MtxVec v3

Users Guide to MtxVec for VS.NET

MtxVec version v3, rev 1.8
© 1999-2008 Dew Research
www.dewresearch.com

MtxVec v3 2

Users Guide to MtxVec for VS.NET 2

1 Introduction 4

1.1 Why MtxVec 4

1.2 How fast is MtxVec 5

2 Organization 5

2.1 Compiler support 5

3 Quick start 6

4 MtxVec programming interface 7

4.1 Object hierarchy 7

4.2 Mathematical expressions and operator overloading 7

4.2.1 Element by element Vector/Matrix expression types 8
4.2.2 Linear algebra Vector/Matrix expression types 9
4.2.3 Linear algebra with TVec and TMtx types 9
4.2.4 Implicit type conversions 10

4.3 Method conventions 10

4.4 Range checking 10

4.5 Making use of the abstract class 11

4.5.1 Writing abstract class code 12

4.6 Function parameters 12

4.7 Indexes, ranges and sub-ranges 13

4.8 TVec and TMtx methods as functions 14

4.9 Complex data 14

4.10 MtxVec types 15

4.10.1 TSample 15
4.10.2 TCplx 15
4.10.3 TSampleArray, TCplxArray 15

5 Accessing values of Vector and Matrix 17

5.1 Array property access 17

5.2 Handling of complex data: 17

6 Memory management 17

6.1 Introduction 17

6.2 Using TVec and TMtx with managed and unmanaged memory 18

6.3 In-place/not-in-place operations 19

7 Range checking 19

8 Why and how NAN and INF 19

9 Serializing and streaming 20

9.1 Streaming with TMtxComponent 20

9.2 Streaming of TVec, TMtx and TSparseMtx 20

9.3 Write TMtx/Matrix to a text file 21

9.4 Read TMtx/Matrix from a text file 21

10 Input-output interface 21

10.1 Reading and writing raw data ... 21

10.2 Formatting floating point values – FloatToString ... 21

10.3 Displaying the contents of the TVec and TMtx ... 22
10.3.1 Displaying data as a delimited text ... 22
10.3.2 Displaying data within a grid. ... 22
10.3.3 Viewing data by using built-in Values editor ... 23

10.4 Charting and drawing ... 24

11 Programming style ... 25

11.1 Mixing TVec/TMtx and Matrix/Vector types. ... 25

11.2 Try-finally blocks and unmanaged memory ... 25

11.3 Raising exceptions ... 25
11.3.1 Invalid parameter passed: ... 26

11.4 When to use CreateIt/FreeIt ... 26

12 Getting up to speed – the benchmarks ... 27

12.1 Floating point code vectorization ... 27

12.2 Block based processing ... 28

12.3 Garbage Collector and High Performance numerical math ... 31

12.4 Common pitfalls ... 33

12.5 Code vectorization methods ... 35

12.6 Enabling the expressions for Multi-core CPU's ... 36

12.7 Intel complex number VML functions. ... 36

13 Debugging MtxVec ... 37

13.1 Viewing the values of Vector and Matrix in the debugger as an array ... 37

13.2 Printing current values of variables ... 37

13.3 Calling DrawIt and ViewValues ... 38

13.4 Memory leaks ... 39

13.5 Memory overwrites ... 39

14 Getting ready to deploy ... 40

14.1 Compact MtxVec ... 40

14.2 Managing the threads ... 40

15 Major function groups ... 41

15.1 Basic vector math ... 41

15.2 Statistical ... 43

15.3 Complex number specific ... 43

15.4 Size, streaming and storage ... 44

15.5 FFT's ... 44

15.6 Linear algebra ... 44

15.7 Matrix conversions ... 45

15.8 Miscellaneous matrix routines ... 45

1 Introduction

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

1. Comprehensive set of mathematical and statistical functions
2. Substantial performance improvements of floating point math by exploiting the SSE2, SSE3 and SSE4 instruction sets.
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). 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 CPU's provided by Intel with their Math Kernel library. MtxVec will also take advantage of all features of AMD CPU's.

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

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

1.1 Why MtxVec

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

1.2 How fast is MtxVec

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

2 Organization

MtxVec library is organized in to three levels - computational level, objects level and component level. The interface to the computational level is a set of class static functions, which are declared in Dew.Math.Units.nmkl, Dew.Math.Units.ippspl, and other classes and implemented as external DLLs - MtxVec.Lapack2d.dll, MtxVec.Spld2.dll, MtxVec.Sparse2d.dll, MtxVec.FFT.dll, MtxVec.Random.dll and MtxVec.Vmld.dll. The first level is not documented. The second level is written in Delphi.NET and the third partly in Delphi.NET and partly in C#. This level introduces vector and matrix objects, complex numbers and a number of class static utility functions declared in:

Dew.Math.Units.MtxVec,
Dew.Math.Units.Polynoms,
Dew.Math.Units.Math387,
Dew.Math.Units.Probabilities,
Dew.Math.Units.SpecialFuncs,
Dew.Math.Units.Optimization,
Dew.Math.Units.Toeplitz,
Dew.Math.Units.Sparse,
Dew.Math.TeeChart.

The second level is the “run-time” part of the MtxVec library. Third level is formed by a set of components which are build on the second level to offer a centralized and quick access to large parts of the library many times also offering ready to use user interface.

This user’s guide concentrates mostly on the second level, specifically the classes declared within the main Dew.Math namespace, Dew.Math.Units.Math387 class, Dew.Math.Units.MtxExpr class and touches some features from Dew.Math.TeeChart class. It gives a good overview on the concept and core features of MtxVec.

2.1 Compiler support

MtxVec v3 for VS.NET supports the following compilers:

	Versions
Visual Studio C#, VB.NET, C++	2005.NET, 2008.NET

3 Quick start

```
using Dew.Math;
using Dew.Math.Units;
using Dew.Math.Editors;
using Dew.Math.Controls;
using Dew.Math.Tee;

private void button1_Click(object sender, EventArgs e)
{

    TCplx ac = "1+2i";
    Matrix am = MtxExpr.RandGauss(5, 5, true); //5x5 complex matrix with Gaussian
    Vector av = MtxExpr.Ramp(25);  //real vector = [0, 1, 2,...., 23, 24]
    Matrix bm = am * av + ac + 2; // (*,/) treat Matrices as vectors
                              // Same as: ./  and .* operators (not linear algebra)

    MtxVecEdit.ViewValues(bm,"Complex matrix",true); //display the matrix contents


// To make linear algebra multiplication and divison use functions


    bm = MtxExpr.Divide(am,bm) + 2;  //matrix divison and add 2 to all elements
    bm = MtxExpr.Mul(am,bm) + 3;  //matrix multiply

//of course you can mix matrices and vectors

    double[] arr = null;
    av.Resize(5,false);
    av = MtxExpr.Mul(av,bm) + 2;  //vector from left and matrix multiply
    av.CopyToArray(ref arr);   //copy "complex" data to array of double length
    Vector bv = (Vector) arr;  //copy data from array to vector,

    MtxVecEdit.ViewValues(bv, "Vector from array", true); //display the vector
    for (int i = 0; i < am.Rows; i++)   //standard loop example
    {
          av.Values[i] = Math387.Real(am.CValues[i,0]*av.Values[i]);
    }
}
```

