real and complex numbers in the same expression even for indexed versions.

All functions also accept complex data where applicable. The math expression column is useful when writing a custom function and some expressions can be grouped for faster execution.

16.1 Basic vector math

Function	Class	Math expression	Features
Abs	TMtxVec	$a[i] = a[i] $ $a[i] = b[i] $	SSE2/SSE3
Add	TMtxVec	$a[i] = a[i] + B$	SSE2/SSE3, RCX
Add	TDenseMtxVec	$a[i] = b[i] + c[i]$ $a[i] = a[i] + S*c[i]$	SSE2/SSE3, RCX
AddProduct	TDenseMtxVec	$a[i] = a[i] + b[i]*c[i]$	SSE2/SSE3
ArcCos	TMtxVec	$a[i] = \text{ArcCos}(a[i])$ $a[i] = \text{ArcCos}(b[i])$	SSE2/SSE3, SMP
ArcCosh	TMtxVec	$a[i] = \text{ArcCosh}(a[i])$ $a[i] = \text{ArcCosh}(b[i])$	SSE2/SSE3, SMP
ArcCot	TMtxVec	$a[i] = \text{ArcCot}(a[i])$ $a[i] = \text{ArcCot}(b[i])$	SSE2/SSE3, SMP
ArcCoth	TMtxVec	$a[i] = \text{ArcCoth}(a[i])$ $a[i] = \text{ArcCoth}(b[i])$	SSE2/SSE3, SMP
ArcCsc	TMtxVec	$a[i] = \text{ArcCsc}(a[i])$ $a[i] = \text{ArcCsc}(b[i])$	SSE2/SSE3, SMP
ArcCsch	TMtxVec	$a[i] = \text{ArcCsch}(a[i])$ $a[i] = \text{ArcCsch}(b[i])$	SSE2/SSE3, SMP
ArcSec	TMtxVec	$a[i] = \text{ArcSec}(a[i])$ $a[i] = \text{ArcSec}(b[i])$	SSE2/SSE3, SMP
ArcSech	TMtxVec	$a[i] = \text{ArcSech}(a[i])$ $a[i] = \text{ArcSech}(b[i])$	SSE2/SSE3, SMP
ArcSin	TMtxVec	$a[i] = \text{ArcSin}(a[i])$ $a[i] = \text{ArcSin}(b[i])$	SSE2/SSE3, SMP
ArcSinh	TMtxVec	$a[i] = \text{ArcSinh}(a[i])$ $a[i] = \text{ArcSinh}(b[i])$	SSE2/SSE3, SMP
ArcTan	TMtxVec	$a[i] = \text{ArcTan}(a[i])$ $a[i] = \text{ArcTan}(b[i])$	SSE2/SSE3, SMP
ArcTan2	TMtxVec	$a[i] = \text{ArcTan2}(b[i],c[i])$	SSE2/SSE3, SMP
ArcTanh	TMtxVec	$a[i] = \text{ArcTanh}(a[i])$ $a[i] = \text{ArcTanh}(b[i])$	SSE2/SSE3, SMP
Cbrt	TMtxVec	$a[i] = (a[i])^{1/3}$ $a[i] = (b[i])^{1/3}$	SSE2/SSE3, SMP
Copy	TMtxVec	$a[i] = b[i]$	SSE2/SSE3
Ceil	TmtxVec	$a[i] = \text{Cos}(a[i])$ $a[i] = \text{Cos}(b[i])$	SSE2/SSE3, SMP
Cos	TMtxVec	$a[i] = \text{Cos}(a[i])$ $a[i] = \text{Cos}(b[i])$	SSE2/SSE3, SMP
Cosh	TMtxVec	$a[i] = \text{Cosh}(a[i])$ $a[i] = \text{Cosh}(b[i])$	SSE2/SSE3, SMP
Cot	TMtxVec	$a[i] = \text{Cot}(a[i])$	SSE2/SSE3, SMP

