

# **MtxVec v2.1**

**A programming guide to MtxVec**

<b>MtxVec v2.1</b> .....	2
<i>A programming guide to MtxVec</i> .....	2
<b>1 Introduction</b> .....	5
<b>1.1 Why MtxVec</b> .....	5
<b>1.2 Why MtxVec for .NET</b> .....	6
<b>1.3 How fast is MtxVec</b> .....	6
<b>1.4 Why written in Delphi and not C#</b> .....	6
1.4.1 Global functions .....	6
1.4.2 Operator overloading.....	6
<b>2 Organization</b> .....	7
<b>2.1 Compiler support</b> .....	7
<b>3 MtxVec programming interface</b> .....	8
<b>3.1 Namespaces</b> .....	8
<b>3.2 Object hierarchy</b> .....	8
<b>3.3 TMtxVec object</b> .....	8
3.3.1 Error checking.....	9
3.3.2 Getting and setting values .....	9
3.3.3 Making use of the abstract class.....	9
3.3.4 Writing abstract class code.....	11
3.3.5 Indexes, ranges and subranges .....	12
<b>3.4 TVec and TMtx methods as functions</b> .....	14
<b>3.5 Create and Free</b> .....	14
<b>3.6 Complex data</b> .....	15
<b>3.7 MtxVec types</b> .....	16
3.7.1 TSample .....	16
3.7.2 TCplx .....	16
3.7.3 TSampleArray, TCplxArray.....	17
<b>4 Programming style</b> .....	17
<b>4.1 Try-finally blocks</b> .....	17
<b>4.2 Raising exceptions</b> .....	18
4.2.1 Invalid parameter passed: .....	18
4.2.2 Reformat the exception .....	19
<b>4.3 By all means indent code</b> .....	19
<b>4.4 Do not create objects within procedures and return them as result:</b> .....	20
<b>4.5 Use CreateIt/FreeIt only for dynamically allocated objects whose lifetime is limited only to the procedure being executed</b> .....	20
<b>5 Accessing values of TVec and TMtx</b> .....	21
<b>5.1 Array property access</b> .....	21
<b>5.2 Direct dynamic array pointer</b> .....	21
<b>6 Vector/Matrix operations</b> .....	24
<b>6.1 Matrix, Matrix multiply: <math>C = A * B</math>;</b> .....	24
<b>6.2 Matrix, Vector multiply: <math>B = A * X</math></b> .....	24
<b>6.3 Vector, Matrix multiply: <math>B = X * A</math></b> .....	24
<b>6.4 Vector, Vector multiply: <math>A = B * X</math></b> .....	24

6.5	Sparse matrix, vector multiply: $B = S * X$ .....	24
6.6	Vector, sparse matrix multiply: $B = X*S$ .....	24
6.7	Matrix, sparse matrix multiply: $B = A*S$ .....	24
6.8	Sparse matrix, matrix multiply: $B = S*A$ .....	25
6.9	Other types of operations .....	25
7	<i>Memory management</i> .....	25
7.1	Introduction .....	25
7.2	In-place/not-in-place operations .....	26
8	<i>Range checking</i> .....	26
9	<i>Why and how NAN and INF</i> .....	26
10	<i>Serializing and streaming</i> .....	27
10.1	Streaming with TMtxComponent.....	27
10.2	Streaming of TVec, TMtx and TSparseMtx .....	28
10.3	Write TMtx to a text file.....	28
10.4	Read TMtx from a text file.....	29
11	<i>Input-output interface</i> .....	29
11.1	Reading and writing raw data .....	29
11.2	Formatting floating point values – FloatToString.....	30
11.3	Printing current values of variables .....	31
11.4	Displaying the contents of the TVec and TMtx .....	31
11.4.1	As a delimited text .....	31
11.4.2	Within a grid. ....	31
11.5	Charting and drawing .....	32
12	<i>Getting up to speed</i> .....	34
12.1	Floating point code vectorization.....	34
12.2	Block based processing .....	36
12.3	Common pitfalls .....	38
13	<i>Debugging MtxVec</i> .....	41
13.1	Memory leaks .....	41
13.1.1	Debugging under .NET .....	41
13.2	Memory overwrites .....	41
14	<i>Getting ready to deploy</i> .....	42
15	<i>Function groups</i> .....	43
15.1	Basic vector math.....	43
15.2	Statistical.....	45
15.3	Complex number specific .....	45
15.4	Size, streaming and storage.....	46
15.5	FFT's.....	46
15.6	Linear algebra .....	46
15.7	Matrix conversions.....	47
15.8	Miscellaneous matrix routines .....	47

<b>16</b>	<b><i>Delphi 2006 and operator overloading</i></b> .....	<b>48</b>
<b>16.1</b>	<b>Implicit type conversions</b> .....	<b>48</b>
<b>16.2</b>	<b>How to organize your code</b> .....	<b>49</b>
<b>17</b>	<b><i>C++ Builder support</i></b> .....	<b>50</b>
<b>17.1</b>	<b>Extra features</b> .....	<b>50</b>
17.1.1	Smart pointers .....	50
17.1.2	Vector, Matrix, CVector, CMatrix, SparseMatrix.....	50
17.1.3	Operator overloading.....	51
17.1.4	Working with elements and objects .....	52
17.1.5	Subranges .....	52
17.1.6	Methods and functions .....	53
17.1.7	Range Checking .....	53
<b>17.2</b>	<b>Building and debugging</b> .....	<b>53</b>
<b>17.3</b>	<b>Advanced Topics</b> .....	<b>54</b>
17.3.1	Preallocated objects.....	54
17.3.2	Without redundant copying.....	54
17.3.3	More about subranges .....	55
<b>17.4</b>	<b>Open array parameters and SetIt method</b> .....	<b>55</b>
<b>17.5</b>	<b>Loading and saving a matrix in a text file</b> .....	<b>56</b>
<b>18</b>	<b><i>Compatibility breaking changes</i></b> .....	<b>58</b>
<b>18.1</b>	<b>From version 1.x</b> .....	<b>58</b>
<b>18.2</b>	<b>From version 2.0</b> .....	<b>58</b>

## 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 P4 SSE2 and SSE3 instruction sets.
3. Improved compactness and readability of code.
4. Significantly shorter development times by protecting the developer from a wide range of possible errors.

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](http://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 P4 CPU's provided by Intel with their Math Kernel library. Future versions of MtxVec will also support Lapack libraries provided by AMD.

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 Pentium 4.

### 1.1 Why MtxVec

- 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 instruction set, the resulting code can never be as optimal as a hand optimized version.
- Many linear algebra routines are multithreaded, including FFT's and sparse matrix solvers.
- 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 Why MtxVec for .NET

- Optimizing the performance of .NET code can be a real challenge. MtxVec delivers patterns which allow fast executing floating point code to be also written fast.
- The advantage of P4 SSE2 and SSE3 instruction sets can be exploited only with unmanaged code. And only hand optimized code can be truly optimal.
- Provides an interface to LAPACK to be used from any .NET language.
- Unmanaged code is encapsulated in to efficient and thin wrappers making the code “safe” for the end user, with little or no overhead associated with transitions from managed to unmanaged code. Many difficult to trace bugs and performance issues are avoided in this way.
- Bypasses .NET garbage collector to manage large vectors and matrices more efficiently. The .NET garbage collector uses a special heap for objects greater than 80kBytes which can not be compacted. This can result in excessive memory usage especially when working with large vectors and matrices, or out-of-memory errors, when using the application for a certain period of time.
- Especially for complex numbers, the performance gain against native .NET code can be very big.

## 1.3 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. For .NET applications these factors should be multiplied by about 1.3.

## 1.4 Why written in Delphi and not C#

### 1.4.1 Global functions

Delphi.NET supports global functions. It is possible to write an expression like this:

```
a := log(10) + a + sin(c);
```

In C# the next syntax is a requirement

```
a = Math.log(10) + a + Math.sin(c);
```

The class name can not be omitted. Consequently, the math code is more readable in Delphi as in C# and thus makes Delphi an interesting alternative for scientific applications written in .NET.

### 1.4.2 Operator overloading

In Delphi 2006 and C++ it is possible to write vector expressions directly:

```
var a,b: Vector;  
    i: integer;  
begin  
    for i := 0 to 1000 do a := a + b;  
end;
```

where a and b are vectors (array objects) which do not require create and destroy, because they are reference counted value classes (records) whose memory is freed immediately on the exit from the procedure. With garbage collector this kind of loops result in excessive memory usage and multiple GC collects while the loop is running.

## 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, ndspl.pas files and implemented as external DLLs - MtxVec.Lapackd.dll, MtxVec.Spld.dll and bdspl.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.

