MtxVec v1.5

A programming guide to MtxVec

MtxVec version 1.5

www.dewresearch.com
Written by Janez Makovšek
Edited by prof. David Bell

1 Introduction

MtxVec is an object oriented numerical library. There have been quite a few attempts to make such a library for numerical analysis object oriented. Many people have tried to force a class hierarchy on numerical algorithms, but the most successfull libraries today do not use class hierarchies. For example, the basic element of Matlab (by Mathworks) is one single object. That object type can represent real or complex single value, vector, matrix or sparse matrix or even a string. MtxVec introduces two classes:

TVec for vector operations
TMtx for matrix operations

Both objects can hold either real or complex data. Both classes should be considered as sealed objects. (that is to say - no classes should be derived from them). Both classes hold a list of methods which may be called like this:

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

If the method can put its result in one object, then the result is placed in the object on the left. In the following example the result is placed in the objects on the right:

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

Its 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 opearation which has a TMtx type result will be a part of TMtx class.

1.1 Indexes, ranges and subranges

Most methods support indexing:

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

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

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

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

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

IValues points to the same memory as Values and CValues and is simply an integer array type pointer. 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. You can use FindAndGather 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);
```

1.2 TVec and TMtx methods as functions

Ideally a mathemathical expression could be written like this:

a := a*b + b;

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

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

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

a := c.Mul(a,b);

1.3 Create and destroy

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

```
a := TVec.Create;
B := TMtx.Create;
try
....
finally
    a.Destroy;
    B.Destroy;
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.

1.4 Complex data

Both TVec and 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;).

2 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.

2.1 Try-finally blocks.

Every time you call CreateIt/FreeIt or Create/Destroy pair, you should place it 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 has 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 code much more readable.

2.2 Raising exceptions

Because all code is now protected with try-finally blocks, exceptions can now 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.

2.2.1 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.

2.2.2 Reformat the exception

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

```
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.

2.3 By all means indent code.

A procedure should look like this:

```
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;
```

2.4 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 and breaks a few good rules of programming.

2.5 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:

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

3 Accessing values of TVec and TMtx

3.1 Default array property access.

Example:

```
var a: TMtx;
begin
       CreateIt(a)
       try
              a[1,0] := 2;
       finally
              FreeIt(a);
       end;
end;
```

Default array property allows a clean access, but is not very fast. It's use for large matrices is discouraged. The default array property performs explicit range checking.

3.2 Dynamic array pointer.

Example: `a.Values[1,0] := 2;` {where a is TMtx object}

Values is a dynamic array pointer. The access is significantly faster then with default array properties.

3.3 Direct dynamic array pointer.

Example:

```
var ap: T2DSampleArray;
begin
      CreateIt(a);
      try
             a.Length := 4;
             Enlist(a,ap);  {is the same as: ap := a.Values}
             ap[1,0] := 2;
             Dismiss(ap)
      finally
             FreeIt(a)
      end;
end;
```

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

Handling complex data :

Default array property access is not available for complex data because the object can have only one default array property. 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
       CreateIt(a);       {similar as a := TVec.Create;}
       try
              a.Rows := 4;
              a.Cols  := 4;
              a.Complex := True;
              Enlist(a,ac)      {is similar to: ac:= a.CValues}
              ac[1,0] := Cplx(2,0);
              Dismiss(ac);
              //.... {same as:}
              a.Values[2,0] := 2;
              a.Values[3,0] := 0;
              //.... {same as:}
              a.CValues[1,0].Re := 2;
              a.CValues[1,0].Im := 0;
              ...
       finally
              FreeIt(a);  {similar as: a.Destroy; a := nil}
       end;
end;
```

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

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

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

4 Memory management

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

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

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

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

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

There are some special issues that need to be taken in to account when working matrices. TMtx interfaces highly optimized FORTRAN code. Dynamic memory allocation of two dimensional matrices in Delphi is not done in one single block as expected by fortran routines. For that purpose, TMtx uses it's own memory allocation to dynamically allocate two dimensional arrays in a single block of memory. Therefore, there are two more pointers available from TMtx:

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

These two pointers 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 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];

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

a.Size(4,4,False);

Size method will ensure that no more memory is allocated than necessary when resizing. Imagine a 5x10000 matrix, being resized to 10000x5, but you set the rows to 10000 first and get 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.