		$a[i] = \text{Cot}(b[i])$	
Coth	TMtxVec	$a[i] = \text{Coth}(a[i])$ $a[i] = \text{Coth}(b[i])$	SSE2/SSE3, SMP
Csc	TMtxVec	$a[i] = \text{Csc}(a[i])$ $a[i] = \text{Csc}(b[i])$	SSE2/SSE3, SMP
Csch	TMtxVec	$a[i] = \text{Csch}(a[i])$ $a[i] = \text{Csch}(b[i])$	SSE2/SSE3, SMP
CumSum	TDenseMtxVec	$a[i] = a[i] + A[i-1]$ $a[i] = b[i] + A[i-1]$	
Difference	TDenseMtxVec	$a[i] = A[i+1] - a[i]$ $a[i] = B[i+1] - b[i]$	
Divide	TMtxVec	$a[i] = a[i]/b[i]$ $a[i] = b[i]/c[i]$	SSE2/SSE3, RCX
DotProd	TDenseMtxVec	$S = \text{Sum}(a[i]*b[i])$	SSE2/SSE3
Exp	TMtxVec	$a[i] = \text{Exp}(a[i])$ $a[i] = \text{Exp}(b[i])$	SSE2/SSE3, SMP
Exp10	TMtxVec	$a[i] = \text{Exp10}(a[i])$ $a[i] = \text{Exp10}(b[i])$	SSE2/SSE3, SMP
Exp2	TMtxVec	$a[i] = \text{Exp2}(a[i])$ $a[i] = \text{Exp2}(b[i])$	SSE2/SSE3, SMP
Frac	TMtxVec	$a[i] = \text{fractional part of } a[i]$ $a[i] = \text{fractional part of } b[i]$	SSE2/SSE3, SMP
IntPower	TMtxVec	$a[i] = (a[i])^i$ $a[i] = (b[i])^i$	SSE2/SSE3
Inv	TMtxVec	$a[i] = 1/a[i]$ $a[i] = 1/b[i]$	SSE2/SSE3, SMP
InvCbrt	TMtxVec	$a[i] = (a[i])^{-1/3}$ $a[i] = (b[i])^{-1/3}$	SSE2/SSE3, SMP
InvSqrt	TMtxVec	$a[i] = (a[i])^{-1/2}$ $a[i] = (b[i])^{-1/2}$	SSE2/SSE3, SMP
IsEqual	TMtxVec	$a[i] = ? = b[i]$	
Ln	TMtxVec	$a[i] = \text{Ln}(a[i])$ $a[i] = \text{Ln}(b[i])$	SSE2/SSE3, SMP
Log10	TMtxVec	$a[i] = \text{Log10}(a[i])$ $a[i] = \text{Log10}(b[i])$	SSE2/SSE3, SMP
Log2	TMtxVec	$a[i] = \text{Log2}(a[i])$ $a[i] = \text{Log2}(b[i])$	SSE2/SSE3, SMP
LogN	TMtxVec	$a[i] = \text{LogN}(a[i])$ $a[i] = \text{LogN}(b[i])$	SSE2/SSE3, SMP
Max	TMtxVec	$S = \text{Max}(a[i])$	SSE2/SSE3
MaxMin	TMtxVec	$S1 = \text{Max}(a[i]), S2 = \text{Min}(a[i])$	SSE2/SSE3
Mean		$S = 1/\text{Len} * \text{Sum}(a[i])$	SSE2/SSE3
Min		$S = \text{Max}(a[i])$	SSE2/SSE3
Mul	TMtxVec	$a[i] = a[i] * B$	SSE2/SSE3, RCX
Mul	TDenseMtxVec	$a[i] = a[i] * b[i]$ $a[i] = b[i] * c[i]$	SSE2/SSE3, RCX
Normalize	TMtxVec	$a[i] = (a[i] - B)/C$	SSE2/SSE3
Product	TMtxVec	$S = \text{Product}(a[i])$	SSE2/SSE3
Power	TMtxVec	$a[i] = (a[i])^B$ $a[i] = (b[i])^C$ $a[i] = (B)^{c[i]}$ $a[i] = (b[i])^{c[i]}$	SSE2/SSE3, RCX, SMP
Round	TMtxVec	$a[i] = \text{nearest integer to } a[i]$ $a[i] = \text{nearest integer to } b[i]$	SSE2/SSE3, SMP
Sec	TMtxVec	$a[i] = \text{Sec}(a[i])$ $a[i] = \text{Sec}(b[i])$	SSE2/SSE3, SMP
Sech	TMtxVec	$a[i] = \text{Sech}(a[i])$ $a[i] = \text{Sech}(b[i])$	SSE2/SSE3, SMP
SetVal	TMtxVec	$a[i] = B$	SSE2/SSE3