**.NET specifics:**

Visual Studio .NET version of MtxVec contains both second and third level in the same Dew.Math.dll. The packages used by Delphi VCL.NET version of MtxVec, use assemblies named as DewRes.MtxVec.\*.dll.

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

### 2.1 Compiler support

MtxVec 2.1 supports the following compilers:

**Borland Delphi W32**

**Borland Delphi VCL.NET**

**Borland Delphi Winforms (.NET)**

**Borland C++ Builder**

**Visual Studio C# (VB.NET, C++)**

**Versions**

Versions 4 and 5 are supported with MtxVec 1.5. Newer versions of MtxVec support only 6,7,2005 and 2006.

Delphi 2005, Delphi 2006

Delphi 2005, Delphi 2006

Version 6, BDS 2006

2003.NET, 2005.NET

## 3 MtxVec programming interface

### 3.1 Namespaces

.NET framework introduced namespaces. In Delphi.NET or Delphi.W32 MtxVec does not use namespaces. However in C# all MtxVec classes and types are available within Dew.Math namespace. MtxVec also declares many global functions. Because C# does not recognize global functions, they are placed within Dew.Math.Units namespace as class static members of classes which have the type name equal to the unit name. To use MtxVec.NET assemblies from Visual Studio add Dew.Math.dll to the references of your project. Add Dew.Math.dll also to the Toolbox by selecting Tools->Add/Remove Toolbox items. Several components will be added to the currently selected Toolbox tab. You also need to add the following to the C# using clause:

```
using Dew.Math; // contains all classes defined by MtxVec
using Dew.Math.Units; // - contains all global functions accessible via classes named after unit names.
```

Some units had to be completely rewritten so that they can be used in Winforms applications. They are accessible via:

```
using Dew.Controls;
using Dew.Controls.Units;
```

This will give access to visual controls which can be used with Winforms: the charting support for the .NET version of the TeeChart component, number editor for real and complex numbers, interface to load and save data from MtxVec to a database and others.

### 3.2 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:

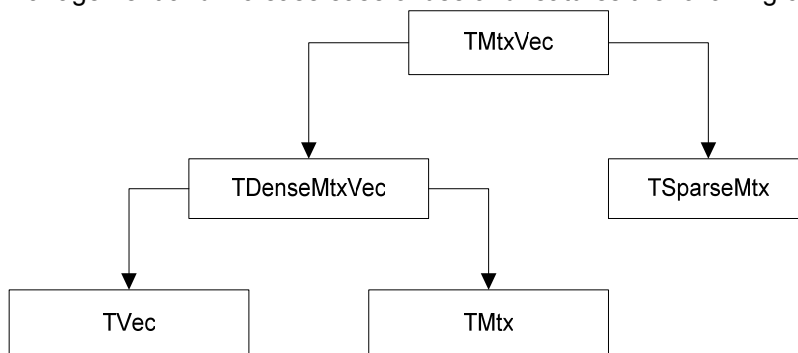


Figure 1 MtxVec class hierarchy

### 3.3 TMtxVec object

TMtxVec is the basic vector/matrix object. It handles memory allocation and basic vector arithmetic's. TMtxVec works with memory obtained directly from the operating system. The object bypasses the default Delphi memory manager and frees it from the load for which it is not very well suited. Memory allocation is cached via an "object cache" mechanism.

**.NET specifics:**

Under .NET TMtxVec allocates unmanaged memory. This removes the need to copy the memory when it is passed to unmanaged code. Since TMtxVec allocates unmanaged resource (memory) the garbage collector can not be efficient in automatically freeing the object once it is no longer needed. Therefore, all descendants from TMtxVec must be explicitly freed once they are no longer needed. If the object is not freed explicitly a memory leak will occur. The finalize pattern is not used because of side effects. MtxVec counts the number of created and destroyed objects and if there is a mismatch between created and destroyed object count it will notify the user when the application ends.

### 3.3.1 Error checking

All methods and properties of TMtxVec descendants are explicitly "range checked". Range checking ensures that the user can not read or write values past the size of the allocated memory. Once the code is compiled without **assertations**, range checking is disabled and higher performance can be achieved in some cases. Every effort has been made to prevent the user of the library to make an error that would result in memory overwrite. (Writing or reading to parts of the memory which were not allocated before and thus overwriting data of another part of the application.)

### 3.3.2 Getting and setting values

The object can hold either real or complex double precision data. TMtxVec classes have a list of methods which may be called like this:

```
vector_object_a.Add(vector_object_b);
```

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.

### 3.3.3 Making use of the abstract class

Many methods can accept any TMtxVec descendant class:

```
Delphi:  
TVec.Copy(Src: TMtxVec);
```

```
C#:  
TVec.Copy(TMtxVec Src);
```

This is one of the most powerful features of MtxVec. In this case, the source can be a 2D matrix, sparse matrix or also a vector:

- 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:

```
Delphi:
TMtx.Copy(Src: TMtxVec);
TSparseMtx.Copy(Src: TMtxVec);
```

```
C#:
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:

```
Delphi:
TMtxVec.Size(Src: TMtxVec, aComplex: boolean);
```

```
C#:
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

```
Delphi:
aTMtxVecObject.Values1D[1] := 1; //sets real value at index 1 to 1
```

```
C#:
aTMtxVecObject.Values1D[1] = 1; //sets real value at index 1 to 1
```

**TMtxVec.CValues1D** - array property to access complex values

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

```
C#:
aTMtxVecObject.CValues1D[1] = Cplx(1,0); //sets complex value at index 1 to 1
```

**TMtxVec.PValues1D** - function returns a pointer to real value

```
Delphi:
var aPointer: Pointer;
...
aPointer := aTMtxVec.PValues1D(1); //returns a pointer to value at index 1
```

C#:

```
IntPtr aPointer;
aPointer = aTMtxVec.PValues1D(1); //returns a pointer to value at index 1
```

**.NET specifics:**

Under .NET TMtxVec uses unmanaged memory. The PValuesXX method's return an IntPtr type pointer to unmanaged memory. This pointer can be incremented or passed to unmanaged code safely without the need to worry about the garbage collector, because unmanaged memory will not move during the life of the object.

TMtxVec.PCValues1D - function returns a pointer to complex value

Delphi:

```
var aCplxPointre: Pointer;
aCplxPointer := aTMtxVec.PCValues1D(1); //returns a pointer to complex
```

C#:

```
IntPtr aPointer;
aCplxPointer = aTMtxVec.PCValues1D(1); //returns a pointer to complex value at
                                         index 1
```

### 3.3.4 Writing abstract class code

This is best examined by an example:

Delphi:

```
procedure CustomExpj(Dst, SrcOmega: TMtxVec);
begin
    Dst.Size(SrcOmega, True);
    CustomExpjNoSize(Dst, SrcOmega);
end;
```

C#:

```
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 (IntPtr in .NET): PValues1D, PCValues1D, PValues1D,
- Getting/settings values: Values1D, CValues1D, IValues1D
- Setting size: Complex, Length, Size(Src: TMtxVec, IsComplex: boolean);
- all methods of TMtxVec class.

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

Delphi:

```

procedure CustomExpj(Dst, SrcOmega: TMtxVec);
var a: TVec;
begin
    CreateIt(a);
    try
        Dst.Size(SrcOmega, True);
        a.SetSubRange(Dst);
        a.Expj(SrcOmega); //call a TVec only method here
    finally
        FreeIt(a);
    end;
end;
    
```

C#:

```

private void CustomExpj(TMtxVec Dst, TMtxVec SrcOmega)
{
    TVec a;
    MtxVec.CreateIt(out a);
    try {
        Dst.Size(SrcOmega, true);
        a.SetSubRange(Dst);
        a.Expj(SrcOmega); //call a TVec only method here
    }
    finally {
        MtxVec.FreeIt(ref a);
    }
}
    
```

### 3.3.5 Indexes, ranges and subranges

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

Delphi:

```

function Exp: TMtxVec; overload;
function Exp(X: TMtxVec): TMtxVec; overload;
function Exp(Index, Len: integer): TMtxVec; overload;
function Exp(X: TMtxVec; XIndex, Index, Len: integer): TMtxVec; overload;
    
```

C#:

```

private TMtxVec Exp();
private TMtxVec Exp(TMtxVec X);
private TMtxVec Exp(int Index, int Len);
private TMtxVec Exp(TMtxVec X, int 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 3.3.3 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 is 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:

```
Delphi/C#:  
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:

```
Delphi/C#:  
  
b.SetSubRange(2, 10);  
a.Copy(b);
```

Which is the same as:

```
Delphi/C#:  
  