4 MtxVec programming interface

4.1 Object hierarchy

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

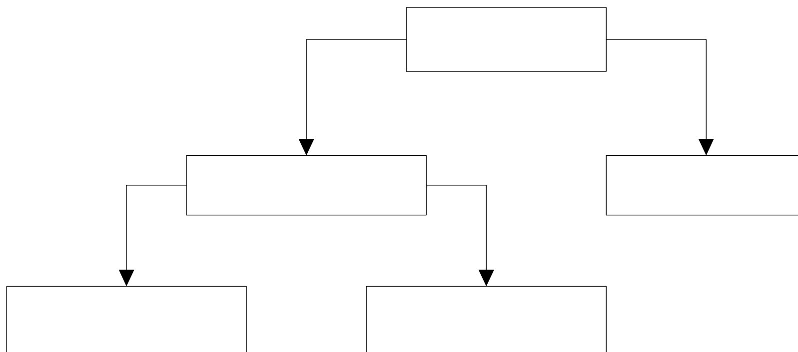


Figure 1 MtxVec class hierarchy

TVec and TMtx classes are from Delphi 2006 on also encapsulated with Vector and Matrix classes:

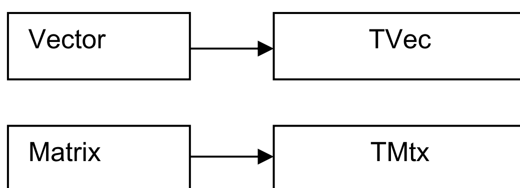


Figure 2 Mapping to value classes (struct).

All methods and properties of TVec and TMtx are also accessible from Vector and Matrix types which are declared as value classes (struct). Vector and Matrix are not simply derived from TVec and TMtx because in Delphi Vector and Matrix are value classes, for which it is not necessary to call constructors explicitly. Because that is not also a feature of C#, for all practical purposes, Vector and Matrix behave as if though they are derived from TVec and TMtx.

4.2 Mathematical expressions and operator overloading

Custom operator overloading means that the programmer can use /,*,-,+ operators also for his own types and not only with built in types. To support operator overloading MtxVec declares types: Vector and Matrix. The usage of the new types is best demonstrated with an example:

```
private void Test()
{
    TVec c = new TVec();
    TVec b = new TVec();

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

When using Vector objects:

```
private void Test()
```

```
{
    Vector c = new Vector(0);

    b = MtxExpr.RandUniform(1000, 0.5, 1.5);
    Probabilities.BoseEinsteinPDF(b, 0.3, 0.1, c);
}
```

All properties and methods of TVec and TMtx are mapped to Vector and Matrix value classes including Values and CValues array properties. The performance of the code using Vector and Matrix objects is on pair 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.

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

```
private Vector Test(TVec p)
{
    Vector bv = new Vector(0);

    bv.Copy(p);
    Vector Result = bv * p; //same as Result.Sqr(p);
    return Result;
}
```

Dew.Math.Units.MtxExpr declares also functions which return Vector or Matrix as a result. These are equivalent to the methods of TMtxVec and its descendants. For example, the following calls give the same result:

```
private void Test()
{
    Vector bv = new Vector(0);
    Vector av = new Vector(0);
    TVec avt = new TVec();

    avt.Sin(av);
    bv.Sin(av);
    Vector cv = MtxExpr.Sin(av);
}
```

All those methods of TVec and TMtx are also available as functions in MtxExpr.pas, where the result of the function is a single Vector (TVec) or a single Matrix (TMtx).

4.2.1 Element by element Vector/Matrix expression types

Multiply two vectors element by element:

```
Vector bv = new Vector(0);
Vector av = new Vector(0);
//..
Vector cv = av*bv;

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

To multiply vector and matrix element by element:

```
Matrix bm = new Matrix(0,0);
Vector av = new Vector(0);
//..
Vector cv = av*bm;

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

The same principles are used for addition (+), substraction (-) and divison. It is of course possible to mix Matrix and Vector types with integers, doubles, reals and complex numbers in the expressions.

4.2.2 Linear algebra Vector/Matrix expression types

Matrix, Matrix multiply: C = A * B, C = MtxExpr.Mul(A,B);

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

Matrix, Vector multiply: b = A*x, b = MtxExpr.Mul(A,x);

Vector, Matrix multiply: b = x*A, b = MtxExpr.Mul(x,A);

Vector, Vector multiply: A = b*x, A = MtxExpr.Mul(b,x);

Vector, Matrix divide : b = x/A, b = MtxExpr.Divide(x,A);

Matrix, Matrix divide : B = C/A, B = MtxExpr.Divide(C,A);

Vector, Matrix ldivide : b = A\x, b = MtxExpr.LDivide(A,x);

Matrix, Matrix ldivide : B = A\C, B = MtxExpr.LDivide(A,C);

4.2.3 Linear algebra with TVec and TMtx types

When working with TVec vectors and TMtx 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:

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

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

Matrix, Vector multiply: b = A * x, b.TensorProd(A,x);

Vector, Matrix multiply: b = x * A, b.TensorProd(x,A);

Vector, Vector multiply: A = b * x, A.TensorProd(b,x);

Sparse matrix, vector multiply: B = S * b, S.MulRight(b,B);

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

Vector, sparse matrix multiply: B = x*S, S.MulLeft(x,B);

Matrix, sparse matrix multiply: B = A*S, S.MulLeft(A,B);

Sparse matrix, matrix multiply: B = S*A, S.MulRight(A,B);

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 Vector.Diag

method. To set a diagonal of a matrix there is a Matrix.Diag method. To get a row/column of the matrix call Vector.GetRow (Vector.GetCol) and to set one: Matrix.SetRow. (Matrix.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: Matrix.Eig, Matrix.SVD, Matrix.LQR.

4.2.4 Implicit type conversions

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

```
TCplx ac = "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 double[] or TCplx[] will result in a copy operation.

Passing Vector or Matrix to a function accepting TVec or TMtx will pass the contained object.

```
Matrix am = MtxExpr.RandGauss(5, 5, true); //5x5 complex matrix with Gaussian
Vector av = MtxExpr.Ramp(25);  //real vector = [0, 1, 2,...., 23, 24]
Matrix bm = am * av + ac + 2; // (*,/) treat Matrices as vectors
                              // Same as: ./  and .* operators (not linear algebra)

// To make linear algebra multiplication and divison use functions


bm = MtxExpr.Divide(am,bm) + 2;  //matrix divison and add 2 to all elements
bm = MtxExpr.Mul(am,bm) + 3;  //matrix multiply

//of course you can mix matrices and vectors

double[] arr = null;
av.Resize(5,false);
av = MtxExpr.Mul(av,bm) + 2;  //vector from left and matrix multiply
av.CopyToArray(ref arr);   //copy "complex" data to array of double length
Vector bv = (Vector) arr;  //copy data from array to vector,
```

4.3 Method conventions

TMtxVec classes and descendants have a list of methods which may be called like this:

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

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

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

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

4.4 Range 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 **assertions,** 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.)