SetZero	TMtxVec	$a[i] = 0$	SSE2/SSE3
Sgn	TMtxVec	$a[i] = \text{signum}(a[i])$	
Sign	TMtxVec	$a[i] = -a[i]$	SSE2/SSE3
Sin	TMtxVec	$a[i] = \text{Sin}(a[i])$ $a[i] = \text{Sin}(b[i])$	SSE2/SSE3, SMP
SinCos	TMtxVec	$b[i] = \text{Sin}(a[i]), c[i] = \text{Cos}(a[i])$	SSE2/SSE3, SMP
Sinh	TMtxVec	$a[i] = \text{Sech}(a[i])$ $a[i] = \text{Sech}(b[i])$	SSE2/SSE3, SMP
SinhCosh	TMtxVec	$b[i] = \text{Sech}(a[i]), c[i] = \text{Sech}(a[i])$	SSE2/SSE3, SMP
Sqr	TMtxVec	$a[i] = (a[i])^2$ $a[i] = (b[i])^2$	SSE2/SSE3, SMP
Sqrt	TMtxVec	$a[i] = (a[i])^{1/2}$ $a[i] = (b[i])^{1/2}$	SSE2/SSE3, SMP
Sub	TMtxVec	$a[i] = a[i] - B$	SSE2/SSE3, RCX
Sub	TDenseMtxVec	$a[i] = a[i] - b[i]$ $a[i] = b[i] - c[i]$	SSE2/SSE3, RCX
SubFrom	TDenseMtxVec	$a[i] = B - a[i]$	SSE2/SSE3, RCX
Sum	TMtxVec	$S = \text{Sum}(a[i])$	SSE2/SSE3
Tan	TMtxVec	$a[i] = \text{Tan}(a[i])$ $a[i] = \text{Tan}(b[i])$	SSE2/SSE3, SMP
Tanh	TMtxVec	$a[i] = \text{Tanh}(a[i])$ $a[i] = \text{Tanh}(b[i])$	SSE2/SSE3, SMP
Trunc	TMtxVec	$a[i] = \text{integer part of } a[i]$ rounded to zero $a[i] = \text{integer part of } b[i]$ rounded to zero	SSE2/SSE3, SMP
ThreshBottom ThreshTop	TMtxVec	Limit the upper or lower value range	SSE2/SSE3

16.2 Statistical

Kurtosis	TDenseMtxVec	$a[i] = \text{Kurtosis}(a[i])$ $a[i] = \text{Kurtosis}(b[i])$	SSE2/SSE3
Skewness	TDenseMtxVec	$a[i] = \text{Skewness}(a[i])$ $a[i] = \text{Skewness}(b[i])$	SSE2/SSE3
StdDev	TDenseMtxVec	$a[i] = \text{StdDev}(a[i])$ $a[i] = \text{StdDev}(b[i])$	SSE2/SSE3
RMS	TDenseMtxVec	$a[i] = \text{RMS}(a[i])$ $a[i] = \text{RMS}(b[i])$	SSE2/SSE3
Mean	TDenseMtxVec	$S = \text{Mean}(a[i])$	SSE2/SSE3

16.3 Complex number specific

Expj	TMtxVec	$a[i] = \exp(-j*w*b[i])$	SSE2/SSE3, SMP
Flip	TMtxVec	$a[i].\text{Re} = b[i].\text{Im}, a[i].\text{Im} = b[i].\text{Re}$	SSE2/SSE3
CartToPolar/PolarToCart	TMtxVec	$a[i] = \text{CartToPolar}(b[i], c[i])$ $a[i] = \text{PolarToCart}(b[i], c[i])$	SSE2/SSE3
CplxToReal/RealToCplx	TMtxVec	$a[i] = \text{CplxToReal}(b[i], c[i])$ $a[i] = \text{RealToCplx}(b[i], c[i])$	SSE2/SSE3
ExtendToComplex	TMtxVec	$a[i].\text{Re} = b[i]; a[i].\text{Im} = a[i]$ $a[i].\text{Im} = 0$	SSE2/SSE3
ImagPart	TMtxVec	$a[i] = b[i].\text{Im}$	SSE2/SSE3
RealPart	TMtxVec	$a[i] = b[i].\text{Re}$	SSE2/SSE3
Mull	TMtxVec	$a[i] = a[i]*i$ $a[i] = b[i]*i$	SSE2/SSE3