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:

```
Delphi/C#:  
  
b_vec.SetSubRange(aMatrix, 2, 10);  
a.Copy(b_vec);
```

This is the same as:

```
Delphi/C#:  
  
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:

```
Delphi/C#:
```

```
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:

```
Delphi:
a.IValues[0] := 1;
```

```
C#:
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:

```
Delphi/C#:
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:

```
Delphi:

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

```
C#:

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

### 3.4 TVec and TMtx methods as functions

Ideally a mathematical expression could be written like this:

```
a := a*b + b;
```

For vectors and matrices this can not be done. The closest syntax allowed is this:

```
a.Add(c.Mul(a,b),b); //where a,b,c are TVec objects
```

Almost every method of TVec and TMtx returns Self. This allows nesting of calls like in the example above. Syntax like this will result in a memory leak:

```
a := c.Mul(a,b); //don't do this
```

### 3.5 Create and Free

Before objects can be used, they have to be created. TVec and TMtx objects can be created and destroyed in the standard way:

Delphi:

```
a := TVec.Create;
b := TMtx.Create;
try
  ....
finally
  a.Free; //this is mandatory in .NET also!!!!
  b.Free;
end;
```

C#:

```
a = new TVec();
b = new TVec();
try
{
  ...
}
finally {
  a.Free();
  b.Free()
}
```

Or in a fast way, by using object cache:

Delphi:

```
CreateIt(a);
CreateIt(b);
try
  ....
finally
  FreeIt(a); //this is mandatory in .NET also!!!!
  FreeIt(b);
end;
```

C#:

```
TVec a,b;
MtxVec.CreateIt(out a, out b);
try
{
  ...
}
finally {
  MtxVec.FreeIt(ref a, ref 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.

### 3.6 Complex data

Both TVec and TMtx can hold real and complex data. Here is an example:

Delphi:

```
a.Length := 10;
a.Complex := True;
```

C#:

```
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:

Delphi:

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

C#:

```
a.Values[0] = 1;
a.CValues[0] = Math387.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 (TCplx = record Re,Im: double; end;).

### 3.7 MtxVec types

MtxVec declares many different types, but some should be mentioned explicitly:

#### 3.7.1 TSample

This type is used everywhere where it is necessary to declare a floating point number. It can be declared as double or single (in Math387).

Delphi:

```
{ $IFDEF TTDDOUBLE } type TSample = double; { $ENDIF }
{ $IFDEF TTSINGLE } 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. Type aliasing is not supported by the C# language.

#### 3.7.2 TCplx

Declared as:

Delphi:

```
TCplx = packed record
    Re, Im: TSample;
end;
```

C#:

```
public struct TCplx {
    public TSample Re, Im;
}
```

This is the default complex number type used by MtxVec. Many object oriented libraries declare the complex type as an object type (TComplex = class(TObject)). This is not very good, because the consecutive elements within the array are then no longer stored at consecutive memory locations. The consequence are extreme performance penalties. (up to 300x).

### 3.7.3 TSampleArray, TCplxArray

Declared as:

Delphi:  
 TSampleArray = **array of** TSample;  
 TCplxArray = **array of** TCplx;

C#:  
 TSampleArray and TCplxArray are not available as types in C#, because C# does not allow type aliasing. When there is a reference to TSample use double, when there is reference to TSampleArray use double[], and when there is a reference to TCplxArray use TCplx[] to declare the variable.

## 4 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.

### 4.1 Try-finally blocks.

Every time a call is made to CreateIt/Freelt or Create/Destroy pair, it should be placed within a try-finally block like this:

Delphi:

```
var a,b,c,d: TVec;
begin
    CreateIt(a,b,c,d);
    try
        ..
        //Your code here.
    finally
        FreeIt(a,b,c,d);
    end;
end;
```

C#:

```
TVec a,b,c,d;
MtxVec.CreateIt(out a, out b, out c, out d);
try
{
    ...
}
finally {
    MtxVec.FreeIt(ref a, ref b, ref c, ref d);
}
```

These have two purposes:

If there is an exception within the try-finally block, the allocated objects and memory will be freed and the program user will be able to retry the calculation with other parameters. It is now easier to track what is created and what is destroyed, because it is clearly visible where create and where destroy is called. MtxVec has internal variables tracking the state of the object cache. If those variables are not zero when application terminates, a call to FreeIt has somewhere been left out.

Do not write code like this:

```
CreateIt (b);  
  
//...some code here  
  
CreateIt (a);  
yourProc (a,b);  
Freeit (b);  
  
//.. some code here..  
FreeIt (a);
```

This makes it difficult to see, if all calls to CreateIt have FreeIt pairs. It is also a good rule of housekeeping to group the code allocating the memory separately from the code doing calculations. This makes code much more readable.

## 4.2 Raising exceptions

Because all code is now 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:

- Allocate memory for the calculation.
- Start calculation
- Display results.
- 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.

### 4.2.1 Invalid parameter passed:

Delphi:

```
begin  
    if a = nil then raise Exception.Create('a= nil');  
    ...  
end;
```

C#:

```
{  
    if (a == nil) throw (new ArgumentNullException());  
    ...  
}
```

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.

### 4.2.2 Reformat the exception

Once an exception has been caught, one might want to notify the user, not of some variable being nil, but actually which procedure failed:

Delphi:

```
try
...
except
  on E: Exception do
    raise Exception.Create(YourMessageHere + ' : ' E.Message);
end;
```

C#:

```
try {
  ...
}
catch (Exception e) {
  new Exception(YourMessageHere + " : " + e.Message);
}
```

This example retains the old messages, adds custom string to it and then raises the exception again, this time with a new message. Every exception will show up in the program as a message box. ShowMessage or MessageDlg or MessageBox should not be used to indicate an error condition. An exception should be raised instead. Raising an exception will safely exit all nested routine calls, free associated memory and finally also show the message box.

### 4.3 By all means indent code.

A procedure should look like this:

Delphi:

```
var a,b,c,d: TVec;
    i: integer;
begin
  CreateIt(a,b,c,d);
  try
    // SomeCodeHere..
    for i := 0 to a.Length-1 do
      begin
        // MoreCodeHere.....
      end;
    finally
      FreeIt(a,b,c,d);
    end;
  end;
end;
```

C#:

```
TVec a,b,c,d;
int i;

MtxVec.CreateIt(out a, out b, out c, out d);
try
{
  //Some code here
  for (i = 0; i < a.Length; i++)
  {
    //More code here
  }
}
```

```
}  
finally {  
    MtxVec.FreeIt(ref a, ref b, ref c, ref d);  
}
```

#### 4.4 Do not create objects within procedures and return them as result:

Delphi:

```
procedure GetVector(var a: TVec);  
begin  
    CreateIt(a);  
end;
```

C#:

```
public void GetVector(ref TVec a)  
{  
    MtxVec.CreateIt(out a);  
}
```

This makes it difficult to track CreateIt/Freelt and Create/Destroy pairs and breaks a few good rules of programming, but has become usual for .NET because of the garbage collector. MtxVec does not rely on the garbage collector to clean up all its memory.

#### 4.5 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 CreateIt/Freelt 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.

## 5 Accessing values of TVec and TMtx

### 5.1 Array property access

Example:

Delphi:

```
var a: TMtx;
begin
    CreateIt(a);
    try
        a[1,0] := 2;
        a.Values[1,0] := 2; //same as above
    finally
        FreeIt(a);
    end;
end;
```

C#:

```
TMtx a;
int i;

MtxVec.CreateIt(out a);
try
{
    for (i = 0; i < a.Length; i++)
    {
        a[1,0] = 2;
        a.Values[1,0] = 2; //same as above
    }
}
finally {
    MtxVec.FreeIt(ref a);
}
}
```

Array properties allow a clean range checked access, but are not the fastest. The array properties performs explicit range checking when assertions are enabled (they can be disabled via a compiler switch).

#### **.NET specifics:**

There is significant performance difference for this type of value access between W32 and .NET. It is recommended to avoid traversing vectors and matrices with array properties in .NET. Instead it is recommended to copy the TMtx/TVec to an array before the loop (TMtxVec.CopyToArray, TMtxVec.CopyFromArray) which can not be vectorized and copying the values back to the TMtx/TVec objects after the loop. The same purpose is achieved with EnlistIt/DismissIt pair.

