MtxVec v6.0

Users Guide to MtxVec for Delphi W32/W64 and Firemonkey

MtxVec version v6, rev 3.1

www.dewresearch.com

1 Introduction

MtxVec numerical library has two main purposes:

1.) Provide framework, which allows cost-efficient development of algorithms that run fast.
2.) Deliver a wide range of ready-to-use algorithms, which execute at impressive speeds out-of-the-box.

This user’s guide deals primarily with the intricates of the programing framework. For the list of available ready to use algorithms, please look at the library reference available here:

https://www.dewresearch.com/Help/Delphi/MtxVec/contents.html

and search the content by Index.

2 The framework

By combining all the framework features of the library, it is possible to write fast executing code with minimum effort. To some extend it can be assumed, that if an algorithm can be speeded up, it can be done with this library and that the result will not be much less than the optimal code path in the absolute sense.

For the numerical code to run fast on modern CPUs, the concepts built in to the library address the following:

1.) Computers main memory is a shared resource. When it comes to multi-threading, any shared resource becomes a bottleneck as long as it is shared.
2.) Each CPU core has a dedicated memory called Level 1 cache. If the code is working only with Level 1 cache, there is no resource sharing and the speed can scale linearly with core count.
3.) Only Level 1 cache is fast enough to feed the CPU, when the code is using AVX2 and AVX512 instructions sets. These instructions allow for the fastest possible code.

2.1 Reduce memory sharing issues

Memory sharing as a problem grows proportionally with the frequency of memory allocation requests. By reducing the frequency of memory allocation calls, we are also reducing this bottleneck and allow the code to scale better with core count. To reduce this frequency, MtxVec implements for its key objects (TVec/TMtx/TVecInt/TMtxInt) thread-locking free features:

a.) Capacity property. Each time the object is sized below the size of the Capacity, a memory allocation call is saved.
b.) Object cache. A small set of key objects is pre-allocated as a cache in a global structure. Each thread gets its own thread-local cache. Each object comes with pre-allocated memory. It suffices to fetch the object to also get (a preset) amount of ready-to-use memory.
c.) Subranges. It is possible to segment or subrange the memory of key objects. This can be done in-place, not-place and the Subrange calls can be also nested. Matrix memory can be reinterpreted as a vector and vice versa.
d.) For the most part, the library does not create objects or allocate any memory on the heap behind the scenes. If some working memory is needed, this is fetched from the object cache.

2.2 Simplifying the use of CPU cache L1

CPU Cache L1 is used, when:

a.) The same continuous block of memory is accessed more than once.

b.) The total size of all blocks accessed does not exceed the size of CPU cache L1 for 90% of the time. Typical CPU L1 cache size is 32KB.

This is dealt with in two ways:

a.) Because the key objects (TVec/TMtx/TVecInt/TMtxInt) obtain their memory from the object cache, this results in reusing of the memory blocks at the same physical memory address. Typically, about 5-6 objects can be used within a tight loop not to exceed the size of the CPU's L1 cache.

b.) Additionally, MtxVec implements BlockInit, BlockNext and BlockEnd methods for its key objects. This pattern simplifies sizing of the memory within tight loops while keeping the memory usage below CPU L1 cache size.

2.3 Take advantage of the latest CPU instruction sets

The library implements a comprehensive set of "vectorized" math functions like Sin, Cos, Exp, ... Their speed of execution is typically about 15x higher with AVX512 and 8x higher with AVX2 than Delphi built-in Math functions. Nearly all methods of key objects (TVec/TMtx/TVecInt/TMtxInt), which work on array of elements, are vectorized (hardware accelerated). MtxVec will run on all Intel x86 compatible CPU's old and new, but will achieve highest performance on latest CPU generation.

2.4 Reducing the cost of code development

MtxVec was designed with simplicity and ease of use in mind:

a.) Object oriented numerical library.
b.) Improved compactness and readability of code when using vectors versus scalars.
c.) Shallow depth of methods/function calls. Simple functions with clear purpose.
d.) Protects the developer from a wide range of possible errors.
e.) Natural math expression syntax for vectors and matrices with full support for operator overloading for vectors, matrices and complex numbers.
f.) Parameters are explicitly range checked, before they are passed to the dll routines. This ensures that all dll calls are safe to use by protecting the programmer from expensive-to-debug, memory-overwrite errors.
g.) All MtxVec functions must pass strict automated tests. It is these tests, which give the library the highest possible level of reliability, accuracy and error protection.
h.) Many routines are multi-threaded, including 1D FFT, sparse matrix solvers, matrix multiply, and large parts of Lapack.

2.5 LAPACK and BLAS library

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 in two flavors:

1.) On Windows and Linux a hardware accelerated version provided by Intel with their Math Kernel library.
2.) A version written entirely in Delphi pascal for maximum portability available on all platforms.

LAPACK functions can 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. When calling Lapack routines, MtxVec also automatically compensates for the fact that in FORTRAN the matrices are stored by columns and in other languages by rows.

3 Organization

MtxVec library is organized in to three levels - computational level, objects level and component level. The interface to the computational level is a set of functions, which are declared in nmkl.pas, ippspl.pas, and other files and implemented as external DLLs - MtxVec.Lapackd.X.Y.Z.dll, MtxVec.Spld.X.Y.Z.dll, MtxVec.FFT.X.Y.Z.dll, MtxVec.Random.X.Y.Z.dll, MtxVec.Vml.X.Y.Z. and MtxVec.Vmld.X.Y.Z.dll. The first level is not documented. The second level is written in Delphi. This level introduces vector and matrix objects, complex numbers and a number of utility functions delcared in: MtxVec.pas, Polynoms.pas, Math387.pas, Probabilities.pas, SpecialFuncs.pas, Optimization.pas, Toeplitz.pas, Sparse.pas and MtxVecTee.pas. The second level is the “run-time” part of the MtxVec library. Third level is formed by a set of components which are built 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 guide concentrates mostly on the second level, specifically the MtxVec.pas and Math387.pas units and touches some features from MtxVecTee.pas. It gives a good overview on the concept and core features of MtxVec.

3.1 Compiler support

MtxVec v6 supports the following compilers:

	Versions
Embarcadero/CodeGear Delphi W32	Versions 4 and 5 are supported with MtxVec 1.5. Delphi 6, 7 and 2005 are supported by MtxVec 2.0 till 3.52. Delphi 2006 is supported until MtxVec 4.3. MtxVec v6 supports Delphi versions from including Delphi 2007. Mixed single and double precision is supported since v2009. Combined single and double precision is possible since v6. 64bit support requires at least Delphi XE2.
Embarcadero FireMonkey	Cross platform support for Android, iOS and OSx is available from MtxVec v6 and requires latest version of Delphi Berlin.
CodeGear Delphi VCL.NET	Delphi 2005, Delphi 2006, Delphi 2007
CodeGear Delphi Winforms (.NET)	Delphi 2005, Delphi 2006, Delphi 2007
CodeGear C++ Builder	Version 6 with MtxVec v2.1, BDS 2006, BCB 2007/2009 with v3. MtxVec from including v4 requires at least BCB 2006.
Visual Studio C# (VB.NET, C++)	2015.NET, 2017.NET, 2019.NET with .NET Core 5.0 and newer.

3.2 Windows OS and CPU Support

Minimum supported Windows versions are Windows 7 and Windows Server 2008 R2 (32bit and 64bit). Either AMD or Intel CPU with at least SSE3 instruction set is required for MtxVec v6. To run on older machines and OS’s, older version of MtxVec can be used.