4.5 Making use of the abstract class

Many methods can accept any TMtxVec descendant class:

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

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

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

The following versions will also work flawlessly:

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

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

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

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

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

```
TMtxVec.Size(TMtxVec Src, bool aComplex);
```

Note: The size method does not preserve the data in the destination object. This is not needed because the destination is overwritten anyway. To resize the object and keep existing data call the Resize method.

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

TMtxVec.Values1D - array property to access real values

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

//aTMtxVecObject can be TVec, TMtx, Matrix or Vector
```

TMtxVec.CValues1D - array property to access complex values

```
aTMtxVecObj.CValues1D[1] = Math387.Cplx(1, 0); //sets complex value at index 1 to 1
```

TMtxVec.PValues1D - function returns a pointer to real value

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

TMtxVec.PCValues1D - function returns a pointer to complex value

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

4.5.1 Writing abstract class code

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

```
private void CustomExpj(TMtxVec Dst, 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): PValues1D, PCValues1D, PIValues1D,
- 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:

```
private void CustomExpj(TMtxVec Dst, SrcOmega)
{
    Vector a = Vector(0);
    Dst.Size(SrcOmega,true);
    a.SetSubRange(Dst);
    a.Expj(SrcOmega);     //call a TVec only method here
}
```

4.6 Function parameters

It is recommended to declare parameters of methods as TMtxVec, TDenseMtxVec, TVec, or TMtx and not of Vector or Matrix type. Regardless if the Vector or Matrix type is passed with a ref parameter or not, they will always behave as if passing an object; by reference. It is also possible to mix TVec/TMtx and Vector/Matrix types in expressions and to copy an array of doubles to Vector:

```
private void Test1(TVec v)
…
private void Test2()
{
    Double[] av;
```

```
    test1((Vector) av)); //this will work, but not by reference
    //..
}
```

The array will be not be passed by reference. Any changes made to v, will not be visible in av. That is because the explicit typecast to Vector created a new Vector variable that copied the data from the array in to the new temporary variable.

4.7 Indexes, ranges and sub-ranges

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

```
public TMtxVec Exp();
public TMtxVec Exp(TMtxVec X);
public TMtxVec Exp(int Index,int Len);
public 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 4.5 on how the size operation is performed).

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

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

More examples:

```
a.Copy(b, 2,0,10);
```

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

```
b.SetSubRange(2,10);
a.Copy(b);
```

Which is the same as:

```
b.SetSubIndex(2,11);
a.Copy(b);
```

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

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

This is the same as:

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

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

```
a.Gather(b,null,TIndexType.indIncrement,2,0);
a.Log10();
a.Exp();
a.Scatter(b,null,TIndexType.indIncrement,2,0);
```

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

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

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

```
a.FindMask(b, "=",c);
```

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

```
a.FindAndGather(b, "<>", Math387.NAN, Indexes);
a.Scale(2);
b.Offset(1);
a.Scatter(b,Indexes);
```

4.8 TVec and TMtx methods as functions

Ideally mathematical expression is written with Vector and Matrix records like this:

a = a*b + b;

For TVec and TMtx 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 “this” or the Self. This allows nesting of calls like in the example above. Syntax like this can result in a memory leak, if TVec is allocated in unmanaged memory:

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

4.9 Complex data

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

```
a.Length = 10;
```

```
a.Complex = true;
```

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

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

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

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

```
public Vector GetSomething()
{
    Vector av = new Vector(10); //vector size
    TCplx ac = "1+2i";  //Convert from string to complex number


    Vector Result = av + ac + 2;  //always by value operations
    // ./  and .*  (not linear algebra)
     return Result;
}
```

4.10 MtxVec types

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

4.10.1 TSample

This type is used everywhere where it is neccessary to declare a floating point number. It can be declared as double or single.

```
{$IFDEF TTDOUBLE}  type TSample = double; {$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, but it can be helpful when reading help files.

4.10.2 TCplx

Declared as:

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

This is the default complex number type used by MtxVec. Many object oriented libraries declare the complex type as an object tpye (public class Complex)). This means that the consecutive elements within the array are then no longer stored at consecutive memory locations which affects performance. TCplx has overloaded all the necessary operators and it is possible to write nearly any expression the same as with real valued numbers.

4.10.3 TSampleArray, TCplxArray

Declared as:

```
Delphi:
TSampleArray = array of TSample; //  double[]
TCplxArray = array of TCplx;      //  TCplx[]
```

5 Accessing values of Vector and Matrix

5.1 Array property access

Example:

```
Matrix am = new Matrix(5, 5);
TMtx am2 = new TMtx();

am.Values[0, 0] = 2;
am2.Values[0, 0] = 3;
```

Array properties allow a clean range checked access. The array properties perform explicit range checking when assertions are enabled (they can be disabled via a compiler switch).

5.2 Handling of 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:

```
TMtx a = new TMtx();
a[1, 0] = 2;
```

instead of:

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

The following access methods are semantically equivalent:

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

6 Memory management

6.1 Introduction

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

```
Matrix am = new Matrix(0,0);

a.Rows = 4; //allocates nothing
a.Cols = 4;  //a now holds 16 elements
a.Rows = 0; // deallocates memory
a.Size(4,4,False);  //same as:  a.Complex = false; a.Rows = 4; a.Cols = 4;
```

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

There are some special issues that need to be taken in to account when working matrices. TMtx/Matrix interfaces highly optimized FORTRAN code. There are two more properties available from TMtx and Matrix:

```
a.Values1D[i]
a.CValues1D[i]
```

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

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

This will access the same matrix element as:

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

or

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

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

```
a.Size(4,4,false,false);
```

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

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

6.2 Using TVec and TMtx with managed and unmanaged memory

TVec and TMtx are always created explicitly. They are never created inside a function and returned as result by MtxVec library or any of its add-on packages. Although some functions return TVec and TMtx these are only methods of TVec and TMtx which return “this”. Furthermore TVec and TMtx can be created in two different ways:

1. From the managed memory where the memory is garbage collected:

```
private void Test()
{
    TVec c = new TVec();
    TVec b = new TVec();

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

2. From unmanaged memory, where the memory has to be freed explicitly, but giving zero pressure on the garbage collector logic.

```
private void Test()
{
    TVec a;
    TVec b;

    MtxVec.CreateIt(out a, out b);
    try
    {
        b.Size(1000);
        b.RandUniform(0.5, 1.5);
        Probabilities.BoseEinsteinPDF(b, 0.3, 0.1, c);
    }
    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

(unmanaged) 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. In compare to the garbage collector, it can sometimes deliver speed gains of 5x or more because it increases memory reuse, and thus effectiveness of the CPU cache. The memory allocated by CreateIt is never moved. You can freely mix operations between TVec/TMtx and Vector/Matrix objects allocating unmanaged or managed memory without performance penalties.