### 5.2 Direct dynamic array pointer.

Example:

Delphi:

```
var ap: T2DSampleArray;
begin
    CreateIt(a);
    try
        a.Length := 4;
        EnlistIt(a,ap); {is the same as: ap := a.Values}
```

```

        ap[1,0] := 2;
        DismissIt(a,ap) //does nothing, but is required
    finally
        FreeIt(a);
    end;
end;

C#:

double[][] ap;
TMtx a;

MtxVec.CreateIt(out a);
try
{
    a.Length = 4;
    MtxVec.EnlistIt(a,out ap); //Size the array to match the matrix
    // MtxVec.Enlist(a,out ap); //Size the array and copy data from the matrix
    ap[1][0] = 2;
    MtxVec.DismissIt(a,ref ap); //copy array to matrix
}
finally {
    MtxVec.FreeIt(ref a);
}

```

This is the fastest method and recommended for long loops and large matrices. The speed of access is equivalent to static arrays.

**.NET specifics:**

Although under .NET MtxVec also features Enlist and Dismiss, there is one important difference. Under W32 the enlisted arrays point to the same memory as TMtxVec objects, but under .NET, the memory is copied to the declared array. This is OK, as long as the enlisted array is not used to store new values. That is why new methods named EnlistIt and DismissIt have been introduced. DismissIt methods now also offer an additional TMtxVec type object to be passed to the routine to which the array will be copied to. Although the syntax is the same under .NET and W32, the copying takes place only under .NET:

- Enlist copies values to array, Enlist it only sets size of the array.
- DismissIt has two overloads. The one that takes TMtxVec as a parameter also copies values from array.

This is only to ensure same code source between .NET and W32.

Although it is unfortunate that the copying is sometimes (when loop can not be vectorized) required under .NET, the possible performance benefits are far greater. The speed of the copy operation used by Enlist/DismissIt is 4x greater than if the loop is written in C# for long vectors. It is also probably more readable under .NET to call the TMtxVec.CopyToArray/TMtxVec.CopyFromArray

Handling complex data :

Default array property access is not available for complex data because the object can have only one default array property. Default array properties allow the property name to be left out:

```
a[1,0] := 2;
```

instead of:

Delphi:

```
a.Values[1,0] := 2;
```

C#:

```
a.Values[1,0] = 2;
```

The following access methods are semantically equivalent:

Delphi:

```
a.CValues[1,0] := Cplx(2,0); {Cplx is a function that returns a TCplx record type}
```

...

```
var ac: T2DCplxArray;
begin
CreateIt(a);          {similar as a := TVec.Create;}
try
    a.Rows := 4;
    a.Cols := 4;
    a.Complex := True;
    Enlist(a,ac)      {is similar to: ac:= a.CValues}
    ac[1,0] := Cplx(2,0);
    DismissIt(a,ac);
    //.... {same as:}
    a.Values[2,0] := 2;
    a.Values[3,0] := 0;
    //.... {same as:}
    a.CValues[1,0].Re := 2;
    a.CValues[1,0].Im := 0;
    ...
finally
FreeIt(a);  {similar as: a.Destroy; a := nil}
end;
end;
```

C#:

```
a.CValues[1,0] = Math387.Cplx(2,0); //{Cplx is a function that returns a TCplx struct type}
```

...

```
TCplx[][] ac;
TMtx a;
```

```
MtxVec.CreateIt(out a);
try
{
    a.Rows = 4;
    a.Cols = 4;
    a.Complex = true;
    MtxVec.EnlistIt(a,out ac); //Size the array to match the matrix
    ac[1][0] = Math387.Cplx(2,0);
    MtxVec.DismissIt(a,ref ac); //copy array to matrix
    //same as
    a.Values[2,0] = 2;
    a.Values[3,0] = 0;
    //same as
    a.CValues[1,0] = Math387.Cplx(2,0);
} finally {
    MtxVec.FreeIt(ref a);
}
}
```

There is one important consideration, because of the way how Delphi works with dynamic arrays. This should not be attempted in Delphi W32 (does not apply to .NET):

```
var ar: TSampleArray;
    a: TVec;
begin
    CreateIt(a);
```

```
    try
        a.Length = 10;
        a.SetSubIndex(2,2);
        ar := a.Values;    //problem here
        ar[0] := 1;
    finally
        FreeIt(a);
    end;
end;
```

a.Values should **not** be assigned to TSampleArray variable after a call to SetSubIndex or SetSubRange. This could temporarily corrupt the data in the "a" vector. Enlist/Dismiss pair should be used instead.

## 6 Vector/Matrix operations

When working with vectors and matrices it is important to remember, that in most cases the type of the result defines the object which has that method. A few examples:

### 6.1 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.

### 6.2 Matrix, Vector multiply: $B = A*X$

B.TensorProd(A,X);

### 6.3 Vector, Matrix multiply: $B = X*A$

B.TensorProd(X,A);

### 6.4 Vector, Vector multiply: $A = B*X$

A.TensorProd(B,X);

### 6.5 Sparse matrix, vector multiply: $B = S * X$

S.MulRight(X,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.

### 6.6 Vector, sparse matrix multiply: $B = X*S$

S.MulLeft(X,B);

### 6.7 Matrix, sparse matrix multiply: $B = A*S$

S.MulLeft(A,B);

## 6.8 Sparse matrix, matrix multiply: $B = S \cdot A$

```
S.MulRight(A,B);
```

## 6.9 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 TVec.Diag method. To set a diagonal of a matrix there is a TMtx.Diag method. To get a row/column of the matrix call TVec.GetRow (TVec.GetCol) and to set one: TMtx.SetRow. (TMtx.GetCol).

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: TMtx.Eig, TMtx.SVD, TMtx.LQR.

# 7 Memory management

## 7.1 Introduction

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

Delphi:

```
a := TMtx.Create;
a.Rows := 4; {allocates nothing}
a.Cols := 4; {a now holds 16 elements}

a.Rows := 0; {deallocates memory}
a.Free;      {same as a.Rows := 0; a.Cols := 0; a.Free}

CreateIt(a); //similar to a := TMtx.Create;
a.Size(4,4,False); //same as: a.Complex := False; a.Rows := 4; a.Cols := 4;
FreeIt(a); // similar as a.Destroy; a := nil;
```

C#:

```
a = new TMtx();
a.Rows = 4; //a still holds 0 elements
a.Cols = 4; //a now holds 16 elements.

a.Rows = 0; //deallocates memory
a.Free(); //same as a.Rows = 0; a.Cols = 0; a.Free();}

MtxVec.CreateIt(out a); //similar to a = TMtx.Create
a.Size(4,4,false,false); //same as a.Complex = false; a.Rows = 4; a.Cols = 4;
MtxVec.FreeIt(ref a); //similar to a.Free(); a = nil;
```

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

There are some special issues that need to be taken into account when working with matrices. TMtx interfaces highly optimized FORTRAN code. Dynamic memory allocation of two dimensional matrices in Delphi is not done in one single block as expected by FORTRAN routines. For that purpose, TMtx uses its 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:

```
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+];
```

This will access the same matrix element as:

```
a1 := a.Values[j,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.

## 7.2 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.

## 8 Range checking

Due to dynamic memory allocation in one single block, the array range checking is not performed by the compiler for matrices, but TMtx does perform explicit range checking:

```
a[i,j] := 2; {The property setters checks the indexes i and j to be within the bounds of the matrix}
```

This additional range checking is enabled when the code is compiled with assertions turned on. When the assertions are disabled, the additional range checking is also disabled.

## 9 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:

Delphi:

```
var a: TSample;
begin
  a := StrToSample('NAN');
  if IsNan(a) then raise Exception.Create('a = NAN');
end;
```

C#:

```
double a;

a = Math387.StrToSample("NAN");
if (Math387.IsNan(a)) {throw new Exception("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:

Delphi:

```
if a = NAN then ...
```

C#:

```
if (a == Math387.NAN) { ... }
```

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:

Delphi:

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

C#:

```
if (Math387.IsNan(a)) {throw new Exception("a = NAN"); }
if (Math387.IsInf(a)) {throw new Exception("a = NAN"); }
if (Math387.IsNanInf(a)) {throw new Exception("a = NAN"); }
```

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

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

## 10 Serializing and streaming

### 10.1 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 Delphi/CBuilder 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.

## 10.2 Streaming of TVec, TMtx and TSparseMtx

Both TVec and TMtx type objects will be streamed when declared as published properties of a component. 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.) However, sometimes it is necessary to read a text file. Here is how this can be done:

### **.NET Specifics:**

When using the streaming methods from C#, the methods also accept the default CLR Stream type. Only binary streaming is supported.