4 Quick start

```
uses MtxExpr, Math387;

var am,bm: Matrix;
    av: Vector;
    ac: TCplx;
    arr: TDoubleArray;
    i: integer;
begin
    ac := '1+2i';  //Convert from string to complex number
    am := RandGauss(5,5, mvDoubleComplex); //5x5 complex matrix with Gaussian noise
    av := Ramp(25, mvDouble);    //real vector = [0, 1, 2,...., 23, 24]

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

    //of course you can mix matrices and vectors

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

    for i := 0 to am.Rows-1 do  //standard loop example for complex values
    begin
         av.CValues[i] := am.CValues[i,0]*av.CValues[i];
    end;

    bm := [ [1, 2.1], [3, 4] ] // initializes 2x2 matrix and assigns values.
                                //this syntax requires Delphi XE7 or newer

end;
```

5 MtxVec programming interface

5.1 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:

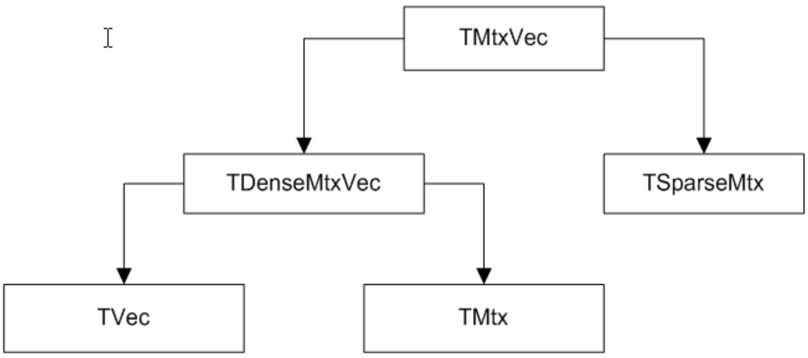


Figure 1 MtxVec class hierarchy

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

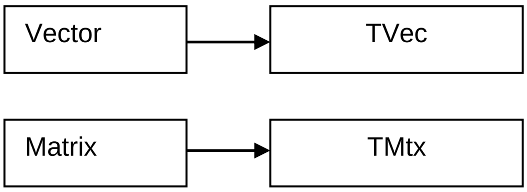


Figure 2 Mapping to records with methods.

All methods and properties of TVec and TMtx are also accessible from Vector and Matrix types which are declared as records. Vector and Matrix types are implicitly casted to TVec and TMtx respectively where required. Consequently, it is possible to pass Vector as a parameter to all functions expecting TVec for example. The most important difference between TVec/TMtx and Vector/Matrix is that Vector/Matrix records do not have to be created and destroyed manually as other objects and that they can be used in expressions. In all other aspects TVec and Vector behavior is the same.

5.2 Combined floating-point precision

First introduced with MtxVec v6, combined floating-point precision allows floating point precision to be defined at run-time. Algorithm needs to be written only once, but can run as either single or double precision and this can be also decided by the end-user of the application. Single precision has the advantage of being in best case two times faster and using only half of the memory in compare to double precision. MtxVec source code can be compiled with preference for double or for single precision. This comes to effect when sizing objects without specifying their precision explicitly.

5.3 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 from including Delphi 2006 MtxVec declares two new types: Vector and Matrix. These two types are declared in MtxExpr unit as records which encapsulate TVec and TMtx objects respectively. The usage of the new types is best demonstrated with an example:

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

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

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

When using Vector record:

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

All properties and methods of TVec and TMtx are mapped to Vector and Matrix records including Values/SValues and CValues/SCValues 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:

```
function test1(p: TVec): Vector; inline;
var bv: Vector;
begin
       bv.Copy(p);
       Result := bv*p; //same as: Result.Sqr(p);
//     Result := p*p; //that would not work
end;
```

MtxExpr.pas 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:

```
var bv,av: Vector;
       aTVec: TVec;
begin
       aTVec.Sin(av);
       bv.Sin(av);
       bv := Sin(av);
end;
```

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).

5.3.1 Element by element Vector/Matrix expression types

Multiply two vectors element by element:

```
var a,bv,av: Vector;
begin
a := av*bv;

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

To multiply vector and matrix element by element:

```
var av,a: Vector;
    bm: Matrix;
begin
a := av*bm;

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

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

5.3.2 Conversion between single and double precision

MtxVec functions will not auto-convert floating-point precision. An exception will be raised, if the precisions do not match. Dedicated functions are provided for precision conversion. To optionally allow precision conversion use:

```
a.CopyTo(b, mvDouble);
```

This command will copy data from “a” to “b” and “optionally” apply floating-point precision conversion. Several methods accept the additional precision parameter. Usually however, the precision is simply preserved:

```
a.Copy(b); // a will be auto-sized to have the same precision as the source b
```

When auto-size is not possible, the precisions need to match:

```
a.Copy(b, 1, 2, 3); // copies values from b[1..3] to a[2..4];
```

5.3.3 Important computational precision considerations for scalars

- Delphi will convert integer constants to single precision, when floating point value is expected. This can cause problems when calling functions Ln(2), Sin(3), which will call single precision overloads. For this reason, the math functions like Ln, Sin, Cos are not overloaded for single precision parameters. Instead, the C style naming conventions are used: Lnf, Sinf, Cosf,..

- Avoid using overloads for functions with “single” type params. When a function accepts multiple parameters and all except for one of them is double, even Delphi 10.3 will select single precision overload, over the variant with double precision params. With older Delphi compilers the problems are more extensive.

- When using 64bit compiler and vectorized SSE instructions, there is a considerable cost for converting between single precision floating value and double precision floating point value. This can slow down the computation by about 5x. There is no warning from the compiler that these conversions are taking place. This is also the reason why single precision variants for all scalar Math functions are provided within the Math387 unit.

- MtxVec code is compiled with {$EXCESSPRECISION OFF} compiler flag. This affects only the 64-bit compiler. If you plan to use single precision within your code, it is recommended to specify the same.

- Single precision floating point has equal speed to double precision when working with scalars. The speed will be more (2x more ideally) only when using vector and matrix functions. When working with scalars it can be quite a challenge not to lose performance in compare to double precision, because of missing single/double implicit conversion compiler warnings.

5.3.4 Linear algebra Vector/Matrix expression types

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

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

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

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

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

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

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

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

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

5.3.5 Linear algebra with TVec and TMtx types

When working with TVecvectors 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.

5.3.6 Implicit type conversions

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

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

When a function requires an array of double, Vector or Matrix can be passed instead. Implicit type conversions will result in dereferencing Vector.Data.Values1D pointer which is an array of double. Explicit type conversions of Vector/Matrix to TDoubleArray or TCplxArray will result in a copy operation and will also perform floating point precision conversion if needed.

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

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

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

    //To make linear algebra multiplication and divison use functions

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

    //of course you can mix matrices and vectors

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

5.4 Vector and matrix initialization

From including Delphi XE7 the following syntax is supported:

```
    av: Vector;
    am: Matrix;
begin
    av := [1 2, 3, 4];  //create real vector with 4 elements
    am := [[1 2], [3, 4]];  //create 2x2 real matrix with 4 elements
```

For Delphi versions older than XE7 the following is possible (discouraged for newer versions):

```
    av: Vector;
    am: Matrix;
begin
```

```
    av.SetIt(false, [1 2, 3, 4]);  //create real vector with 4 elements
    am.SetIt(2,2, false, [1 2, 3, 4]);  //create 2x2 real matrix with 4 elements
```

It is always possible to use the "CopyFromArray":

```
    av: Vector;
    am: Matrix;
    ad: TDoubleArray;
begin
    ad := TDoubleArray.Create(1,2,3,4);
    //create real vector with 4 elements
    av.CopyFromArray(ad);

    am.Size(2,2);
    am.CopyFromArray(ad);  //create 2x2 real matrix with 4 elements
```

The precision of the resulting av and mv variables will be set to the "default" or preferred precision with which MtxVec was compiled. For finer control over precision, the Size method needs to be called. The SetIt and CopyFromArray methods will preserve the current precision of the av and am variables. This is the case, where no extra parameter is provided to specify precision, but clearly a sizing operation will happen.

5.5 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.

5.6 Range checking

All methods and properties of TMtxVec descendants are explicitly "range checked". Range checking ensures that the user cannot 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.)

5.7 Making use of the abstract class

Many methods can accept any TMtxVec descendant class:

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

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

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(Src: TMtxVec, aComplex: Boolean);
```

Similarly, if both the complex and precision properties are to be changed but Length preserved:

```
TMtxVec.Size(Src: TMtxVec, aFloatPrecision: TMtxFloatPrecision);
```

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

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

TMtxVec.Values1D	- array property to access real double values
TMtxVec.SValues1D	- array property to access real single values

```
aTMtxVecObject.Values1D[1] := 1; //sets real value at index 1 to 1
aTMtxVecObject.SValues1D[1] := 1; //sets real value at index 1 to 1


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

TMtxVec.CValues1D	- array property to access double complex values
TMtxVec.SCValues1D	- array property to access single complex values

```
aTMtxVecObject.CValues1D[1] := Cplx(1,0); //sets complex value at index 1 to 1
aTMtxVecObject.SCValues1D[1] := CplxSingle(1,0);//set complex value at index 1 to 1
```

TMtxVec.PValues1D	- function returns a pointer to double value
TMtxVec.PSValues1D	- function returns a pointer to single value

```
Delphi:
```

```
var aPointer: Pointer;
...
       aPointer := aTMtxVec.PValues1D(1); //returns a pointer to value at index 1
       aPointer := aTMtxVec.PSValues1D(1); //returns a pointer to value at index 1
```

TMtxVec.PCValues1D	- function returns a pointer to double complex value
TMtxVec.PCValues1D	- function returns a pointer to single complex value

```
Delphi:
var aCplxPointer: Pointer;
aCplxPointer := aTMtxVec.PCValues1D(1); //returns a pointer to complex
aCplxPointer := aTMtxVec.PSCValues1D(1); //returns a pointer to complex
```

5.7.1 Writing abstract class code

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

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

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). The floating point precision of the SrcOmega will be assigned to the Dst.