Matrix and Vector types can be created implicitly and internally to procedures and functions which provides a more programmer friendly user interface, but at the same time, the programmer looses the control over the memory allocation and its performance related cost.

```
private void Test()
{
    Vector c = new Vector(0);  //note that Vector(0) constructor is called (!)

    b = MtxExpr.RandUniform(1000, 0.5, 1.5);
    Probabilities.BoseEinsteinPDF(b, 0.3, 0.1, c);
}
```

This very fine level of memory allocation control can be very beneficial when there is a need to optimize the code for speed (See Benchmarks chapter).

6.3 In-place/not-in-place operations

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

7 Range checking

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 checks the indexes i and j to be within the bounds
```

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 and higher performance can be obtained.

8 Why and how NAN and INF

NAN is short for Not a Number and INF is short for infinity. CLR does not raise an exception when a division by zero occurs or an invalid floating operation is performed. 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 class static routines for string to number and number to string conversions in Math387 class (StrToVal ,StrToCplx, FormatCplx, FormatSample, StrToSample, SampleToStr) and drawing routines in MtxVecTee class (DrawValues, DrawIt) capable of handling NAN and INF values. By using those routines, the user will avoid most of the problems when working with NAN and INF values. StrToSample for example will convert a NAN or INF string to its floating point presentation:

```
double a = 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:

```
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:

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

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

```
if (a.Find(Math387.Nan) > 0)  ..//test if "a" vector holds a NAN value
```

9 Serializing and streaming

9.1 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

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 form TMtxComponent have the capability to store their states (properties) to the stream, file or to assign from another object of the same type.

The "template" routines set a 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.

Important: With initial release MtxVec v3.1 for VS.NET these methods of TMtxComponent are not functional.

9.2 Streaming of TVec, TMtx and TSparseMtx

All TVec/TMtx and Vector/Matrix types are marked as [Serializable]. 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 and will save and load data only 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:

9.3 Write TMtx/Matrix to a text file

```
Matrix amtx = new Matrix(20,20,true);
amtx.RandUniform(-1,2);
TStringList StringList = new TStringList();
amtx.ValuesToStrings(StringList, "\t","",""); // use tab = (char) 9 as delimiter
StringList.SaveToFile("C:\\ASCIIMtx.txt"); // Save matrix values to txt file
```

9.4 Read TMtx/Matrix from a text file

```
Matrix amtx = new Matrix(0,0);
TStringList StringList = new TStringList();
StringList.LoadFromFile("C:\\ASCIIMtx.txt"); // Load matrix values from txt file
amtx.StringsToValues(StringList, "\t"); // use tab = (char) 9 as delimiter
MtxVecEdit.ViewValues(amtx);
```

10 Input-output interface

10.1 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 System.IO.Stream 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:

```
enum TPrecision {
                prDouble, prSingle, prInteger, prCardinal,
                prSmallInt, prWord, prShortInt,
                prByte, prMuLaw, prALaw, prInt24
                }
```

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

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

10.2 Formatting floating point values – FloatToString

Math387 class declares the following routines:

```
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:

```
public string CplxToStr(TCplx Z, int Digits);
```

```
public string SampleToStr(double X, int Digits, int Precision);
```

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

```
double a;
string myString;

a = 12.123456;
myString = Math387.SampleToStr(a, 0, 3); // myString = '12.1';

a = 12.123456;
myString = Math387.SampleToStr(a, 0, 15); // myString = '12.123456';
```

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

```
public TCplx StrToCplx(string Source);
public double StrToSample(string Source);
```

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

10.3 Displaying the contents of the TVec and TMtx

10.3.1 Displaying data as a delimited text

Vector and Matrix have two methods for getting and setting their values to and from a text file. They are called StringsToValues and ValuesToStrings. You can use Dew.Math.TStringList class together with Vector or Matrix ValuesToStrings method to write Vector or Matrix values to Dew.Math.TStringList string array:

```
using Dew.Math;
using Dew.Math.Controls;

TStringList strarray = new TStringList();

Matrix aMtx = new Matrix(3, 5);
aMtx.RandGauss();
aMtx.ValuesToStrings(strarray,"\t", "
      0.#####;-0.#####", "+0.#####i;-0.#####i");
```

You can also write values to single text line:

```
aMtx.Size(5, 3, true);
aMtx.RandGauss(3, 1);
string txt = aMtx.ValuesToText("\t", "
      0.#####;-0.#####", "+0.#####i;-0.#####i");
```

In both cases the methods support:

- handling of complex and real values,
- accept NAN and INF values,
- can get/set only a sub vector or a sub matrix,
- control the displayed precision.

10.3.2 Displaying data within a grid.

The best way to display the contents of a matrix is within a DataGridView control. For this you can use Dew.Math.GridServices ValuesToGrid, GridToValues and PrepareGrid routines OR Dew.Math.Controls.MtxGridView control.

For small and medium object size (few fundred matrix columns, rows or vector elements) the preffered way is to use MtxGridView control:

```
using Dew.Math;
using Dew.Math.Controls;

// create complex matrix
Matrix m= new Matrix(20, 5, true);
m.RandGauss();

// Create MtxGridView and connect it to Matrix m
MtxGridView mtxGridView1 = new MtxGridView();
mtxGridView1.DataObject = m;
```

MtxGridView control allows you easy viewing/editing of real or complex matrices or vectors. The control is derived from DataGridView, inherits all it's properties/methods and introduces full support for complex numbers. This means that any changes to MtxGridView cell values are automatically propagated to Matrix and Values.

Another way to connect values to DataGridView is to use Dew.Math.GridServices routines. This approach offers even more customization, but it lacks the automation MtxGridView has built-in. The following code copies all values from matrix to DataGridView:

```
using Dew.Math;
using Dew.Math.Controls;

// create complex matrix
Matrix m= new Matrix(20, 5, true);
m.RandGauss();

// Grid should be 20 rows x 10 columns, two colums per complex number
GridServices.PrepareGrid(dataGrid1,20,10);
GridServices.ValuesToGrid(m,dataGrid1,0,0);
```

In this case changes in dataGrid1 are not automatically updated to Matrix m Values. This has to be done by using GridServices.GridToValues routine:

```
using Dew.Math;
using Dew.Math.Controls;

Matrix m;

// Grid is 20 rows x 10 columns, two colums per complex number,
// no headers
GridServices.GridToValues(dataGrid1,m, true, false);
```

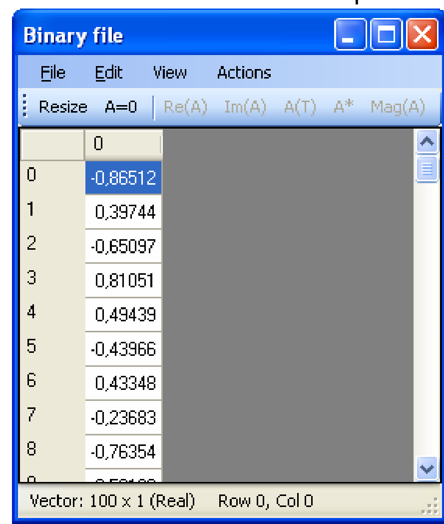
Check Dew Math demo and help files for differences between two approaches.