## 10.3 Write TMtx to a text file

Delphi:

```
AMtx.Size(20,20,true);
AMtx.RandUniform(-1,2);
StringList := TStringList.Create;
try
    { use tab = chr(9) as delimiter }
    AMtx.ValuesToStrings(StringList,#9);
    { Save matrix values to txt file }
    StringList.SaveToFile('ASCIIMtx.txt');
finally
    StringList.Free;
end;
```

C#:

```
MtxVec.CreateIt(out a);
try
{
    string List;
```

```

        a.Size(10,10);
        a.RandGauss();
        a.ValuesToStrings(out List," ", "0.000", "0.000");
        StreamWriter Stream = File.CreateText("C:\\\\ASCIIMtx.Txt");
        Stream.Write(List);
        Stream.Close();
    }
    finally {
        MtxVec.FreeIt(ref a);
    }

```

## 10.4 Read TMtx from a text file

Delphi:

```

StringList := TStringList.Create;
CreateIt(tmpMtx);
try
    { get matrix values from text file }
    StringList.LoadFromFile('ASCIIMtx.txt');
    { use tab = chr(9) as delimiter }
    tmpMtx.StringsToValues(StringList, #9);
    if CheckBox1.Checked then ViewValues(tmpMtx);
finally
    FreeIt(tmpMtx);
    StringList.Free;
end;

```

C#:

```

MtxVec.CreateIt(out a);
try
{
    string List;
    a.Size(10,10);
    StreamReader Stream = File.OpenText("C:\\\\ASCIIMtx.Txt");
    List = Stream.ReadToEnd();
    a.StringsToValues(List, " "); //use space as delimited
    Stream.Close();
}
finally {
    MtxVec.FreeIt(ref a);
}

```

## 11 Input-output interface

### 11.1 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:

Delphi:

```

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

```

C#:

```

enum TPrecision int

```

```
{
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. (double in C#)

## 11.2 Formatting floating point values – FloatToString

Math387 unit (Math387 class for C#) declares the following routines:

```
Delphi:
function FormatCplx(Z: TCplx; const ReFormat: string = ' 0.###;-0.###';
                  const ImFormat: string = '+0.###i;-0.###i'): string;
function FormatSample(X: TSample; Format: string = '0.###'): string; overload;

C#:
public string FormatCplx(TCplx Z, string ReFormat, string ImFormat)
public string FormatSample(double X, string Format)
```

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:

```
Delphi:
function CplxToStr(const Z: TCplx; const Digits: integer = 0): string; overload;
function SampleToStr(const X: TSample; const Digits: integer = 0;
                    {$IFDEF TTSINGLE} Precision: integer = 7 {$ENDIF}
                    {$IFDEF TTDOUBLE} Precision: integer = 15 {$ENDIF}): string; overload;

C#:
public string CplxToStr(TCplx Z, int Digits)
public string SampleToStr(double X, int Digits, int Precision)
```

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:

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

a := 12.123456;
myString := SampleToStr(a); // myString = '12.123456';
```

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

```
Delphi:
function StrToCplx(const Source: string): TCplx; overload;
function StrToSample(const Source: string): TSample; overload;

C#:
public TCplx StrToCplx(const Source: string)
public double StrToSample(const Source: string)
```

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.

## 11.3 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 this 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:

```
Delphi:
a,b: TVec;
begin
....
Report.Print([a,b], ['Matrix a', 'Matrix b']);
Report.NewLine();
Report.SaveToFile('C:\test.txt');
Report.Clear();

C#:
TVec a,b;

....
MtxVec.Report.Print(new TMtxVec[] {a, b}, new string[] {"Matrix A","Matrix 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 usefull methods declared next to those already defined in TStringStream are:

```
report.PrintVec
report.PrintMtx
report.PrintSample
report.PrintCplx
report.PrintSampleArray
report.PrintCplxArray
```

## 11.4 Displaying the contents of the TVec and TMtx

### 11.4.1 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

### 11.4.2 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 (MtxVecEdit class for C#).

## 11.5 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 2. 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. These 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.

Delphi:

```
var a: TVec;
begin
    CreateIt(a);
    try
        a.LoadFromFile('c:\test.vec');
        ViewValues(a); //display a window showing values in "a"
        DrawIt(a); //display a chart of values in "a"
        ...
    finally
        FreeIt(a);
    end;
end;
```

C#:

```
TMtx a;
MtxVec.CreateIt(out a);
try
{
    a.Size(10,10);
    a.RandGauss();
    MtxVecEdit.ViewValues(a, "Test", false, false, false);
    MtxVecTee.DrawIt(a, "");
}
finally
{
    MtxVec.FreeIt(ref a);
}
```



Figure 2

In the top left corner on Figure 2 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 3 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).

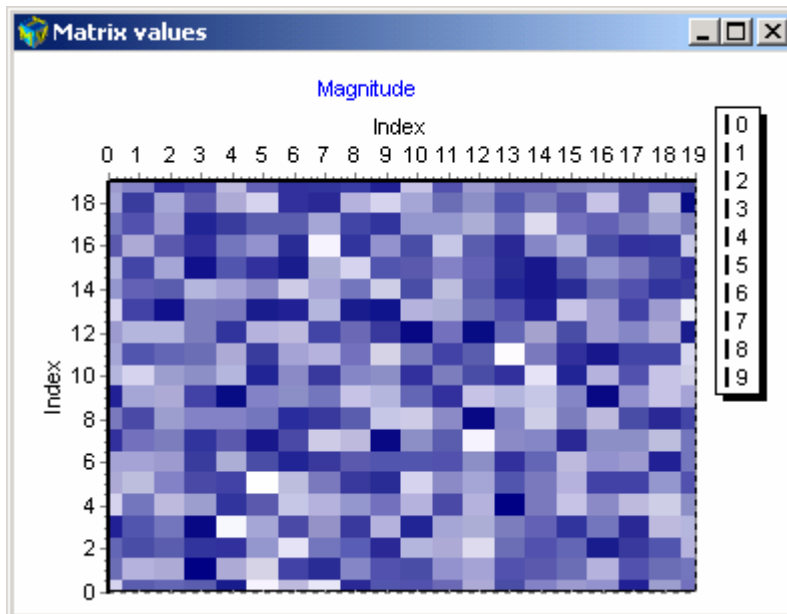


Figure 3

## 12 Getting up to speed

### 12.1 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:

Delphi:

```
for i:= 0 to 1000000-1 do
begin
    Y[i] := (c1*Ax[i]+c2)/Power(1.0 + Power(Bx[i],eA), eB);
end;
```

C#:

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

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

Delphi 2006 with operator overloading:

```
ax.Length := 2000;
bx.Length := 2000;
for i := 0 to 499 do
begin
    Result := YourFunc(Ax,Bx,c1,c2,ea,eb);
end;

function YourFunc(const ax,bx: Vector; c1,c2,ea,eb: TSample): Vector;
begin
    Result := (c1*Ax+c2)/Power(1.0 + Power(Bx,eA), eB);
end;
```

Delphi 2005 and before:

```
ax.Length := 2000;
bx.Length := 2000;
for i := 0 to 499 do
begin
    YourFunc(ax,bx,c,c1,c2,ea,eb);
end;

procedure YourFunc(a,b,Result: TVec; c1,c2,ea,eb: TSample);
var a1,b1: TVec;
begin
    if a.Length <> b.Length then Eraise('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);
    end;
end;
```

C#:

```

a.Length = 2000;
b.Length = 2000;

for (i = 0; i < 499; i++) {

    YourFunc(a,b,c,c1,c2,ea,eb);
}

public void YourFunc(TVec a,TVec b, TVec Result, double c1, double c2 double ea,
double eb)
{

    TVec a1,b1;

    MtxVec.CreateIt(out a1,out b1);
    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 {
        MtxVec.FreeIt(ref a1, ref 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.