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

- Pointers (IntPtr in .NET): PValues1D, PCValues1D, PIValues1D, PSValues1D, PSCValues1D
- Getting/settings values: Values1D, CValues1D, IValues1D, SValues1D, SCValues1D, 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 TMtx source data:

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

5.8 Function parameters

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

```
procedure Test1(v: TVec);
…
function Test2;
var av: TDoubleArray;
begin
       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.

5.9 Indexes, ranges and subranges

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

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

The first function version takes no parameters. The result overwrites the source data. The source data can be either real or complex and the method will apply the appropriate code to compute the result. The second version first checks the size of the source objects and tries to match the destination object to be of the same size. If the size operation is successful, the appropriate code is applied to compute the result.(See chapter 5.7 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 100x) for some numerical algorithms. The entire CPU architecture is based on the assumption that memory is accessed by consecutive memory locations in about 90% of cases. It is therefore best to first gather the scattered data into one dense vector, perform math operations and then scatter the gathered data back to the original location:

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

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

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

There are more routines that can help with scattered data:

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

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

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

Best performance can be obtained when using masks:

```
a.FindMask(b, '=',c);
d.GatherByMask(c, a);
c.ScatterByMask(d, a);
```

because the masks can be vectorized and indexed approach can’t be.

5.10 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 cannot be done. The closest syntax allowed is this:

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

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

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

5.11 Create and Free

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

```
a := TVec.Create;
b := TMtx.Create;
try
....
finally
    a.Free;
    b.Free;
end;
```

Or in a fast way, by using object cache:

```
CreateIt(a);
CreateIt(b);
try
....
finally
   FreeIt(a);
   FreeIt(b);
end;
```

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

It is also possible to specify how Vector and Matrix allocate TVec/TMtx. By default, they always allocate objects from object cache. If, however you need to allocate Vector type variable as a global variable, it is possible to request that the TVec is created in the standard way:

```
av := Vector.Create(false);
```

5.12 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 = record Re,Im: double; end;).

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

```
function GetSomething: Vector;
var av: Vector;
    ac: TCplx;
begin
    ac := '1+2i';  //Convert from string to complex number
    av.Size(10);    //vector size

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

5.13 MtxVec types

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

5.13.1 TCplx and TSCplx

Declared as:

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

TSCplx = packed record
        Re, Im: single;
        end;
```

This is the default complex number type used by MtxVec. Many object oriented libraries declare the complex type as an object tpye (TComplex = class(TObject)). This means that consecutive elements within the array are not stored at consecutive memory locations and thus not allowing for vectorized operations on arrays. TCplx has overloaded all the necessary operators and it is possible to write nearly any expression the same as with real valued numbers.

5.13.2 TDoubleArray, TCplxArray

Declared as:

```
Delphi:
TDoubleArray = array of double;
TCplxArray = array of TCplx;
```

6 Accessing values of Vector and Matrix

6.1 Array property access

Example:

```
var a: Matrix;
begin
       a[1,0] := 2;
       a.Values[1,0] := 2;  //same as above
end;
```

Array properties allow a clean range checked access, but are not the fastest. The array properties perform explicit range checking when assertions are enabled (they can be disabled via a compiler switch). Typically, when they are used within loops, their access code is inlined and loops are unrolled by the compiler. This substantially increases their performance.

6.2 Direct dynamic array pointer.

Example:

```
Delphi:

var ap: T2DDoubleArray;
begin
       a.Length := 4;
       EnlistIt(a,ap);  {is the same as: ap := a.Values}
       ap[1,0] := 2;
       DismissIt(a,ap);
end;
```

This is the fastest method. The speed of access is equivalent to static arrays.

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:

```
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 record type}

...

```
var ac: T2DCplxArray;
begin
       a.Rows := 4;
       a.Cols  := 4;
       a.Complex := True;
       Enlist(a,ac)       {is similar to: ac:= a.CValues}
       ac[1,0] := Cplx(2,0);
       DismissIt(a,ac);
        //.... {same as:}
```

```
        a.Values[2,0] := 2;
        a.Values[3,0] := 0;
         //.... {same as:}
         a.CValues[1,0].Re := 2;
         a.CValues[1,0].Im := 0;
         ...
end;
```

There is one important consideration, because of the way how Delphi works with dynamic arrays. This should not be attempted:

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

a.Values should **not** be assigned to TDoubleArray variable after a call to SetSubIndex or SetSubRange. This could temporarily corrupt the data in the "a" vector. Enlist/Dismiss pair should be used instead for ar or simply avoid assigning arrays internal to TVec/TMtx to local variables.

7 Memory management

7.1 Introduction

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

```
var a: Matrix;
begin
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 is to 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. Dynamic memory allocation of two dimensional matrices in Delphi is not done in one single block as expected by fortran routines. For that purpose, the Matrix uses it's own memory allocation to dynamically allocate two dimensional arrays in a single continuous block of memory. Therefore, there are two more properties available from TMtx and Matrix:

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

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

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

This will access the same matrix element as:

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

or

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

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

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

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

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

7.2 In-place/not-in-place operations

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

7.3 The Capacity property

TVec/TMtx/TVecInt/TMtxInt objects have a Capacity property. Initially this value is zero. When set to value larger than zero, this will be the minimum number of elements for which the object will allocate memory. The advantages are:

- Sizing the object becomes free from thread locking in case of multi-threading, if the “capacity” is not exceeded. The performance will scale linearly with thread count even when object is resized.
- Calling the Resize method is free from memory re-allocations and of the copy operation, if done under the “Capacity”.
- Each object can have different value of “Capacity”, but default is 0.

Important:

- The Capacity is not meant to be used to append new elements to an array one by one, resizing the vector/matrix for each element. It would not be efficient for this purpose.
- The Capacity has no effect when used with cached objects and the value of Capacity is less than object cache length.

8 Range checking

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

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

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

9 Why and how NAN and INF

Floating-point calculations are subject to various possible errors such as overflows or division by zero. Normally, Delphi will in such a case raise a floating-point exception, which the programmer can handle with try-except clauses. The behavior in other languages, like C and Fortran, is different. The floating-point processor (FPU) will produce special floating-point values like NAN (not a number) or INF (infinity) instead of raising exceptions.

When MtxVec is compiled to interface external high performance dlls, which are partially based on C and Fortran code, the floating-point exceptions need to be disabled to avoid unexpected behavior. Additionally, all algorithms of MtxVec written in pascal are also implemented with the assumption, that the floating-point exceptions are disabled. The unit Math387, which is required by the code for vector and matrix operations, disables floating-point exceptions during its initialization.

The Delphi is shipped with libraries, which are invariant to the state of the FPU exceptions. However, there could be third-party libraries (none that we would be aware off) or your own code, which assumes that FPU exceptions are enabled.

9.1 The advantages of writing code with FPU Exceptions disabled

- Delphi compiler on ARM CPU emulates FPU exceptions in software to maintain backward compatibility. This slows down most Delphi RTL math functions (sin, cos, etc…) by roughly 30x. (True for Delphi 10.3 Rio). MtxVec uses various tricks to work around this limitation.

- Allows code in loops without the if-then checks and without try-except blocks. This on its own speeds up the code. Instead, you can concentrate on the code purpose and let the CPU work out the details.

- When working with arrays, this can be 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.

- Newer instruction sets (SSE2 SSE4, AVX, AVX2, AVX512 etc..), introduced after year 1997 to speed up vectorized math operations, have limited use for exceptions. When applying one math operation to vector, some elements would compute OK, while for other, an exception would be raised. An exception would indicate, that computation for at least one element (group) failed, without providing info for which indices the math worked and for which not, either because the elements were processed concurrently or because they reside in not yet processed part of the array. Even if the result for some elements of the array would not be useful, this does not automatically invalidate the usefulness of other parts of the array.

9.2 *Support for NAN and INF provided by MtxVec*

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