ConjMul	TMtxVec	$a[i] = a[i]*Conj(b[i])$ $a[i] = b[i]*Conj(c[i])$	SSE2/SSE3
Conj	TMtxVec	$a[i] = a[i].Re - a[i].Im$ $a[i] = b[i].Re - b[i].Im$	SSE2/SSE3
PhaseSpectrum	TMtxVec	$a[i] = Arctan2(b[i].Im/b[i].Re)$	SSE2/SSE3
PowerSpectrum	TMtxVec	$a[i] = (sqr(b[i].Re) + sqr(b[i].Im))$	SSE2/SSE3
FlipConj	TMtxVec	$a[i].Re = b[i].Im$ $a[i].Im = -b[i].Re$	SSE2/SSE3

16.4 Size, streaming and storage

CopyBinaryFromArray, CopyFromArray, LoadFromFile, CopyToArray, LoadFromStream, ReadHeader, ReadValues, SaveToFile, SaveToStream, SetCplx, SetDouble, SetInteger, SetInt, SetSingle, SizeToArray, WriteHeader, WriteValues, Size, Resize

16.5 FFT's

Function	Class	Math expression	Features
FFT/IFFT	TDenseMtxVec TVec	$A = FFT(A), A = IFFT(A)$ $A = FFT(B), A = IFFT(B)$	SSE2/SSE3, SMP
FFTFromReal	TDenseMtxVec TVec	$A = FFT(A)$ $A = FFT(B)$	SSE2/SSE3, SMP
IFFTToReal	TDenseMtxVec TVec	$A = IFFT(A)$ $A = IFFT(B)$	SSE2/SSE3, SMP
FFT1D/IFFT1D	TMtx	$A(i) = FFT(A(i)), A(i) = IFFT(A(i))$ $A(i) = FFT(B(i)), A(i) = IFFT(B(i))$	SSE2/SSE3, SMP
FFT2D	TMtx	$A = FFT2D(A)$ $A = FFT2D(B)$	SSE2/SSE3, SMP
FFT2DFromReal	TMtx	$A = FFT2D(A)$ $A = FFT2D(B)$	SSE2/SSE3, SMP
FFT1DFromReal	TMtx	$A(i) = FFT(A(i))$ $A(i) = FFT(B(i))$	SSE2/SSE3, SMP
IFFT1DToReal	TMtx	$A(i) = IFFT(A(i))$ $A(i) = IFFT(B(i))$	SSE2/SSE3, SMP
DCT	TVec	$A = DCT(B)$	SSE2/SSE3
IDCT	TVec	$A = IDCT(B)$	SSE2/SSE3

16.6 Linear algebra

Function	Class	Math expression	Features
LUSolve	TMtx	$A*X = B$, solves for X	SSE2/SSE3, SMP
LQRSolve	TMtx	Least squares solution	SSE2/SSE3, SMP
SVDSolve	TMtx	Singular value solution	SSE2/SSE3, SMP
Eig	TMtx	Eigvalues and eigen vectors	SSE2/SSE3, SMP
EigGen	TMtx	Generalized eigen values	SSE2/SSE3, SMP
Transp	TMtx	Transpose	SSE2/SSE3
Adjung	TMtx	Adjungate	SSE2/SSE3
Cholesky	TMtx	Cholesky factorization	SSE2/SSE3, SMP
Determinant	TMtx	Determinant	SSE2/SSE3, SMP
Inverse	TMtx	A^{-1}	SSE2/SSE3, SMP
LU	TMtx	LU factorization	SSE2/SSE3, SMP
LQR	TMtx	LQ and QR factorization	SSE2/SSE3, SMP
SVD	TMtx	SVD decomposition	SSE2/SSE3, SMP
Mul	TMtx	Matrix multiply	SSE2/SSE3, SMP
MtxFunction	TMtx	Matrix function: $A = Fun(B)$	SSE2/SSE3, SMP
MtxSqrt	TMtx	$A^{-0.5}$	SSE2/SSE3, SMP

MtxPower	TMtx	A^p	SSE2/SSE3, SMP
MtxIntPower	TMtx	A^i	SSE2/SSE3
TensorProd	TMtx	$aMtx = \text{alfa} * bMtx * Vec$ $aMtx = \text{alfa} * Vec * Mtx$ $aMtx = Vec1 \times Vec2$	SSE2/SSE3
AddTensorProd	TMtx	$Mtx = \text{alfa} * Vec1 \times Vec2 + Mtx.$	SSE2/SSE3
Sylvester	TMtx	The Sylvester equation	SSE2/SSE3

16.7 Matrix conversions

Function	Class	Math expression	Features
BandedToDense	TMtx	Banded matrix to dense	
DenseToBanded	TMtx	Dense matrix to banded	