## 12.2 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:

Delphi:

```
function MaxwellPDF(x, a: TSample): TSample;
var xx: TSample;
begin
  if (x >= 0) and (a > 0) then
  begin
    xx := x*x*a;
    Result := Sqrt(4*a*INVTWOPI)*xx*Exp(-0.5*xx);
  end
  else Result := NAN;
end;
```

C#:

```
public double MaxwellPdf(double x, double a)
{
  double xx;
  if ((x >= 0.0) && (a > 0))
  {
    xx = x*x*a;
    return Math.Sqrt(4*x*Math387.INVTWOPI)*xx*Math.Exp(-0.5*xx);
  } else return Math387.NAN;
}
```

Vectorized function:

Delphi 2006 with operator overloading:

```
function MaxwellPDF(x : TVec; a: TSample): Vector;
var xx: Vector;
begin
  xx := Sqr(x);
  Result := Sqrt(4*a*INVTWOPI)*xx*a*Exp(-0.5*xx*a);
end;
```

Delphi 2005 and before:

```
procedure MaxwellPDF(X: TVec; a: TSample; Res: TVec);
var Res1: TVec;
begin
  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);
  end;
end;
```

C#:

```
private void MaxwellPdf(TVec X, double a, TVec Res)
{
    TVec res1;

    MtxVec.CreateIt(out Res1);
    try
    {
        tmp.Sqr(x);
        res.Copy(tmp);
        res.Mul(-0.5*a);
        res.Exp();
        res.Mul(tmp);
        res.Mul(Math.Sqrt(4*Math387.INVTWOPI)*a);
    }
    finally {
        MtxVec.FreeIt(ref Res1);
    }
}
```

Block vectorized function:

Delphi 2006 with operator overloading:

```
procedure MaxwellPDF_BlockVec(x, Result: TVec; a: TSample);
var xx: Vector;
begin
    Result.Size(x);
    Result.BlockInit;
    X.BlockInit;

    while not x.BlockEnd do
    begin
        xx := Sqr(x);
        Result.Copy(Sqrt(4*a*INVTWOPI)*a*xx*Exp(-0.5*a*xx));

        Result.BlockNext;
        X.BlockNext;
    end;
end;
```

Delphi 2005 and before:

```
procedure MaxwellPDF(X: TVec; a: TSample; Res: TVec);
var Res1: TVec;
begin
    CreateIt(Res1);
    try
        Res.Size(X);
        Res.BlockInit;
        X.BlockInit;
        while not X.BlockEnd do
        begin
            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;
        end;
    finally
        FreeIt(Res1);
    end;
end;
```

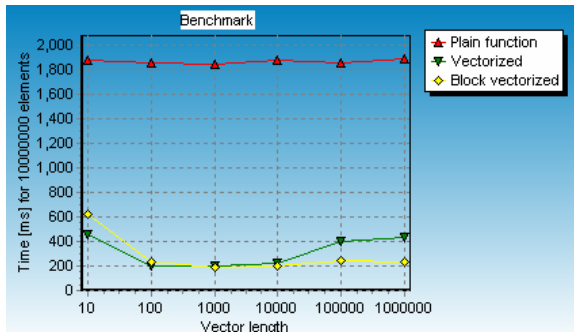
end;

C#:

```
private void MaxwellPdf(TVec X, double a, TVec Res)
{
    TVec res1;

    MtxVec.CreateIt(out Res1);
    try {
        Res.Size(X);
        res.BlockInit();
        x.BlockInit();
        while (!x.BlockEnd) {
            tmp.Sqr(x);
            res.Copy(tmp);
            res.Mul(-0.5*a);
            res.Exp();
            res.Mul(tmp);
            res.Mul(Math.Sqrt(4*Math387.INVTWOPI)*a);
            res.BlockNext();
            x.BlockNext();
        }
    } finally {
        MtxVec.FreeIt(ref Res1);
    }
}
```

On P4 2.4 GHz CPU the vectorized function is about 9.5x faster than the plain function when using Delphi 10. 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 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.

### 12.3 Resolving branching

For the reasons of simplicity the expression `if (x >= 0) and (a > 0) then` has been left out in the vectorized versions. Adding the `(a > 0)` is very simple, but `(x >= 0)` takes a bit more effort. The standard approach to code vectorization is this:

- a.) separate the if's from the loops by using the TVec.FindAndGather or similar methods.
- b.) Write vectorized code instead of loops.
- c.) Use TVec.Scatter for each branch result.
- d.) Try reducing the number of FindAndGather/Scatter pairs by relying on the functions to return NAN or INF.

Example for MaxwellPDF on how to resolve branching:

Delphi 2006:

```

procedure MaxwellPDF(X, Result: TVec; a: TSample);
begin
    Result.Size(x);

    if a <= 0 then
    begin
        Result.SetVal(NAN);
        Exit;
    end;

    TVec(x1).FindAndGather(x, '>=', 0, Indexes);
    TVec(Result1).Size(x1);
    TVec(Result).SetVal(Nan); //assume all are NAN, this is faster,
                               //than checking each individual value

    xx := Sqr(x1);
    Result1 := Sqrt(4*a*INVTWOPI)*a*xx*Exp(-0.5*a*xx);

    TVec(Result).Scatter(Result1, Indexes);
end;

```

## 12.4 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 SSE 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 heavy 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. Under .NET the access to Values/CValues array properties is several times slower than under W32. This is the limitation of the .NET platform. Performance critical code should not traverse the values of the vector via these properties. One way to avoid this is to use the Enlist and DismissIt functions. If possible feed the values to TVec, TMtx and TSparseMtx via arrays by using CopyToArray, CopyFromArray methods. This makes sense only for arrays larger than

- about 10-50 values. In all other aspects the .NET version will perform as fast as the W32 version.
8. For CPU's without SSE, the only way to improve the performance is to strictly follow the rules of block processing. (CPU cache size).
  9. 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.
  10. 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.
  11. When using Delphi 2006 and its new Vector and Matrix types, do not declare input parameters to the functions as being Vector and Matrix types. You can still pass Vector and Matrix types variables to all functions accepting TVec and TMtx classes because of implicit type conversions. Declare Vector and Matrix types only as local variables. As soon as a single Vector or Matrix type variable is present in an expression the enhanced syntax will be available.
  12. Delphi compiler is not capable of an optimization called: collection on common sub expressions. This means, that if there is the same sub expression presented more than once within a larger one, it will be evaluated several times and not only once.
  13. Use parenthesis to notify the compiler which expressions should be evaluated first when using Vector and Matrix classes in Delphi 2006. If you multiply together 4 scalars and one vector it is not irrelevant in which order are they multiplied. Vector can have many thousand elements and 4 scalars can be multiplied together very quickly, but the compiler does not know that. Expression: `vec*scalar*scalar` will call `TVec.Multiply 2x` and expression `scalar*scalar*vec` will call it only once. Therefore, sometimes it is wise to use parenthesis: `vec*(scalar*scalar)`
  14. Try reducing the number of FindAndGather/Scatter calls by relying on the functions to return NAN or INF when resolving the branching. One or two such pairs will do no serious harm, but for complex branching conditions it makes sense to do computation first as if though all the values are valid and then pass the result through a loop that traverses the vector, checks the conditions and fills in any special values. An example of such processing are `TMtxVec.Power` functions. The power function has a very long list of special cases branching especially for processing complex numbers.
  15. There are no compilers in the world today, that can vectorize the code and resolve the branches automatically.

## 13 Debugging MtxVec

Both Delphi and Visual Studio.NET offer an excellent debugger. However there are a few things that have to be addressed separately.

### 13.1 Memory leaks

The programmer should make sure to always match the Create and Free methods of objects and CreateIt/FreeIt pairs for TVec and TMtx objects, as already emphasized. MtxVec unit holds two global variables: Controller.MtxCacheUsed and Controller.VecCacheUsed. Their value will show the number of unfreed objects. 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 FreeIt 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.

#### 13.1.1 Debugging under .NET

In order to see error message about unfreed TVec/TMtx objects (cached or not) under .NET and not some cryptic error message from the CLR, the following line must be placed in the Dispose method of your main unit (the one that is freed last):

```
protected override void Dispose( bool disposing )
{
    if( disposing )
    {
        if (components != null)
        {
            components.Dispose();
        }
    }
    base.Dispose( disposing );
    Dew.Math.Units.MtxVec.Controller.Free(); // used to safely Free memory
    controller
}
```

This code is need only when MtxVec is used together with Winforms. In case of VCL or VCL.NET application, the command is called by a finalization section via dispose pattern. The code line mentioned is “mandatory” in our experience to ensure a memory leak free application under .NET. We analyzed different aspects of implementing finalizers, and came to the conclusion that in case of forgetting to explicitly free an object will result in slow and memory applications where it would become very hard to determine the cause of the problem. So instead of hiding the error, we rather decided to expose it immediately.

### 13.2 Memory overwrites

When using TVec and TMtx memory overwrites should be a thing of the past. But if you chose to work directly with unmanaged 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. For W32 it helps a lot, if range checking offered by the Delphi (W32) compiler is turned on. (Project -> Options -> Compiler -> Run time errors -> Range checking). The range checking mechanism offered by the W32 compiler raises false alarms, if Enlist/Dismiss is used in pair with SetSubRange/SetSubIndex:

```
{$R-} //make sure to disable range checking for this routine
```

```
var a,b: TVec;
    ap: TSampleArray;
begin
    CreateIt(a,b);
    try
        a.LoadFromFile('c:\test.vec');
        b.SetSubrange(a,2,10);
        Enlist(b,ap);
        ap[0] := ...//possible false alarms, but only under W32
        DismissIt(b,ap);
    finally
        FreeIt(a,b);
    end;
end;
```

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 AV's will be raised much closer to the actual cause of the problem.