```
var a: double;
begin
```

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

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

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

This however will not work. NAN and INF are not values, which are defined with all the bits of a floating-point variable. There are just a few bits that need to be set, 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) then ...
if IsInf(a) then ...
if IsNanInf(a) then ...
```

Note that these functions work correctly with different versions e.g. of INF: plus and minus infinity. Additionally, the IsNanInf is about 2x faster from IsNan or IsInf. MtxVec methods and routines correctly handle NAN and INF. It is therefore acceptable to write something like this:

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

9.3 If you need to keep FPU exceptions ON with 64bit Delphi compiler

Call the following in some initialization code after the initialization of Math387 unit:

```
System.Math.SetExceptionMask([]);
// enable all floating-point exceptions (no masking for any of them)
```

Before calling any MtxVec vector or matrix code write:

```
ClearSSEExceptions(false);
SetSSEExceptionMask([exInvalidOp,    exDenormalized,    exZeroDivide,    exOverflow,
exUnderflow, exPrecision]);
//disable all (not quite correct - other rules e.g. for Linux)
```

After finishing with MtxVec vector or matrix code write:

```
ClearSSEExceptions(false);
System.Math.SetExceptionMask([]); // enable all floating-point exceptions
```

10 Serializing and streaming

10.1 Streaming with TMtxComponent

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

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

- SaveTemplateToFile

Most Delphi/CBuilder developers know the first five routines. What is interesting about TMtxComponent is that all components derived from it have all their published properties streamed, without the need to make any changes to the five routines. Therefore, all components derived 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 default Delphi component streaming mechanism has a bug when it comes to streaming properties with a declared default value. This has been fixed for TMtxComponent.

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

10.2 Streaming of TVec, TMtx and TSparseMtx

Both TVec and TMtx type objects will be streamed when declared as published properties of a component. TVec and TMtx also have their own methods for streaming:

SaveToStream
LoadFromStream
SaveToFile
LoadFromFile
Assign

These routines will process all the published properties and data of the TVec and TMtx objects. These methods only save and load data in the binary format. (MtxVec also supports Matrix Market text file format.) However, sometimes it is necessary to read a text file. Here is how this can be done:

10.3 Write TMtx/Matrix to a text file

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

10.4 Read TMtx/Matrix from a text file

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

11 Input-output interface

11.1 Reading and writing raw data

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

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

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

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

11.2 Formatting floating point values – FloatToString

Math387 unit declares the following routines:

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

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:

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

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:

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

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

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

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

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

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

11.3 Printing current values of variables

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

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

Typically the Report object can be used like this:

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

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

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

11.4 Displaying the contents of the TVec and TMtx

11.4.1 As a delimited text

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

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

11.4.2 Within a grid.

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

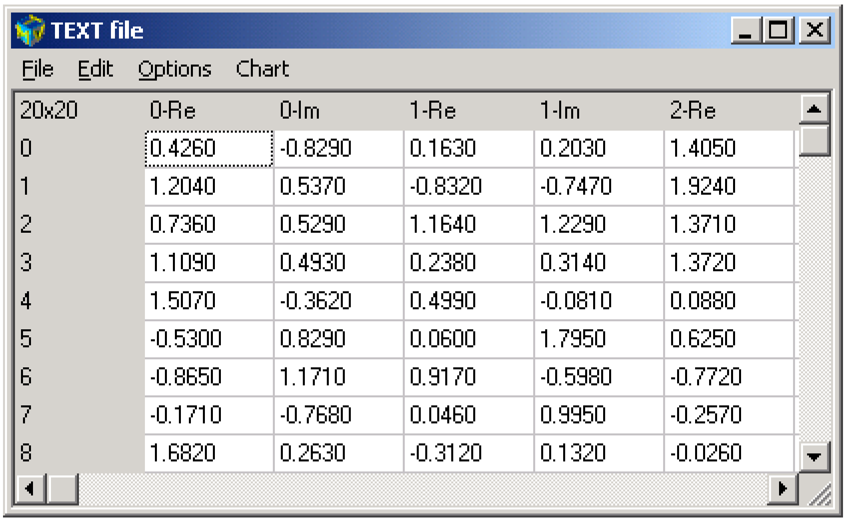
1. handling complex and real values
2. accept NAN and INF values
3. can display column/row headers or not
4. can get/set only a sub vector or a sub matrix.
5. Control the displayed precision

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

11.5 Charting and drawing

MtxVec provides two units: MtxVecEdit and MtxVecTee to support the display and charting of the contents of TVec and TMtx. By calling the ViewValues routine a simple editor will be displayed allowing the user to examine the values of a TVec or TMtx object. See Figure 3. This editor can be displayed modally or not. If it is displayed modally, the values can be changed and the contents of the object will also change. If the editor is not displayed modally, the changes will be discarded. If the changes are not to be saved, then the user can freely select the number formatting. If the changes are to be saved, the number formatting must be full precision or otherwise the values will be truncated to the displayed precision. The values can not be edited unless Editable flag from the Options menu is checked. From the editor Chart menu "Series in rows" or "Series in cols" ca be selected to draw the displayed values as a chart. Vector or matrix values can also be drawn directly on the chart by calling the DrawIt routine. This routine is located in the MtxVecTee unit. MtxVecTee routine contains a large set of DrawValues routines. This routines copy data from TVec or TMtx to the defined TChartSeries. Adding of new values is optimized for the TeeChart version used and charting can be considerably faster, if DrawValues is used. DrawValues routines also take care of any NAN's and INF's.

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



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

Figure 3

In the top left corner on Figure 3 is the size of the matrix (in this case 20x20). In the left most column are rows indexed starting with 0. The top row shows column labels. "0-Re" means that this is the first column of a complex matrix and shows the Real part of the complex number, "0-Im" column shows the imaginary component of the complex number stored in the first column of the matrix. On Figure 4 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).

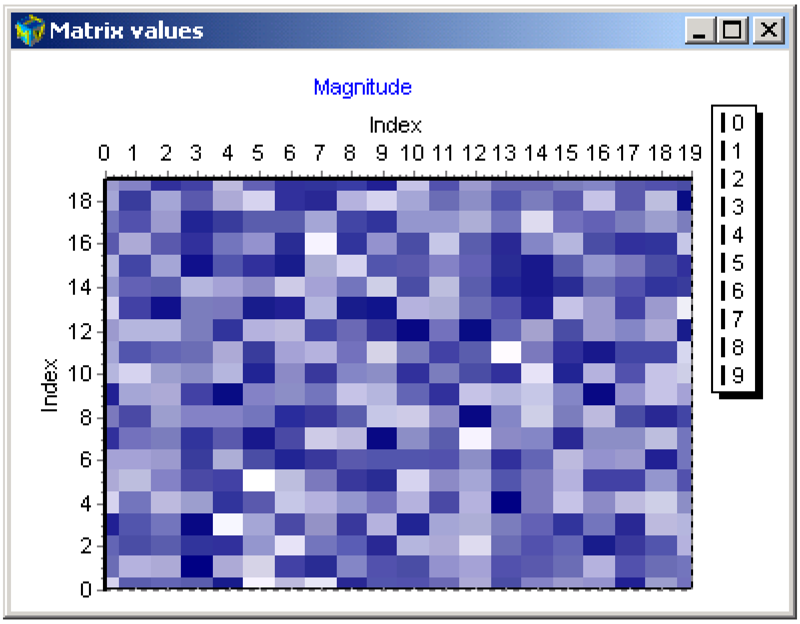

Matrix values
Magnitude
Index
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
18
16
14
12
10
8
6
4
2
0
Index
0
1
2
3
4
5
6
7
8
9

Figure 4

12 Programming style

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

12.1 Mixing TVec/TMtx and Matrix/Vector types.

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

12.2 Inlining of functions that use Vector and Matrix types

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

12.3 Try-finally blocks.

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

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

These have two purposes:

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

Do not write code like this:

```
CreateIt(b);

//...some code here

CreateIt(a);
yourProc(a,b);
Freeit(b);

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

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

When Vector and Matrix types are declared, the compiler automatically generates try-finally around them to ensure that the memory they allocate will be freed.

12.4 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.

12.4.1 Invalid parameter passed:

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

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

12.4.2 Reformat the exception

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

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

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

12.5 Indent the code.

A procedure looking like this is much more readable:

```
var a,b,c,d: TVec;
```

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

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

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

This makes it difficult to track CreateIt/FreeIt and Create/Destroy pairs. If you need to return a variable use the Vector/Matrix type. With Vector/Matrix records you are free to return them as a function result and this is even recommended, because it improves code readability and greatly simplifies the programming.

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

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

- Object cache might run out of precreated objects and calls to CreateIt/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.

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

```
var av: Vector;
begin
      av := Vector.Create(false); //specifying false avoids CreateIt/FreeIt
```

13 Getting up to speed

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

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

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

//By using expressions:
function YourFunc(const a,b: Vector; c1,c2,ea,eb: TSample): Vector; inline;
begin
       Result := (c1*A+c2)/Power(1.0 + Power(B,ea), eB);
end;

//Using TVec:
procedure YourFunc(a,b,Result: TVec; c1,c2,ea,eb: TSample);
var a1,b1: TVec;
begin
       if a.Length <> b.Length then Eraise('a.Length <> b.Length');
       CreateIt(a1,b1); //work vectors
       try
              a1.Copy(a);
              a1.Scale(c1);
              a1.Offset(c2);
              b1.Power(b,ea);
              b1.Offset(1);
              Result.Power(b1,-eb);
              Result.Mul(a1);
       finally
              FreeIt(a1,b1);
       end;
end;
```

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.

13.2 Block based processing

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

Plain function:

```
Delphi:

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

Vectorized with expressions:

```
function MaxwellPDF(x: Vector; a: double): Vector; inline;
var tmp: Vector;
begin
   tmp := Sqr(x);
   Result := Sqrt(4*a*INVTWOPI)*a*tmp*Exp(-0.5*a*tmp);
end;
```

Vectorized function:

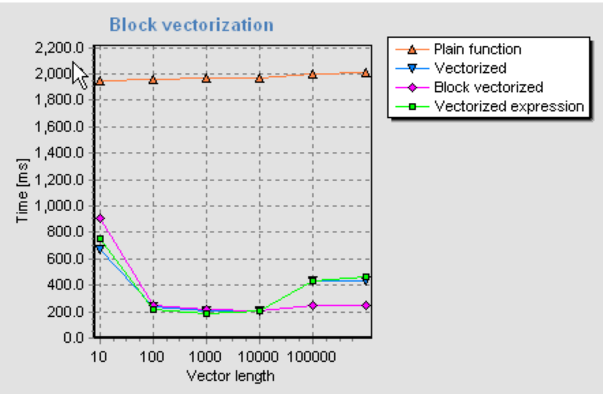
```
  procedure MaxwellPDF(X: TVec; a: TSample; Res: TVec);
  var Res1: TVec;
  begin
    CreateIt(Res1);
    try
      Res1.Sqr(X);
```

```
      Res.Copy(Res1);
      Res.Scale(-0.5*a);
      Res.Exp;
      Res.Mul(Res1);
      Res.Scale(Sqrt(4*a*INVTWOPI)*a);
    finally
      FreeIt(Res1);
    end;
  end;
```

Block vectorized function:

```
  procedure MaxwellPDF(X: TVec; a: TSample; Res: TVec);
  var Res1: TVec;
  begin
    CreateIt(Res1);
    try
      Res.Size(X);
      Res.BlockInit;
      X.BlockInit;
      while not X.BlockEnd do
      begin
          Res1.Sqr(X);
          Res.Copy(Res1);
          Res.Scale(-0.5*a);
          Res.Exp;
          Res.Mul(Res1);
          Res.Scale(Sqrt(4*a*INVTWOPI)*a);
          Res.BlockNext;
          X.BlockNext;
      end;
    finally
        FreeIt(Res1);
    end;
  end;
```

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



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

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

13.3 Common pitfalls

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

Vectorized expressions specifics:

10. Do not forget to inline functions which use vectorized expressions. When using vectorized expressions it is recommended to inline most computationally intensive methods either manually or with the help of “inline” keyword. This will substantially reduce the number of temporary variables allocated by MtxVec and thus help keep the working set of the variables inside the CPU cache.
11. Delphi compiler does not support a specific optimization called: collection on common sub expression. If your expression contains the same expression multiple times, be sure to assign its result to a temporary variable to prevent the expression from being evaluated multiple times.
12. Use parenthesis to indicate which part of the expression should be evaluated first. There is no optimization analysis that would be based on precedence of the same operator. Namely,

multiplying two vectors or multiplying two real numbers is very different in terms of clock cycles used. But the compiler does not know that.

13.4 Code vectorization methods

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

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

Many times, the code vectorization works great, except for a few special values which have to be handled separately. If those 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 subvectors and store the indices of individual values within the vector separately. Once the processing of each separate part has completed, merge the individual parts together again. Good starting point for this approach are TVec.Gather and TVec.Scatter methods.

13.5 Enabling the expressions for Multi-core CPU’s

Functions using Vector/Matrix types can also expect their expressions to execute on multiple CPU cores efficiently. For example:

```
function MaxwellPDF(x, a: Vector): Vector; inline;
var tmp: Vector;
begin
   tmp := Sqr(x);
   Result := Sqrt(4*a*INVTWOPI)*a*tmp*Exp(-0.5*a*tmp);
end;
```

Vector length required for effective threading in this case may not exceed the size of the CPU cache. 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.

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.

13.6 Efficient multithreading of MtxVec code

With MtxVec the code speed-up can be achieved in three steps:

1.) With vectorization
2.) Applying block processing to vectorization
3.) By multithreading the vectorized code in block mode.

The order of steps here is very important: If block processing is not implemented, the multithreading speedup may be even negative.

To demonstrate the issues we start with a piece of code which is based on the article:

Fast computation of scattering maps of nanostrctures using graphical processing units.
Published in Journal of Applied Crystalography, Vol. 44, part 3, 2011.

The examples posted here are part of the MtxVec Demo application. (See the “Efficient Multithreading” example). The code to be accelerated in Delphi looks like this:

```
  nh = 30;
  nk = 30;
  nl = 30;
  nhkl :=nh*nk*nl;
  nx = 30;
  ny = 30;
  nz = 30;
  nxyz :=nx*ny*nz;
  for i := 0 to nhkl-1 do
  begin
      F[i] := C_ZERO;
      for j := 0 to nxyz-1 do
          F[i] := F[i] + Expj(TWOPI * (h[i]*x[j] + k[i]*y[j] + l[i]*z[j]));
  end;
```

The ability to accelerate this code snippet depends a lot on the dimensions: In our current example they are set to 30. Now we can make our first MtxVec version and apply vectorization:

```
Var vx, vy, vz, Res: Vector;
begin
  ..
  for i := 0 to nhkl-1 do
  begin
    Res := Expj(TWOPI * (h[i]*vx + k[i]*vy + l[i]*vz));
    F[i] := Res.Sumc;
  end;
end;
```

Note that we now always read the same three vectors: vx, vy and vz on every iteration of the loop. This means that we have created the opportunity to get and keep that data in CPU cache. To achieve that, we apply block processing:

```
  for i := 0 to nhkl-1 do
  begin
      vx.BlockInit;
      vy.BlockInit;
      vz.BlockInit;

      while not vx.BlockEnd do
      begin
        Res.Expj(TWOPI * (h[i]*vx + k[i]*vy + l[i]*vz));
        F[i] := F[i] + Res.Sumc;

        vx.BlockNext;
        vy.BlockNext;
        vz.BlockNext;
      end;
  end;
```

With this, we have broken vx, vy and vz in to shorter arrays such that when called within the while Loop, the CPU does not need to access the main memory.It can keep this vectors within the L2 or even L1 cache. We also achieved something else: When data is located within the CPU cache it is local to each individual core and that core can run completely independent from other cores. Thus, we also created the basis for efficient threading.This brings us to the next threaded version:

```
procedure   TMtxVecThreadingForm.MyLoopB(IdxMin,   IdxMax:   Integer;   const
Context: TObjectArray);
var   i: Integer;
        Res: Vector;
        vxb, vyb, vzb: Vector;
```

```
begin
  vxb.BlockInit(vx);
  vyb.BlockInit(vy);
  vzb.BlockInit(vz);

  while not vxb.BlockEnd do
  begin
    for i := IdxMin to IdxMax do
    begin
      Res := Expj(TWOPI * (h[i]*vxb + k[i]*vyb+ l[i]*vzb));
      F[i] := F[i] + Res.Sumc;
    end;

    vxb.BlockNext;
    vyb.BlockNext;
    vzb.BlockNext;
  end;
end;
```

this MyLoopB function is passed to a function from MtxForLoop unit:

```
  DoForLoop(0, nhkl-1, MyLoopB, nil, []);   // Code execution will not
//continue until all threads have finished.
```

Important to observe:

1.) The inner most function keeps the vectorized form

```
            Res := Expj(TWOPI * (h[i]*vxb + k[i]*vyb+ l[i]*vzb));
            F[i] := F[i] + Res.Sumc;
```

This means we can write big algorithms with thousands of line of code and apply the threading at the top most function once, by breaking input data in in smaller sections.

2.) Each thread is working on its own distinct set of data. There are no conflicts between threads.
3.) The length of vectors within each threads data is small enough to fit inside of the CPU cache. This is ensured by the BlockInit methods, which automatically breaks vector in to suitably large parts.

And finally, the results of our experiments running on Core i5 4690:

Algorithm variant (dimension 30)	**Timing [s]**
CPU - using pure pascal	39
CPU - using MtxVec, one CPU core:	12
CPU - using MtxVec with blocks	6.9
CPU - using MtxVec with blocks and multithreaded	1.7

The final ratio is 22x. Iif we increase all dimensions from 30 to 40 (the variables, nh, nk etc…):

Algorithm variant (dimension 40)	**Timing [s]**
CPU - using pure pascal	229
CPU - using MtxVec, one CPU core:	97
CPU - using MtxVec with blocks	38
CPU - using MtxVec with blocks and multithreaded	9.5

Another interesting result, if we change computational precision from double to single precision:

Algorithm variant (dimension 40, single precision)	**Timing [s]**
CPU - using pure pascal	191s
CPU - using MtxVec, one CPU core:	32

CPU - using MtxVec with blocks	19
CPU - using MtxVec with blocks and multithreaded	4.1

13.7 Multithreading FFT

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

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

13.8 Managing the threads

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

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

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

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

14 Debugging MtxVec

14.1 Debugger Visualizer

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

- Vector and TVec
- VectorInt and TVecInt
- Matrix and TMtx
- MatrixInt and TMtxInt
- clVector and TOpenCLVector
- clMatrix and TOpenCLMatrix
- 1D static arrays of integers, smallint, bytes, singles, doubles and TCplx.
- 1D dynamic arrays of integers, smallint, bytes, singles, doubles and TCplx.
- TSignal type from DSP Master.

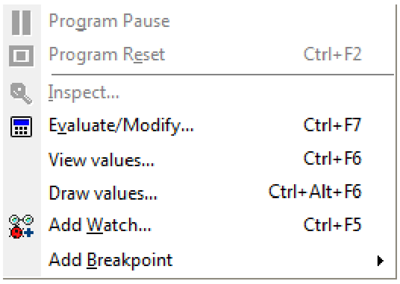


Figure 5 Two new commands in the Run menu.

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

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

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

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

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

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

and requires manual intervention. Similar is the case for the “with” sentence.

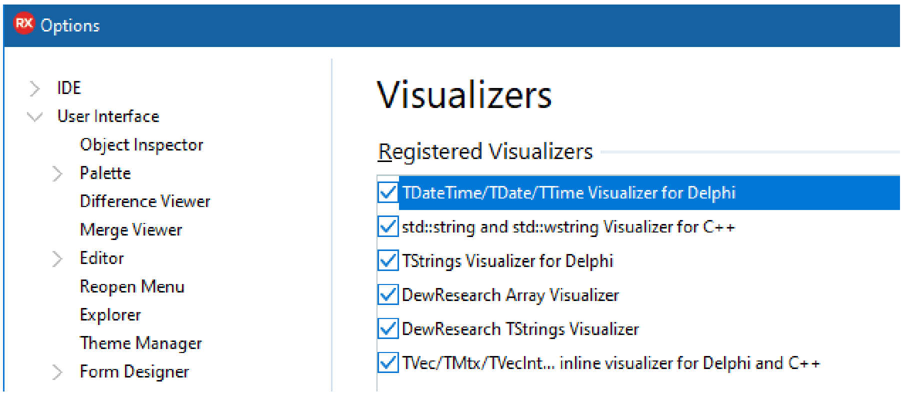
14.2 Viewing the values of Vector and Matrix in the debugger as an array

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

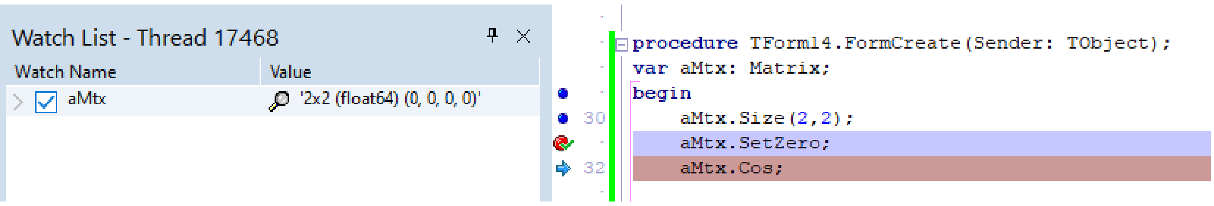
- aVector.fData.Values1D
- aVector.fData.SValues1D
- aVector.fData.CValues1D
- aVector.fData.SCValues1D
- aVector.fData.Values
- aVector.fData.SValues
- aVector.fData.CValues
- aVector.fData.SCValues
- aMatrix.fData.Values1D
- aMatrix.fData.SValues1D
- aMatrix.fData.CValues1D

- aMatrix.fData.SCValues1D

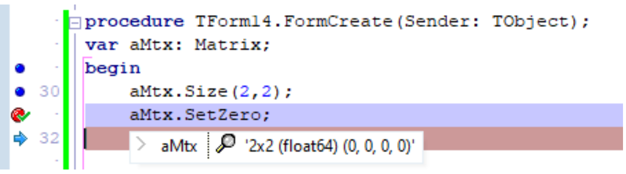
When the debugger watches are no longer sufficient, use the MtxVecTee.DrawIt or MtxVecEdit.ViewValues methods. Since Delphi XE2, the Debugger Visualizers are integrated with the Delphi IDE:



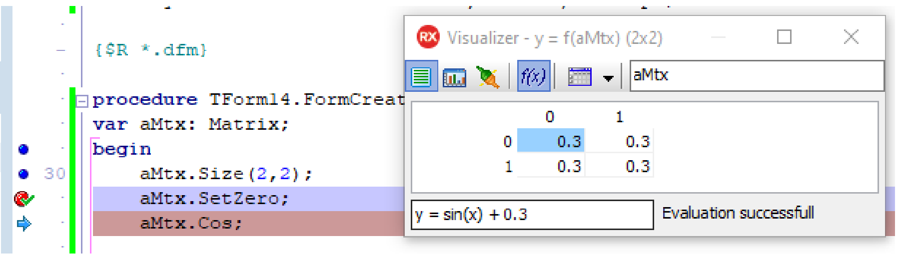
And can be added to watches:



And you can also have popup for the inline visualier for the quick variable inspection:



If the mouse is positioned over the variable while debugging. The visualize all the values, press the hourglass:



Here you can see the data on the chart, in the grid and additionally perform any math on the inspected variables. The expressions are automatically updated each time you step in/over your code. The input variable is always mapped to x and the output is y for the expression. The Visualizer windows can be many and support docking. They are preserved between debugging sessions.

14.3 Memory leaks

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

When using TVec and TMtx, the programmer should make sure to always match the Create and Free methods of objects and CreateIt/FreeIt pairs, as already emphasized. MtxVec unit holds two global variables: Controller.Pool[i].MtxCacheUsed and Controller.Pool[i].VecCacheUsed separately for each thread. Their value will show the number of not freed objects. After the application has finished using MtxVec routines, these two variables should have a value of 0. This means that all objects for which CreateIt was called, were also passed to the FreeIt routine.

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

14.3.1 Debugging method(s) to detect memory leaks

- Declare MtxVec to be compiled in “debug” mode in the build tool or specify “Assertions” to be ON for the compiler when compiling against the source code. This will cause Create/Freeit to create and destroy objects at the time when they are called.
- Follow instructions for the Fast MM Debugger or any other memory leak detection tool.

Once the debugging is finished, make sure to have Assertions OFF, for the release build, to obtain the best performance.

14.4 Memory overwrites

When using TVec and TMtx or Vector and Matrix memory ovewrites should be a thing of the past. But if you chose to work directly with unmanaged memory here are some hints. Memory overwrite errors may not pop up immediately. The reason for this is that TVec and TMtx objects residing in the object cache have preallocated a specified number of elements. Such pre-allocation speeds memory allocation for small vectors and matrices considerably and also makes reallocations faster. MtxVec explicitly checks all parameters passed to TVec and TMtx routines for range-check errors. For W32 it helps a lot, if range checking offered by the Delphi (W32) compiler is turned on. (Project - > Options - > Compiler - > Run time errors - > Range checking). The range checking mechanism offered by the W32 compiler raises false alarms, if Enlist/Dismiss is used in pair with SetSubRange/SetSubIndex:

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

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

The memory preallocation is disabled by calling:

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

The first parameter defines the number of TVec/TMtx objects created in advance and the second parameter defines the number of array elements for which to preallocate the memory. By setting memory pre-allocation to zero AV's will be raised much closer to the actual cause of the problem.

14.5 Open array parameters

Open array parameters in Delphi are declared as a parameter to the function or procedure, without specifying the size:

Procedure test(const a: array of single);

The user is then allowed to call this function in three different ways:

- Test(dynamicArrayOfSingle);
- Test(staticArrayOfSingle);
- Test([0,1,2,3]);

Here we have several issues to address:

- When the declaration is without the "const" modifier, Delphi makes a copy of the array by using the Move procedure, before passing it to the internals routine body. That working memory will be allocated on the heap and it will have memory alignment coming from the Delphi's memory manager.
- When using 64bit compiler, the alignment of the memory on the heap is guaranteed to be aligned on a memory address divisible by 64bit. For 32bit compiler and with Delphi 10.4, this is still only divisible by 32bit (32bit or 4 byte aligned).
- <u>All SSE/AVX instructions have a specific memory alignment requirement. It is only safe to pass TVecInt/TMtxInt and TVec/TMtx allocated memory to external vectorized methods.</u> When this alignment condition is not meet, it is possible that an immediate error, like access violation would occur, but it is also possible that a more subtle memory overwrite will happen. Consequently, before we can pass the array to external vectorized routines, we need to copy the array to aligned memory once more.

First copy was done by Delphi and the second copy we need to do and a third copy is possibly done by the routine, which has to apply whatever operation it was meant to do. This extra copying of course does not contribute to speed. For this reason, we do want to use the "const" modifier, to reduce the amount of the copying by one. Then, however:

- Within the body of the routine, it is not possible to know, what type of memory was passed in by the user, If this is dynamic, static or constant array.
- Delphi will allocate constant arrays (Test([0,1,2,3]) in the code section. The code section is not writable and is not guaranteed to be aligned. It has also not been well researched yet, what could be the interactions between memory allocated in the "code section" and the use of SSE/AVX instruction, where this memory would be only the source. Any attempt to write to the code section would of course give raise to an access violation.
- Consequently, we still need to perform one copy from the code section to newly allocated aligned memory.

When using the assignments like this:

aVector := [1,2,3,4,5];

Delphi will allocate a dynamic array on the heap and then copy the constants on its own, possibly from the code section, in to that newly allocated dynamic array. MtxVec will internally use byte copy type routine, which does not impose restrictions on the alignment of the source data, to copy the memory from the Delphi allocated dynamic array to internal aligned memory.

14.6 Debugging methods to detect SSE/AVX misaligned memory access

Try to obtain a reproducible case and progressively comment out different parts of the code until you find the line or block of code, which makes the problem go away, when commented out. This requires some cross verification, that indeed the correct line of code was found.

14.7 Debugging methods to detect memory overwrites or memory corruption

The following is supported:

1.) Enable range-checking for array access in the Delphi compiler options. MtxVec internally disables range checking, but it uses its own checks, which are controlled by the Assertions
2.) Enable assertions in the Delphi compiler options and build your app with MtxVec source code. This will disable object cache and enable explicit range checks for each array access.
3.) Use FastMM debugging features. FastMM is the default memory manager in Delphi and a separate copy is available on the internet with extended debugging features. This helps both with memory overwrites and memory leaks.

14.8 C++Builder linker related problems

When using MtxVec from C++Builder, the linker used by the C++ will ignore the name of the dlls used by Delphi to import external functions. Consequently, if by any chance you happen to statically link against a dll, which has the same function name as the functions from any of the existing dlls referenced from VCL/FMX/MtxVec by the Delphi pascal code, chances are that a silent redirect will happen during the linking stage. If the function with the same names actually do the same thing, it is not a big problem yet, but otherwise some seriously complicated debugging could be the result.

15 Mixing double and single precision

Before v6, MtxVec build tool allowed to produced two namespaces for the support of double and single precision:

- `Dew.Double`
- `Dew.Single`

This approach has been deprecated with v6.0, because it was not usable from C++Builder and FireMonkey among other things. A completely new implementation was done for v6 with many new features. Each Vector/Matrix, TVec/TMtx object now has a FloatPrecision property to control the computation and storage type at run-time.

16 FireMonkey support

MtxVec pacakages can be installed in to the IDE either for VCL or for the FireMonkey. Filenames are different only for form files. Example: MtxVecTee.pas for VCL and FmxMtxVecTee.pas for FireMonkey version. If the source file is not a form file, the filename does not change. Package names are different with a “Fmx” prepended. The FireMonkey version of MtxVec can be used in VCL projects and vice versa, but the UI components can only be used with corresponding forms.

17 Getting ready to deploy

Once the app has been debugged and is ready to be deployed, files required by MtxVec have to be included in the distribution package. These files are located in the windows\system or windows\System32\ on 32bit systems. On 64bit OS, the 32bit versions of the dlls are located in Windows\SysWOW64\ and the 64bit version of dlls are found in the Windows\system32 directory. Since May 2021, the libraries have their version embedded in their name. This allows coexistence of multiple versions on the same computer.

17.1 Compact MtxVec

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

- MtxVec.Dspd.X.Y.Z.dll. The library is optional and contains many vectorized and threaded functions from Intel IPP (excluding sin, cos, exp…). It is linked in when IPPDLL_DOUBLE is defined in the bdsppdefs.inc. When not linked in, the functions declared in ippspl_core.pas will be called instead. MtxVec.Dsp6s.dll is controlled with IPPDLL_SINGLE.
- MtxVec.Vmld.X.Y.Z.dll. This library is optional and contains many vectorized (but not threaded) math functions from Intel IPP (sin, cos, exp, ..) It is linked in when VMLDLL_DOUBLE and IPPVML_DOUBLE are defined in bdsppdefs.inc. When VMLDLL_DOUBLE is not defined the Delphi based Math387 functions written in assembler will be called instead. VMLDLL_SINGLE and IPPVML_SINGLE are used to control the linking of MtxVec.Vmls.X.Y.Z.dll.
- MtxVec.Vml.X.Y.Z.dll. This library is optional and contains vectorized and threaded math functions from Intel MKL (sin, cos, exp, …). It is linked in when VMLDLL_DOUBLE is defined and IPPVML_DOUBLE is not defined in bdsppdefs.inc. When VMLDLL_DOUBLE is not defined the Delphi based Math387 functions written in assembler will be called instead.
- MtxVec.Random.X.Y.Z.dll is used by the random generators located in RndGenerators.pas unit. The random generators are threaded and vectorized. It can be left out by commenting out {$DEFINE RNDDLL} in bdsppdefs.inc, or by not linking against the unit. The fall back code is shipped with Dew Stats Master only. This dll holds both single and double precision functions and also integer number generators.
- MtxVec.FFT.X.Y.Z.dll. This library contains multithreaded 1D, 2D and 3D DFT and FFT functions. It can be left out, by commenting out {$DEFINE FFTDLL} define in bdsppdefs.inc. When not linked in, the fallback code declared in Lapack_dfti_core will be used instead. This dll holds both single and double precision functions.
- MtxVec.Lapackd.X.Y.Z.dll. The linking is controlled with MKLDLL_DOUBLE and MKLDLL_SINGLE for MtxVec.Lapacks.X.Y.Z.dll
- Libiomp5md.dll. This library is mandatory, if using any other dll and contains OPENMP runtime used by other multithreaded dlls.

If distribution size is a problem, we can make a build of custom size dll's for your specific application. The provided dll's have specialized code of each function for a SSE3 capable CPU, and separate code paths for CPU's with SSE4/AVX/AVX2/AVX512 instruction sets. The appropriate code version is selected automatically, when the libraries are loaded.

17.2 MtxVec Core edition

The Core edition was the key new feature introduced with MtxVec v5. It allows MtxVec to be compiled completely without reference to external dlls and run with pure pascal code only. All the algorithms relevant to the add-on packages of DSP Master and Stats Master have been implemented in pascal. The numerical accuracy is comparable. Great care was taken when translating Fortran Lapack code to pascal to maintain nearly line by line comparability. The code runs slower (of course) than its Windows dll based counterpart, but greatly simplifies distribution to other platforms like Android and OSx/iOS. The following major features which were introduced in pascal with MtxVec v5:

1.) LU Factorization and LU Solvers.
2.) QR Factorization and QR solvers for rank deficient matrices
3.) EIG values and vectors for symmetric and general matrices
4.) SVD values and vectors
5.) SVD based linear system solvers
6.) FFT1D/2D/3D
7.) Remez Exchange algorithm for optimal FIR filter design

Entire code base maintains the following features:

a.) Optionally fully range checked by using Delphi dynamic arrays
b.) Requires Delphi XE6 with Update 1 or newer for substantially better performance.
c.) Features automated tests.

The support for the following features are missing from the Core Edition"

1.) Sparse matrix LU solvers and eigen values
2.) Complex number lapack routines
3.) The higher quality random generators

18 64bit version of MtxVec

Available for Delphi XE2 and later and for .NET. To run MtxVec from a 64bit application, you need to copy 64bit versions of the required dll's to System32 dir on a 64bit OS. 32bit dll's are to be located inside SysWOW64 dir and have the same name as their 64bit counterparts.

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

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

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