16.8 Miscellaneous matrix routines

Function	Class	Math expression	Features
Diag	TMtx	Matrix diagonal	
Eye	TMtx	Matrix with 1's on main diagonal.	
Concat, ConcatHorz, ConcatVert	TMtx	Concatenate matrices	SSE2/SSE3
FlipVer, FlipHor	TMtx	Flip matrices	
Kron	TMtx	Kronecker product	SSE2/SSE3
LowerTriangle, UpperTriangle	TMtx		SSE2/SSE3
Norm1, NormFro, NormInf	TMtx	Matrix norms	SSE2/SSE3
Pascl, VanderMond, Toeplitz	TMtx	Special matrices	
Rotate90	TMtx	Rotate the matrix	SSE2/SSE3
SetCol, SetRow	TMtx	Copies row/column	SSE2/SSE3
SumCols, SumRows	TMtx	Sums columns or rows	SSE2/SSE3
MeanCols, MeanRows	TMtx	Average of columns or rows	SSE2/SSE3

17 Making use of OpenCL

OpenCL syntax follows the same rules as present in MtxExpr.hpp. Instead of including MtxExpr.hpp, include the clMtxExpr.hpp in to your cpp file. Instead of declaring sVector and sMatrix declare oclVector and oclMatrix types. For more information about Open CL read the "Cougar Open CL.pdf" manual included with MtxVec. Before oclVector and oclMatrix can be used, the clPlatform global variable needs to be configured to specify on which device(s) should the code run.

18 Compatibility breaking changes from version 1.x, 2.x

- TVec.Add(a,2); replaced with AddScaled, old signature has new meaning
- TMtx.TensorProd(a,b,True); replaced with TMtx.AddTensorProd(a,b), no new meaning for old signature
- FFT methods have been completely redesigned for version 2. Some functions have been removed and others with new names have been added.

19 Compatibility changes from version 4.0 to 4.1

The following syntax is still supported:

```
//1. add MtxVecCPP.cpp to the project
//2. Include MtxVecCPP.h in the working cpp file

#include "MtxVecCPP.h"

//3. Insert TMtxFunctionEvaluator in to the main form.
// This will trigger the IDE to add required libs to the project linker settings.
// Build and run. The inserted component can then be deleted.
//4. You are now ready to write the code:

{
    TCplx ac;
    Vector av;
    Matrix am,bm;
    TDoubleArray arr;
    int i;
    //
    ac = StrToCplx("1+2i"); //Convert from string to complex number

    am = RandGauss(5,5,true); //5x5 complex matrix with Gaussian noise
    av = Ramp(25); //real vector = [0, 1, 2,....., 23, 24]

    bm = Exp(am)*av + Cos(ac) + 2; // (*,/) treat Matrices as vectors
    // Same as: ./ and .* operators (not linear algebra)

    //To make linear algebra multiplication and division use functions

    bm = Divide(am,bm) + 2; //matrix division
    bm = Mul(am,bm) + 2; //matrix multiply

    //of course you can mix matrices and vectors

    av->Resize(5);
    av = Mul(av,bm) + 2; //vector from left and matrix multiply
    av->CopyToArray(arr); //copy data to array of double
    av->CopyFromArray(arr); //copy data from array to vector

    for (i = 0; i < am->Rows; i++) //standard loop example
    {
        //in case of complex values
        av->CValues[i] = am->CValues[i][0]*av->CValues[i];
    }
}
```

```
    av.CValues(i) = am.CValues(i,0)*av.CValues(i); //same as previous line

    //in case of real values
    av[i] = am[i,0]*av[i];
    av->Values[i] = am->Values[i][0]*av->Values[i];
    av.Values(i) = am.Values(i,0)*av.Values(i); //same as previous lines
}
}
```

Note however that syntax cannot be mixed within the same file between old and new syntax. As soon as MtxVecCPP.h is included the language features of version 4.1 are disabled and MtxExpr.hpp unit may not be included in to the same .cpp file. This allows you to maintain old code unchanged and write new code following the newer style.

Although the coding style remained largely unchanged, the main difference is the behavior of the assign operator for Vectors and Matrices. With MtxVecCPP.h it was automatically a deep copy if needed. With version 4.1, the deep copy needs to be specified explicitly:

```
a.Copy(b);
```

or:

```
a = b();
```

This only copies the pointer:

```
a = b;
```

,if b is not subranged. All assignments should be not be automatically changed to a deep copy. Sometimes that is not required.