## 14 Getting ready to deploy

Once the app has been debugged and is ready to be deployed, three files required by MtxVec have to be included in the distribution package. These files are located in the windows\system or winnt\system32 directory: MtxVec.lapackd.dll, MtxVec.spld.dll and bdspl.dll. If the application uses sparse.pas, MtxVec.sparsed.dll is also required.

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, for P4 SSE2 instruction set and for P4 SSE3 instruction set. The appropriate code version is selected automatically, when the libraries are loaded.

## 15 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 P4 instruction set

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.

### 15.1 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
ArcCosh	TMtxVec	$a[i] = \text{ArcCosh}(a[i])$ $a[i] = \text{ArcCosh}(b[i])$	SSE2/SSE3
ArcCot	TMtxVec	$a[i] = \text{ArcCot}(a[i])$ $a[i] = \text{ArcCot}(b[i])$	SSE2/SSE3
ArcCoth	TMtxVec	$a[i] = \text{ArcCoth}(a[i])$ $a[i] = \text{ArcCoth}(b[i])$	SSE2/SSE3
ArcCsc	TMtxVec	$a[i] = \text{ArcCsc}(a[i])$ $a[i] = \text{ArcCsc}(b[i])$	SSE2/SSE3
ArcCsch	TMtxVec	$a[i] = \text{ArcCsch}(a[i])$ $a[i] = \text{ArcCsch}(b[i])$	SSE2/SSE3
ArcSec	TMtxVec	$a[i] = \text{ArcSec}(a[i])$ $a[i] = \text{ArcSec}(b[i])$	SSE2/SSE3
ArcSech	TMtxVec	$a[i] = \text{ArcSech}(a[i])$ $a[i] = \text{ArcSech}(b[i])$	SSE2/SSE3
ArcSin	TMtxVec	$a[i] = \text{ArcSin}(a[i])$ $a[i] = \text{ArcSin}(b[i])$	SSE2/SSE3
ArcSinh	TMtxVec	$a[i] = \text{ArcSinh}(a[i])$ $a[i] = \text{ArcSinh}(b[i])$	SSE2/SSE3
ArcTan	TMtxVec	$a[i] = \text{ArcTan}(a[i])$ $a[i] = \text{ArcTan}(b[i])$	SSE2/SSE3
ArcTan2	TMtxVec	$a[i] = \text{ArcTan2}(b[i], c[i])$	SSE2/SSE3
ArcTanh	TMtxVec	$a[i] = \text{ArcTanh}(a[i])$ $a[i] = \text{ArcTanh}(b[i])$	SSE2/SSE3
Cbrt	TMtxVec	$a[i] = (a[i])^{1/3}$ $a[i] = (b[i])^{1/3}$	SSE2/SSE3
Copy	TMtxVec	$a[i] = b[i]$	SSE2/SSE3
Cos	TMtxVec	$a[i] = \text{Cos}(a[i])$ $a[i] = \text{Cos}(b[i])$	SSE2/SSE3
Cosh	TMtxVec	$a[i] = \text{Cosh}(a[i])$ $a[i] = \text{Cosh}(b[i])$	SSE2/SSE3
Cot	TMtxVec	$a[i] = \text{Cot}(a[i])$ $a[i] = \text{Cot}(b[i])$	SSE2/SSE3
Coth	TMtxVec	$a[i] = \text{Coth}(a[i])$	SSE2/SSE3

		$a[i] = \text{Coth}(b[i])$	
Csc	TMtxVec	$a[i] = \text{Csc}(a[i])$ $a[i] = \text{Csc}(b[i])$	SSE2/SSE3
Csch	TMtxVec	$a[i] = \text{Csch}(a[i])$ $a[i] = \text{Csch}(b[i])$	SSE2/SSE3
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
Exp10	TMtxVec	$a[i] = \text{Exp10}(a[i])$ $a[i] = \text{Exp10}(b[i])$	SSE2/SSE3
Exp2	TMtxVec	$a[i] = \text{Exp2}(a[i])$ $a[i] = \text{Exp2}(b[i])$	SSE2/SSE3
Frac	TMtxVec	$a[i] = \text{fractional part of } a[i]$ $a[i] = \text{fractional part of } b[i]$	SSE2/SSE3
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
InvCbrt	TMtxVec	$a[i] = (a[i])^{-1/3}$ $a[i] = (b[i])^{-1/3}$	SSE2/SSE3
InvSqrt	TMtxVec	$a[i] = (a[i])^{-1/2}$ $a[i] = (b[i])^{-1/2}$	SSE2/SSE3
IsEqual	TMtxVec	$a[i] = ? = b[i]$	
Ln	TMtxVec	$a[i] = \text{Ln}(a[i])$ $a[i] = \text{Ln}(b[i])$	SSE2/SSE3
Log10	TMtxVec	$a[i] = \text{Log10}(a[i])$ $a[i] = \text{Log10}(b[i])$	SSE2/SSE3
Log2	TMtxVec	$a[i] = \text{Log2}(a[i])$ $a[i] = \text{Log2}(b[i])$	SSE2/SSE3
LogN	TMtxVec	$a[i] = \text{LogN}(a[i])$ $a[i] = \text{LogN}(b[i])$	SSE2/SSE3
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
Round	TMtxVec	$a[i] = \text{nearest integer to } a[i]$ $a[i] = \text{nearest integer to } b[i]$	SSE2/SSE3
Sec	TMtxVec	$a[i] = \text{Sec}(a[i])$ $a[i] = \text{Sec}(b[i])$	SSE2/SSE3
Sech	TMtxVec	$a[i] = \text{Sech}(a[i])$ $a[i] = \text{Sech}(b[i])$	SSE2/SSE3
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
SinCos	TMtxVec	$b[i] = \text{Sin}(a[i]), c[i] = \text{Cos}(a[i])$	SSE2/SSE3
Sinh	TMtxVec	$a[i] = \text{Sech}(a[i])$ $a[i] = \text{Sech}(b[i])$	SSE2/SSE3
SinhCosh	TMtxVec	$b[i] = \text{Sech}(a[i]), c[i] = \text{Sech}(a[i])$	SSE2/SSE3
Sqr	TMtxVec	$a[i] = (a[i])^2$ $a[i] = (b[i])^2$	SSE2/SSE3
Sqrt	TMtxVec	$a[i] = (a[i])^{1/2}$ $a[i] = (b[i])^{1/2}$	SSE2/SSE3
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
Tanh	TMtxVec	$a[i] = \text{Tanh}(a[i])$ $a[i] = \text{Tanh}(b[i])$	SSE2/SSE3
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
ThreshBottom ThreshTop	TMtxVec	Limit the upper or lower value range	SSE2/SSE3

## 15.2 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

## 15.3 Complex number specific

Expj	TMtxVec	$a[i] = \exp(-j*w*b[i])$	SSE2/SSE3
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	$a[i] = a[i]*\text{Conj}(b[i])$ $a[i] = b[i]*\text{Conj}(c[i])$	SSE2/SSE3

Conj	TMtxVec	$a[i] = a[i].\text{Re} - a[i].\text{Im}$ $a[i] = b[i].\text{Re} - b[i].\text{Im}$	SSE2/SSE3
PhaseSpectrum	TMtxVec	$a[i] = \text{Arctan2}(b[i].\text{Im}/b[i].\text{Re})$	SSE2/SSE3
PowerSpectrum	TMtxVec	$a[i] = (\text{sqr}(b[i].\text{Re}) + \text{sqr}(b[i].\text{Im}))$	SSE2/SSE3
FlipConj	TMtxVec	$a[i].\text{Re} = b[i].\text{Im}$ $a[i].\text{Im} = -b[i].\text{Re}$	SSE2/SSE3

## 15.4 Size, streaming and storage

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

## 15.5 FFT's

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

## 15.6 Linear algebra

Function	Class	Math expression	Features
LUSolve	TMtx	$A \cdot X = B$ , solves for X	SSE2/SSE3, SMP
LQRSolve	TMtx	Least squares solution	SSE2/SSE3, SMP
SVDSolve	TMtx	Singular value solution	SSE2/SSE3, SMP
Eig	TMtx	Eigenvalues and eigen vectors	SSE2/SSE3, SMP
EigGen	TMtx	Generalized eigen values	SSE2/SSE3, SMP
Transp	TMtx	Transpose	SSE2/SSE3
Adjung	TMtx	Adjugate	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: $A = \text{Fun}(B)$	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