```
const IMAGE_FILE_LARGE_ADDRESS_AWARE = $0020;
{$SetPEFlags IMAGE_FILE_LARGE_ADDRESS_AWARE}
```

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

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

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

20 Building and deploying for Mobile devices

MtxVec Demo has been ported to FireMonkey and modified so that it will also work on tablets running on Android/iPad with at least 10" screen size. This allows largely the same user interface to be used for testing and demoing on desktop and mobile platforms. Of course, you can run the demo also on a phone with 5" screen, but the user interface of the demo app will not be practically functional.

20.1 Performance considerations

Performance of code running on mobile CPUs in 2015 is roughly in the range of Pentium 3 from 1999 and roughly 10x slower than new desktop computers running Intel Core i5. Open CL support has been disabled on mobile devices by both Apple and Google since Android v4.3. Most Open CL functions will work, but the build command will fail.

To avoid overhead due to automatic reference counting (ARC) used by the Delphi mobile compilers until Delphi Rio version 10.3, it is recommended to use Delphi version 10.4 or newer. FireMonkey for the mobile devices is typically maintained only for two most recent Delphi versions. Until including Delphi version 10.3, the following was a good practice with relation to ARC:

1. Use “const” modifier for all method parameters, which are object type. (procedure Fun(**const** a: TMyObj), or procedure Fun([Unsafe] a: TMyObj)).
2. Use [Unsafe] rather than [Weak] attribute for all fields and vars, which are object type and where you would like to avoid ARC code to be triggered.
3. If you would like all instructions inside of a method to avoid ARC code use: procedure Fun(**const** a: TMyObj); unsafe;

All these approaches are systematically used within MtxVec to achieve good performance.

20.2 Compatibility with desktop compilers

Mobile compilers before Delphi 10.4 by default enable zero based strings. Because of the way this is implemented and because of compatibility with existing code, the following is enabled by MtxVec by default (within bdsppdefs.inc):

```
{$IFDEF NEXTGEN}
{$ZEROBASEDSTRINGS OFF}
{$ENDIF}
```

This allows you to write new code which uses strings the same way as on desktop compilers and reuse existing code directly. With including Delphi 10.4 this is the default behavior also for the mobile device compilers.

20.3 CreateIt/FreeIt and object cache for Delphi 10.3 and earlier

They will work as expected on mobile devices, except for the fact that actual release of the object back to cache may not occur when calling FreeIt, but rather when the function, within which this call is made, ends.

20.4 Project configuration

Inside bdsppdefs.inc uncomment the FIREMK define. This can also be specified in the Project Options “Conditional defines”. The “Unit scope names” field of the Delphi compiler project options needs to contain: “FMX” and “FMXTee” separated by semicolon. This is true for “All configurations”. All platforms and all release/debug configurations targeting MtxVec with FireMonkey.

20.5 Frameworks on iOS and OS X

When a missing framework is reported by iOS or OS X linker, add the corresponding SDK framework to the list of the SDK frameworks and update local file cache. Delphi does not include all SDK frameworks by default to reduce linking time. MtxVec links with “Accelerator” framework on both iOS and OS X. Additional frameworks are linked in for DSP Master to support audio recording and playback.

21 Building for Linux64 and Mac OS 64bit

These two platforms use external linker, which creates shared objects (similar to dlls). For this to work, the command line compiler requires a path to the corresponding SDK. Because this path cant be easily guessed by the MtxVec build tool, the solution is to install FireMonkey components for Windows with the MtxVec build tool and then add MtxVec source code path to the corresponding Platforms (Linux64 and Mac OS 64bit) library search paths manually. See also topic 20.4 (Project configuration).

22 Rebuilding MtxVec packages

MtxVec packages can be rebuilt in installed in two ways:

a.) With the provided build tool. (DewBuildTool.exe)
b.) By opening the a corresponding project group: MtxVec6Delphi27.groupproj, for Delphi version 27.

23 Microsoft Defender

To prevent injection by AntiVirus, it is possible to request from Adminstrators PowerShell executed in the folder where the executable is located:

```
Set-ProcessMitigation -Name ilink32.exe -Enable DisableExtensionPoints
```

If we wish to exclude ilink32.exe from being scanned and/or monitored by the Microsoft Defender. More detail here:

https://docs.microsoft.com/en-us/powershell/module/processmitigations/?view=win10-ps

24 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/AVX/AVX2/AVX512 – indicating any of them indicates support for vectorized 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.

24.1 Basic vector math

Function	Class	Math expression	Features
Abs	TMtxVec	a[i] = |a[i]| a[i] = |b[i]|	SSE2/SSE3
Add	TMtxVec	a[i] = a[i] + B	SSE2/SSE3, RCX
Add	TDenseMtxVec	a[i] = b[i] + c[i] a[i] = a[i] + S*c[i]	SSE2/SSE3, RCX
AddProduct	TDenseMtxVec	a[i] = a[i] + b[i]*c[i]	SSE2/SSE3
ArcCos	TMtxVec	a[i] = 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]) a[i] = Cot(b[i])	SSE2/SSE3, SMP
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

24.2 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

24.3 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

24.4 Size, streaming and storage

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

24.5 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

24.6 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

24.7 Matrix conversions

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

24.8 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

25 Compatibility breaking changes

25.1 from version 1.x, 2.x

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

25.2 From version 5.0 to 5.03

All procedures which required TVec or TMtx to store integer values were modified to use TVecInt instead. Where it was necessary for the calling object to store integers, the functions were moved to TVecInt via class helper. Class helper was used to avoid circular references between units.

25.3 From Version 5.03 to 5.1

MtxVec.Spld5.dll was replaced with MtxVec.Dsp5d.dll. The rename was necessary, because many of the functions from Intel IPP v9 changed the behavior, parameter count etc, but kept the same names in compare to IPP v8.

25.4 From Version 5.32 to 5.4

- Dll version numbers and names have been increased from 5 to 6. This was needed, because many functions were reorganized and the dll API has changed considerably. This will ensure that applications using old and new dlls can coexist on the same computer.
- The MtxVec.Sparse6d.dll has been merged in to MtxVec.Lapack6d.dll.
- The msvcrt120.dll runtime library is no longer required to be included with distribution.
- FindIndexes and FindAndGather return a subranged object. This saves two memory allocations and one copy operation. To reset the object call SetFullRange() before sizing it again.

25.5 From Version 5.41 to 6.0

- Users of “concurrent” precision as implemented in MtxVec v5 and before need to refactor their source code to the new v6 API for supporting double and single precision.
- Deprecated support for C++Builder versions older than XE2.

25.6 From Version 6.0 to 6.0.4

- Affecting deployment, all dlls now have the version of the library embedded in to the name.
- Deprecated support for Embarcader RAD Studio older than XE3.