10.3.3 Viewing data by using built-in Values editor

Dew.Math.Editors.MtxVecEdit class provides several routines which allow easy viewing/editing of matrix/vector values. By calling the Dew.Math.Editors.MtxVecEdit.ViewValues routine a simple editor will be displayed allowing the user to examine the values of a Vector or Matrix object.

```
using Dew.Math;
using Dew.Math.Editors;

Vector v = new Vector(100);
v.RandGass();
MtxVecEdit.ViewValues(v)
```



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 **View** menu is checked.

Editor status bar shows basic object info shown, in this case real vector of size 100x1 elements. Header columns and header rows show object cell indices, starting with 0. Additional information in status bar shows which cell is currently being selected and/or edited (in this case first vector element is being selected). Row and column indices can be shown or hidden by using **Row Index** and **Column Index** menu items in the **View** menu.

If object values are complex numbers, then header columns labels are formatted according to **Split Complex Numbers** menu item value from the **View** menu. If selected, column header labels are named as "0-Re", "0-Im", "1-Re", "1-Im", ..., where 0-Re shows the real part of the first cell complex number, "0-Im" shows the imaginary part of the first complex number, etc... If not selected then each column shows properly formatted object cell complex number.

10.4 Charting and drawing

Vector or matrix values can also be drawn directly on the chart by calling the TeeChart.DrawIt routine. This class is located in the Dew.Math.Tee namespace. TeeChart class contains a large set of class static DrawValues routines. This routines copy data from TVec or TMtx to the defined TChartSeries. Adding of new values is optimized for the TeeChart version used and charting can be considerably faster, if DrawValues is used. DrawValues routines also take care of any NAN's and INF's.

```
Vector a;

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

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).

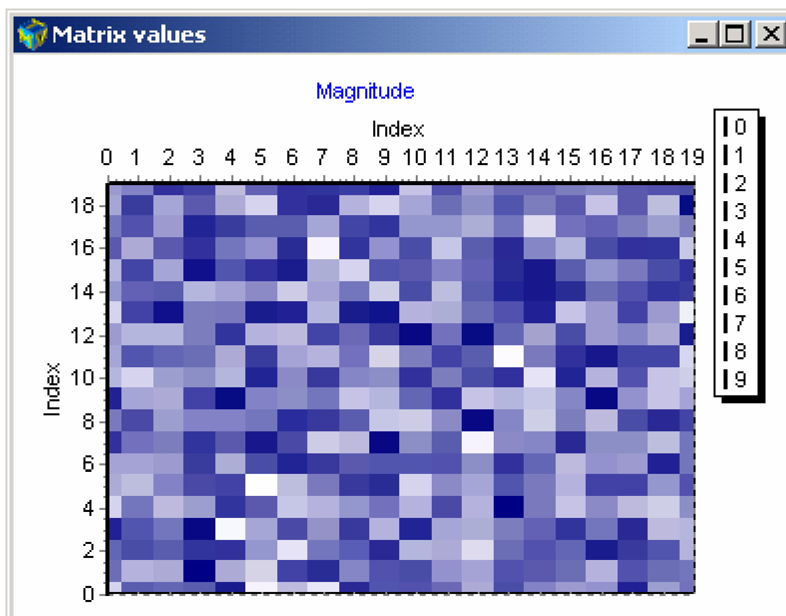


Figure 3

11 Programming style

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

11.1 Mixing TVec/TMtx and Matrix/Vector types.

Handle Vector/Matrix as objects even though they are of struct type. 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.

11.2 Try-finally blocks and unmanaged memory

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

```
TVec a1;
TVec b1;

MtxVec.CreateIt(out a1,out b1); //work vectors
try
{
        //…
}
finally
{
    MtxVec.FreeIt(ref a1, ref b1);
}
```

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.

Do not write code like this:

```
MtxVec.CreateIt(out b);

//...some code here

MtxVec.CreateIt(out a);
yourProc(a,b);
MtxVec.FreeIt(ref b);

//.. some code here..
MtxVec.FreeIt(ref a);
```

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

11.3 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:

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

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

11.3.1 Invalid parameter passed:

```
{
    if (a.Length != b.Length) throw new Exception("a.Length <> b.Length");
      //…
}
```

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 = null" and an OK button.

11.4 When to use CreateIt/FreeIt

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/FreeIt 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.
- Within garbage collected environment it is best if CreateIt/FreeIt pairs used within the same function.

12 Getting up to speed – the benchmarks

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:

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

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

```
a.Length := 2000;
b.Length := 2000;
for (i = 0; i < 500; i++) do
{
    YourFunc(a,b,c1,c2,ea,eb);
}

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

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

    if (a.Length != b.Length) throw new Exception("a.Length <> b.Length");
    MtxVec.CreateIt(out a1,out 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
    {
        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:

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

        xx = x*x;
        return  Math.Sqrt(4*a*Math387.INVTWOPI)*a*xx*Math.Exp(-0.5*a*xx);
}
```

Vectorized with expressions:

```
public Vector MaxwellPDF(Vector x, Vector a)
{
        Vector xx = x * x;
        Vector res = Math.Sqrt(4 * Math387.INVTWOPI * a) *
                     a * xx * MtxExpr.Exp(-0.5 * a * xx);
         return res;
}
```

Vectorized function:

```
private void MaxwellPDF(TVec X, double a, TVec Res)
{
    TVec res1;
    MtxVec.CreateIt(out res1);
    try
    {
        res1.Sqr(X);
```

```
        res.Mul(res1, -0.5 * a);
        res.Exp();
        res.Mul(res1);
        res.Mul(Math.Sqrt(4 * Math387.INVTWOPI * a) * a);
    }
    finally
    {
        MtxVec.FreeIt(ref res1);
    }
}
```

Block vectorized function:

```
private void MaxwellPDF(TVec X, double a, TVec Res)
{
    TVec res1;
    MtxVec.CreateIt(out res1);
    try
    {
        Res.Size(X);
        X.BlockInit();
        res.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 * a * Math387.INVTWOPI) * a);
            res.BlockNext();
            X.BlockNext();
        }
    }
    finally
    {
        MtxVec.FreeIt(ref res1);
    }
}
```

On Q6600 (2.4 GHz, Quad Core 2 CPU) the vectorized function is about 8x faster than the plain function (Figure 4). 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.

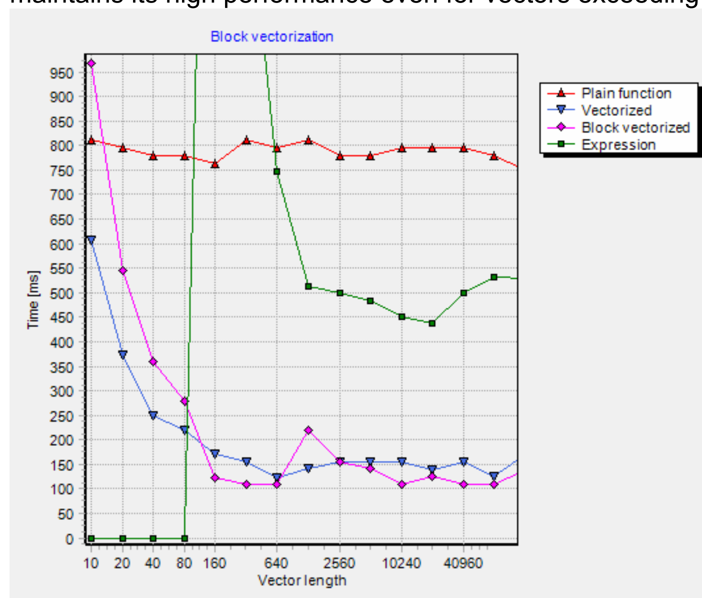


Figure 4 Benchmark

The block vectorized function is only marginally faster than vectorized version due to the use of SSE2/SSE3 instructions. If the CPU does not support SSE, then the gain of the block vectorized version will be much more significant (typical gains are about 6 times). For example, when using older CPU's the speed of the plain function for vectors with length larger than the size of the CPU cache will be higher than that of its vectorized version. The vectorized version has to access memory multiple times, while the plain function version can cache some intermediate results in to FPU registers or CPU cache. The block vectorized version will ensure that the chunk of the vector being processed can fit in to the CPU cache and will thus give optimal performance for long vectors even in that case. The green line shows the behavior of the Vector/Math expressions. Below 640 elements the plain function is much faster, however still 4x slower than vectorized version, which uses CreateIt and FreeIt to allocate memory and does not use vectorized math expressions (only TVec and TMtx).

12.3 Garbage Collector and High Performance numerical math

To measure the effect of the garbage collector on the high performance numerical algorithms we designed a special test and compared impact on both the classical not vectorized functions and on vectorized functions. In both cases, the computational part of the timed loop and the part where the memory was allocated were separate and distinct code parts:

1. No memory was allocated inside the timed loop that was used by the computational part. In fact, the computational part was not allocating nor releasing any managed memory inside the timed loop.
2. All memory allocated in the computation part was non movable (GC pinned or unmanaged)
3. The GC memory allocation part of the algorithm was not doing any computation.

Such a setup can show how would performance of a numerical algorithm, be affected by memory allocations in the same or some other part of the program.

The amount of memory allocated is a parameter to the benchmark. Figure 5 is showing the results and the code for the vectorized version is listed below. The X axis shows the length of the input vector X and the Y axis is time normalized per element. The absolute cost of the garbage collector in our case for arrays longer than 1024 elements is approximately 200-250ms per element. (The complete source code is part of the MtxVec demo under “Memory allocation->Garbage collector benchmark”.)

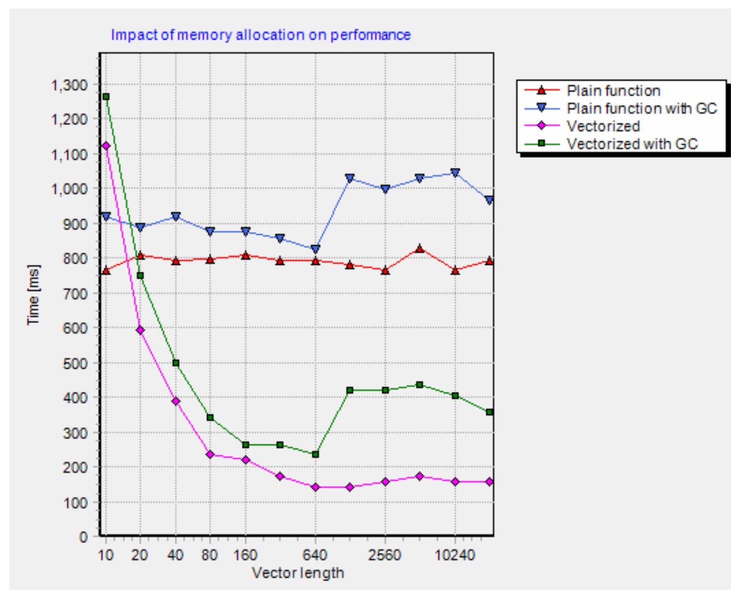


Figure 5 Garbage collector cost for GCIterCount = 5 (see source code)

```
private void MaxwellPdf1(TVec X, double a, TVec Res)
{
    TVec res1;
    MtxVec.CreateIt(out res1);
    try
    {
        res1.Sqr(X);
        res.Mul(res1, -0.5 * a);
        res.Exp();
        res.Mul(res1);
        res.Mul(Math.Sqrt(4 * Math387.INVTWOPI * a) * a);
    }
    finally
    {
        MtxVec.FreeIt(ref res1);
    }
}
```

```
private double MaxwellNoBlock(int Iterations) {
    double a = 1;
    double[] testArray;
       int counter = Environment.TickCount;
       for (int i=0;i<Iterations;i++) {

        MaxwellPdf1(x, a, res);

        for (int k = 0; k < GCIterCount; k++) //GC Loop
         {
              testArray = new double[res.Length];
              testArray[1] = resArray[1];
         }
       }
       return Environment.TickCount - counter;
}
```

An analysis about the type slowdown gives an answer, if the slowdown occurred due to the garbage collector execution cost, or was the numerical part of the code somehow affected the garbage collector like invalidation of the CPU cache.

If the slowdown was purely due to the garbage collector execution cost, then the loop executing without the computational part should be showing the same absolute values for the garbage collector as we measured the first time (Figure 5, 200-250ms, the delta between the pink and green line above for vectors longer than 1000 elements). The results can be seen on Figure 6. When comparing to Figure 5 we can see that slowdown is caused solely by the execution of the GC.

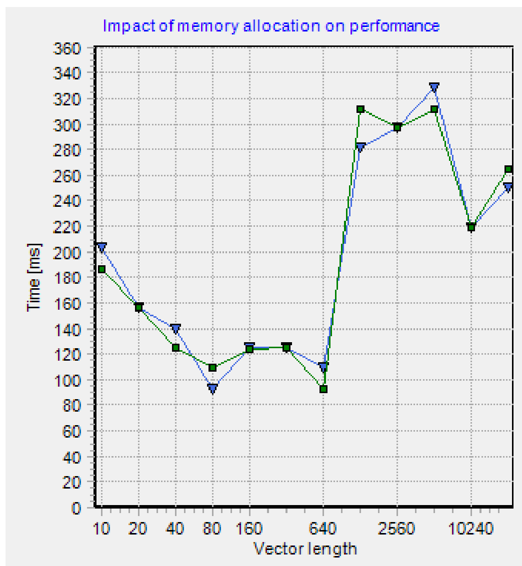


Figure 6 Garbage collector only processing times

Lessons learned: Managed memory should be allocated with care inside tight high performance loops because the processing time rises considerably with the amount of memory allocated.

The code that needs really high performance, should pre-allocate memory before entering the loop; when inside the loop it should allocate minimum amount of arrays and use CreateIt/FreeIt where possible. From Figure 4 we have also learned, that for vectors shorter than about 600 elements, we should be carefull **not** to allocate them repeatedly inside tight loops for best performance:

```
for (Int32 j = 0; j < 10000000; j++)
{
    TVec a = new TVec();  //don't do this
       //..
    a.Size(20);      //this is OK. Calling a.Size does not reallocate if same size
}
```

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 SSE2/SSE3/SSE4.1 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 class includes a utility function called FixAngle which should be called on the argument before it is passed to the sine function, but only if a “large” argument is to be expected. This will fix the accuracy and the speed problem. Of course, if the argument is not large, the speed will decrease and the accuracy will not be improved.
6. There are penalties on processing NAN and INF. Make sure those are fished out from the vectors soon after they might occur, to lower the performance cost on the follow up code.
7. For CPU’s without SSE, the only way to improve the performance is to strictly follow the rules of block processing. (CPU cache size).
8. When running “quick” benchmark tests make sure to pass “valid” parameters to functions. If the function is not defined in a region, the result of the function will be a NAN or INF. All subsequent functions that will receive NAN or INF at the input could run much slower. This is the limitation of the SSE instructions. The program must be guarded against such cases explicitly. (Either by checking the user input or by inserting checks in the code that will abort the algorithm sooner if a NAN or INF is detected.). This can be important when running algorithms that already take a long time to compute with valid data.
9. Intel also warns about denormals. They are another cause for slowdown. Denormals are numbers which get truncated due to limited floating point number range. So again, the input to the algorithm when testing it, should be valid data.

Vectorized expressions specifics:

10. Your compiler may not support a specific optimization called: collection on common sub expression. If your expression contains the same expression multiple times, be sure to assign its result to a temporary variable to prevent the expression from being evaluated multiple times.
11. Use parenthesis to indicate which part of the expression should be evaluated first. There is no optimization analysis that would be based on precedence of the same operator. Namely, multiplying two vectors or multiplying two real numbers is very different in terms of clock cycles used. But the compiler does not know that.

Garbage collector specifics:

12. The processing cost of the garbage collector can be high. MtxVec will deliver best performance when used in accordance with design guidelines for GC.

12.5 Code vectorization methods

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

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

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

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

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

     if fLength = 0 then Exit;


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

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

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

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

    GCOperation(Base);
end;
```

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

12.6 Enabling the expressions for Multi-core CPU’s

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

```
public Vector MaxwellPDF(Vector x, Vector a)
{
       Vector xx = x * x;
       Vector res = Math.Sqrt(4 * Math387.INVTWOPI * a) *
                       a * xx * MtxExpr.Exp(-0.5 * a * xx);
        return res;
}
```

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

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

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

12.7 Intel complex number VML functions.

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

13 Debugging MtxVec

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

The following watches can be specified for variables of Vector (TVec) or Matrix (TMtx) type:

- aVector.RealValues
- aVector.ComplexValues
- aMatrix.RealValues
- aMatrix.ComplexValues

When the debugger watches (Figure 7) are no longer sufficient, it is possible to invoke the debugger visualizer for TVec/TMtx, Matrix and Vector types (Figure 8, Figure 9) by pressing on the magnifying glass icon.

Watch 1

Name	Value	Type
av.RealValues	"Len = 5, False, 0, 1, 2, 3, 4, "	string
av.Values[2]	2.0	double
av.ComplexValues	'av.ComplexValues' threw an exception of type 'Dew.Math.EMtxVec	string {De
am.RealValues	"5x5, False, 0.059685, -0.14196, 0.65631, 0.201676, 0.4948	string

Figure 7 Inspecting contents of Vector "av" and Matrix "am"

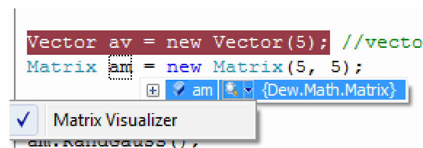


Figure 8 Starting matrix visualizer during debugging.

Matrix values

File Edit View Actions

Resize A=0 Re(A) Im(A) A(T) A* Mag(A)

	0	1	2	3	4
0	0.059684867843829738	-0.14195999171311197	0.65630999036661608	0.20167604233462297	0.49486149210574876
1	-0.41855413232607491	-0.061588493682402969	-1.2351269158196159	-0.0041077685400349581	0.040596532650035246
2	0.25794282584526745	-0.40079528029735256	0.96389919336591412	1.8251460941332778	-0.71878306047260909
3	-1.5761995987994923	0.21741727199683852	0.81602285386036111	-0.051279269334096038	0.0020048082033720009
4	0.31216179781025949	-2.5100556612343832	0.025795532105576679	-0.52507916897392093	0.1349105781390269

Matrix: 5 x 5 (Real) Row 0, Col 0

Figure 9 Displayed matrix contents while debugging

13.2 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 class. Report is of a TMtxVecReport type and encapsulates StringBuilder class. It has been extended with several new methods to support:

1. Saving the stream to system.IO.Stream 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:

```
Vector av = new Vector(5); //vector size
Matrix am = new Matrix(5, 5);

av.Ramp();
am.RandGauss();

MtxVec.Report.Print(new TMtxVec[2] { av, am }, new string[2] { "av
vector","am matrix" });
MtxVec.Report.NewLine();
MtxVec.Report.SaveToFile("C:\\TestReport.txt",false);
MtxVec.Report.Clear();
```

The printout will look this:

```
av vector (5x1) =

 0.0000
 1.0000
 2.0000
 3.0000
 4.0000

am matrix (5x5) =

-1.4301       -1.7811        0.7858       -0.3901        1.7490
 0.8135        0.1826       -1.2526        0.7739       -0.3879
 0.4901        0.3925       -0.9281        0.1107        0.4650
 0.0928       -0.3188        1.2720       -0.3946        1.0211
 0.3266       -0.0072       -0.6555       -0.2365        1.5490
```

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

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

13.3 Calling DrawIt and ViewValues

Of course you can also inspect the contents of objects with the functions provided:

```
Vector av = new Vector(5); //vector size
Matrix am = new Matrix(5, 5);

av.Ramp();
am.RandGauss();

MtxVecEdit.ViewValues(av, "Vector values", true);
TeeChart.DrawIt(av, "Vector values", false);
MtxVecEdit.ViewValues(am, "Matrix values", true);
TeeChart.DrawIt(am, "Matrix values");
```

13.4 Memory leaks

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

When using unmanaged memory with TVec and TMtx we have to always match CreateIt/FreeIt pairs, as already emphasized. MtxVec class 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.