### 15.7 Matrix conversions

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

### 15.8 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

## 16 Delphi 2006 and operator overloading

After many years Delphi 2006 finally got the ability to overload operators. This ability is supported only for records, but with some clever logic it can be made to work for classes also. To support operator overloading in Delphi 2006 MtxVec declares two new types Vector and Matrix. These two types declared in MtxExpr unit are records which encapsulate TVec and TMtx objects respectively. The usage of the new types is best demonstrated with an example:

```
procedure Test;
var b,c: TVec;
begin
  b := TVec.Create;
  c := TVec.Create;

  b.Size(1000);
  c.Size(b);
  b.RandUniform(0.5,1.5);
  BoseEinsteinPDF(b,0.3,0.1,c);

  b.Free;
  c.Free;
end;
```

When using Vector objects:

```
procedure Test;
var b,c: Vector;
begin
  b.Size(1000);
  c.Size(b);
  TVec(b).RandUniform(0.5,1.5);
  BoseEinsteinPDF(b,0.3,0.1,c); //b and c were implicitly converted to TVec
end;
```

All properties of TVec and TMtx are mapped to Vector and Matrix records including Values and CValues array properties. To access methods of encapsulated objects the Vector (Matrix) has to be typecasted to TVec (or TMtx). The performance of the code using Vector and Matrix objects is on par with TVec and TMtx for longer vectors (1000 elements and beyond). The +,-,/ and \* operators are strictly per element operations (+, -, /, \*). To perform matrix multiplication or division use the Mul and Divide functions.

### 16.1 Implicit type conversions

Implicit type conversions can help clean up the code. The string can be automatically converted to a complex number:

```
var a: TCplx;
....
a := '1+2i';
```

When a function requires an array of double, Vector or Matrix can be passed instead. Implicit type conversions will result in dereferencing Vector.Data.Values1D pointer which is an array of double. Explicit type conversions of Vector/Matrix to TSampleArray or TCplxArray will result in a copy operation.

Passing Vector or Matrix to a function accepting TVec or TMtx will pass the contained object. You can also mix TVec/TMtx and Vector/Matrix in the same expressions:

```
var am,bm: Matrix;
    av: Vector;
    ac: TCplx;
begin
```

```

ac := '1+2i'; //Convert from string to complex number
am.Size(5,2); //matrix size
av.Size(10); //vector size

bm := am*av + ac + 2; //always by value operations
                        // ./ and .* (not linear algebra)

//To make linear algebra multiplication and division use functions

bm := Divide(am,bm) + 2; //matrix division with least squares QR system solver
bm := Mul(am,bm) + 2; //matrix multiply

//of course you can mix matrices and vectors

av := Mul(av,bm) + 2; //vector from left and matrix multiply
end;
```

## 16.2 How to organize your code

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 Vector and Matrix will be implicitly converted (dereferenced) to TVec and TMtx anyway. By doing that you will also not have to think about what it means to pass Vector or Matrix as const or as by var parameters. Passing records as parameters also requires more CPU power. All objects declared as global objects or fields should also remain of TVec/TMtx type. Vector/Matrix should be used mostly for local variables to improve the readability of the code. 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 Create/Free usage still apply.

Vector and Matrix can alternatively also be created. In this case, their origin (from cache or not from object cache) can be controlled. In the example below the available options are listed. Usually if the records constructor is called, the object contained is created and is not obtained from object cache.

```

var am: Matrix;
    av: Vector;
begin
    am.Size(5,2); //matrix size, object obtained from cache

    am = Matrix.Create(10,10); //matrix size, object created (!!

    av = Vector.Create(10); //vector size, object created (!!

    av = Vector.Create(10,true); //vector size, complex, object created (!!

    av = Vector.Create(true); //object obtained from cache

    av = Vector.Create(false); //object created
end;
```

Such design allows you to control TVec/TMtx creation as local or global variables.

## 17 C++ Builder support

TVec and TMtx classes are written in Delphi and C++ Builder generates appropriate header files automatically. However C++ syntax allows more flexible language constructs than pascal. *MtxVecCpp.h* is designed to merge native C++ coding style and part of MtxVec library, which is coded in Delphi. C++ developers may easily declare vectors and matrices as local variables and enjoy the support for operator overloading.

### 17.1 Extra features

#### 17.1.1 Smart pointers

By analogy with smart pointers in C++, MtxVec library defines shell classes in *MtxVecCpp.h* – *Vector* and *Matrix* for TVec and TMtx classes respectively. Pascal style forces to declare variables as below:

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

C++ style looks much simpler:

```
Vector v;
...
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.

#### 17.1.2 Vector, Matrix, CVector, CMatrix, SparseMatrix

Classes *Vector*, *Matrix*, *SparseMatrix* are smart shells for classes TVec, TMtx, TSparseMtx respectively. Classes *CVector* and *CMatrix* are defined to allow a more comfortable access to complex elements, but are otherwise identical as *Vector* and *Matrix*.

*MtxVecCpp.h* introduces additional classes *TVector* and *TMatrix*. 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 *TSparseMtx* classes. *Vector*, *Matrix* and *SparseMatrix* should suffice.

Examples of definitions:

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

Matrix m1(10, 10); // real matrix 10x10
```

```
Matrix m2(10,10,true); // complex matrix 10x10
CMatrix m3(10,10);    // complex matrix 10x10
```

### 17.1.3 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.

```
Vector 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
```

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];
//or (faster)
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];
//or (faster)
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];
//or (faster)
v.IValues(0) = v.IValues(1) + v.IValues(2);
```

Matrix also requires the long form to access the elements:

```
Matrix m(3,3);
m->Values[0][0] = m->Values[1][1] + m->Values[1][2];
//or (faster)
m.Values(0,0) = m.Values(1,1)+ m.Values(1,2);
```

Comparative operators <, >, <=, >=, ==, != store the result of the comparison in a vector, each element of which is an integer value of "1" or "0".

```
Vector 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
```

### 17.1.4 Working with elements and objects

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);
```

Many methods of TVec and TMtx are also available as functions to increase the readability of the code.

### 17.1.5 Subranges

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:

```
Vector 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:

```
Matrix 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:

```
Matrix 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: `Matrix.Operator() (int,int,int,int)` which returns a submatrix which is not merely a view of the original, but a copy of selected elements.

```
Matrix 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).

### 17.1.6 Methods and functions

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

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

Is equivalent to:

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

All standard function from `<math>` header file also defined in `MtxVecCpp.h`. So, the next syntax is allowed as well:

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

### 17.1.7 Range Checking

All Value access methods are range checked, when debugging is active. `MtxVecCPP.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.

## 17.2 Building and debugging

If you wish to build your application without using run time packages, you have to add `MtxVecCpp.cpp` to your project. This will add required link statements for the lib files necessary to build the application. If that file is not included BCB will give a very long list of unresolved external symbols. With BCB6 trial and registered and with BCB2006 trial version the file is located within BCB's `include\vcl` directory. With the registered version for BCB2006 the file is located inside the include directory pointed to by `$(MTXVEINCLUDEPATH)` BDS environment variable.

MtxVec stores data within Vector, Matrix, TVec, TMtx, objects. BCB debugger does not allow inspecting an entire array of values, but only one value by one value. For that purpose TVec and TMtx have two properties named RealValues and ComplexValues. They return first 1000 values of the object as single string. The first few characters describe the Length, Rows, Cols, Complex properties values and the rest are the values. This makes it much easier to inspect the values of the object.

## 17.3 Advanced Topics

### 17.3.1 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:

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

Allocate static object:

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

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

### 17.3.2 Without redundant copying

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

```
Vector 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:

```
Vector 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 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.

### 17.3.3 More about subranges

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

```
Vector 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:

```
Vector a(4); a->Ramp(1,1); // a is 1,2,3,4  
Case 1:  
Vector 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:  
Vector 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 3:  
Vector b = a(0,1); // doesn't make a copy  
                 // a and b point to the same data  
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.

## 17.4 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 "MtxVecCPP.h"  
#include "MtxVec.hpp"  
#include "MtxVecTee.hpp"  
#include "MtxVecEdit.hpp"  
//-----  
#pragma package(smart_init)  
#pragma resource "*.dfm"  
TForm1 *Form1;  
  
//-----  
  
void __fastcall TForm1::Button1Click(TObject *Sender)
```

```

{
    Vector x, y, spec;
    Matrix 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);

}
    
```

## 17.5 Loading and saving a matrix in a text file.

One more example:

```

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

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

//-----
void __fastcall TForm1::Button1Click(TObject *Sender)
{
    Vector x, y, spec;
    Matrix 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.

## 18 Compatibility breaking changes

### 18.1 From version 1.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. Some functions have been removed and others with new names have been added.

### 18.2 From version 2.0

- `Sgn` function has been renamed to `SgnMul` and a new `Sgn` function is defined instead.