The memory preallocation is disabled by calling:

```
MtxVec.Controller.SetMtxCacheSize(0, 0);
MtxVec.Controller.SetVecCacheSize(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.

13.5 Memory overwrites

When using TVec and TMtx or Vector and Matrix memory overwrites should be a thing of the past. But if you chose to work directly with unmanaged 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. These mechanisms however don't work in the following case:

```
Vector av = new Vector(5); //show the vector editor
Matrix am = new Matrix(5, 5);

av.Ramp();
am.RandGauss();
av.SetSubRange(am); //am may be freed before we stop using av
```

The "av" object in this case has only absolute memory address to the memory inside of am. This information can be passed to the garbage collector by calling KeepAlive method at the point where "am" memory referenced by "av", is truly no longer required. This rule should be followed for all cases, when we make use of PValues style functions which return IntPtr type absolute address to the memory location.

```
using System.Runtime.InteropServices;

Vector av = new Vector(5); //show the vector editor
Matrix am = new Matrix(5, 5);

av.Ramp();
am.RandGauss();
av.SetSubRange(am); //am may be freed before we stop using av

//…
//computation on av only.
//…

GC.KeepAlive(am);
```

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 windows\system32 directory:

- MtxVec.lapack2d.dll. This library is mandatory and contains lapack functions.
- MtxVec.spld2.dll. This library is mandatory and contains many vectorized math functions.
- libguide40.dll. This library is mandatory and contains OPENMP runtime used by other multithreaded dlls.

On 64bit OS's, the system32 directory is replaced by SysWOW64.

14.1 Compact MtxVec

Compact MtxVec is an initiative to allow the customer to provide additional processing muscles only where needed, and at the same time keep the distribution size as low as possible:

- MtxVec.sparse2d.dll. If the application uses sparse.pas, this library is also required. There is no fall back code.
- MtxVec.Random.dll is used by the random generators located in RndGenerators class. The random generators are threaded and vectorized.
- MtxVec.Vmld.dll contains vectorized and threaded math functions like sin, cos, exp, etc…
- MtxVec.FFT.dll. This library contains multithreaded 1D, 2D and 3D DFT and FFT functions.

If distribution size is a problem, we can make a build of custom size dll's for your specific application. The provided dlls have specialized code of each function for a general purpose Pentium compatible CPU, for P4 SSE2 instruction set and for P4 SSE3/SSE4.1 instruction set. The appropriate code version is selected automatically, when the libraries are loaded.

14.2 Managing the threads

It is possible to control the maximum number of threads used by MtxVec by setting the environment variable: set OMP_NUM_THREADS=2 (for two cores). If this variable is not set, MtxVec will use the maximum number of threads that make sense for each specific algorithm given the available CPU core count. If you want to implement threading in your own code, the threading in MtxVec should be disabled.

Every process running on Windows OS has a priority defined that can also be examined and set from the Task Manager. This priority affects the relative priority of all the threads that this process has started. This gives an additional control over the core usage of multiple applications running on a multi-core machine.

15 Major function groups

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

SSE2/SSE3/SS4.1 – supports 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 usefull 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	$a[i] = \lvert a[i] \rvert$ $a[i] = \lvert b[i] \rvert$	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] = ArcCos(a[i]) a[i] = ArcCos(b[i])	SSE2/SSE3, SMP
ArcCosh	TMtxVec	a[i] = ArcCosh(a[i]) a[i] = ArcCosh(b[i])	SSE2/SSE3, SMP
ArcCot	TMtxVec	a[i] = ArcCot(a[i]) a[i] = ArcCot(b[i])	SSE2/SSE3, SMP
ArcCoth	TMtxVec	a[i] = ArcCoth(a[i]) a[i] = ArcCoth(b[i])	SSE2/SSE3, SMP
ArcCsc	TMtxVec	a[i] = ArcCsc(a[i]) a[i] = ArcCscs(b[i])	SSE2/SSE3, SMP
ArcCsch	TMtxVec	a[i] = ArcCsch(a[i]) a[i] = ArcCsch(b[i])	SSE2/SSE3, SMP
ArcSec	TMtxVec	a[i] = ArcSec(a[i]) a[i] = ArcSec(b[i])	SSE2/SSE3, SMP
ArcSech	TMtxVec	a[i] = ArcSech(a[i]) a[i] = ArcSech(b[i])	SSE2/SSE3, SMP
ArcSin	TMtxVec	a[i] = ArcSin(a[i]) a[i] = ArcSin(b[i])	SSE2/SSE3, SMP
ArcSinh	TMtxVec	a[i] = ArcSinh(a[i]) a[i] = ArcSinh(b[i])	SSE2/SSE3, SMP
ArcTan	TMtxVec	a[i] = ArcTan(a[i]) a[i] = ArcTan(b[i])	SSE2/SSE3, SMP
ArcTan2	TMtxVec	a[i] = ArcTan2(b[i],c[i])	SSE2/SSE3, SMP
ArcTanh	TMtxVec	a[i] = ArcTanh(a[i]) a[i] = ArcTanh(b[i])	SSE2/SSE3, SMP
Cbrt	TMtxVec	$a[i] = (a[i])^{1/3}$ $a[i] = (b[i])^{1/3}$	SSE2/SSE3, SMP
Copy	TMtxVec	a[i] = b[i]	SSE2/SSE3
Ceil	TmtxVec	a[i] = Cos(a[i]) a[i] = Cos(b[i])	SSE2/SSE3, SMP
Cos	TMtxVec	a[i] = Cos(a[i]) a[i] = Cos(b[i])	SSE2/SSE3, SMP
Cosh	TMtxVec	a[i] = Cosh(a[i]) a[i] = Cosh(b[i])	SSE2/SSE3, SMP
Cot	TMtxVec	a[i] = Cot(a[i])	SSE2/SSE3, SMP

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

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

15.2 Statistical

Kurtosis	TDenseMtxVec	a[i] = Kurtosis(a[i]) a[i] = Kurtosis (b[i])	SSE2/SSE3
Skewness	TDenseMtxVec	a[i] = Skewness(a[i]) a[i] = Skewness(b[i])	SSE2/SSE3
StdDev	TDenseMtxVec	a[i] = StdDev(a[i]) a[i] = StdDev (b[i])	SSE2/SSE3
RMS	TDenseMtxVec	a[i] = RMS(a[i]) a[i] = RMS(b[i])	SSE2/SSE3
Mean	TDenseMtxVec	S = Mean(a[i])	SSE2/SSE3

15.3 Complex number specific

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

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

15.4 Size, streaming and storage

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

15.5 FFT's

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

15.6 Linear algebra

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

MtxPower	TMtx	A^p	SSE2/SSE3, SMP
MtxIntPower	TMtx	A^i	SSE2/SSE3
TensorProd	TMtx	aMtx = alfa*bMtx*Vec aMtx = alfa*Vec*Mtx aMtx = Vec1 x Vec2	SSE2/SSE3
AddTensorProd	TMtx	Mtx = alfa*Vec1 x 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