5 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, if the default array property is used:

a[i,j] := 2; {performs the following:

if i > a.Rows-1 then raise Exception...
if i < 0 then raise Exception...

if j > a.Cols-1 then raise Exception...
if j < 0 then raise Exception...

a.Values[i,j] := 2}

If you are ever in doubt, about how much memory to allocate, or if some algorithm might be crossing the bounds of arrays, you can easily perform a test by using the default array property. Once the code is debugged, you can switch back to high speed array pointers.

In-place/not-in-place operations

As opposed to TVec, TMtx does not strictly separate between in-place/not-in-place operations. The reason is the high computational complexity of most matrix operations, thus making that feature almost obsolete. 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 should 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.

6 C++ Builder issues

MtxVec allocates memory for dynamic arrays in one continuous memory block. (fortran-style). This method is not compatible with two or more dimensional dynamic arrays allocated in C++. This type of memory allocation is necessary to make the arrays compatible with fortran libraries. (And use LAPACK). Because most of the matrix operations are based on LAPACK, this type of memory allocation is mandatory.

Consequently, in C++Builder matrix elemens cannot access the elements via Values property (example: a->Values[i][j] = 2;, will not work), because sometimes this will generate range check errors. This range check errors are false alarms raised due to the memory allocation incompatibility issue.

In Delphi, T2DSampleArray type (used by TMtx) can be used also for direct pointer reference, because only the range checking mechanism is broken and, if range checking is turned off range, the app will run just fine. But this is not the case for C++Builder.

The C++ template for Delphi dynamic arrays used in CBuilder uses its own explicit range checking, which cannot be disabled and the T2DSampleArray type therefore cannot be used in C++ Builder to access the memory of TMtx.

To circumvent this problem, elements have to be accessd via a secondary "values" property:

```
a->values[i][j] = 2;
```

or

```
a->cvalues[i][j] = StrToCplx("2+i");
```

Delphi dynamic arrays are wrapped with a template in C++ Builder making each access very slow. By obtaining a pointer of P2DSArray type you avoid this and get full C++ speed. (but no range checking) The P2DSArray type pointer can be conveniently obtained via the Enlist method.

With C++Builder it is recommended to use the Enlist method to access the elements of TMatrix or of TVector object. The Enlist method in C++Builder does not require a dismiss pair, as is the case in Delphi.

The a->values[i][j] access is slower then a memory pointer, but still faster then the template used by CBuilder for Delphi dynamic arrays.

7 Getting ready to deploy

Once the app has been debuged and is ready to be deployed, three files required by MtxVec have to be included in the distribution package. This files are located in the windows\system or winnt\system32 directory: mkl_support.dll, lapack.dll and bdspp.dll.

Three files form a minimum required dll set for MtxVec to run. If you want to support different CPU's (especially P4), there is a set of libraries located on www.dewresearch.com

If distribution size is a problem, we can make a build of custom size dll's for your specific application. Lapack dll's are linked dynamically and any routines not needed can be simply left out.

8 Why and how NAN and INF

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

MtxVec offers specialized routines for string to number and number to string conversions (StrToVal ,StrToCplx, FormatCplx, FormatSample, StrToSample, SampleToStr) and drawing routines (DrawValues, DrawIt) capable of handling NAN and INF values. By using those routines (located in math387 unit), the user will avoid most of the problems when working with NAN and INF values.

8.1 Delphi 4 and Delphi 5 issues with NAN and INF

Delphi 6 brought one important change when dealing with NAN and INF. In earlier versions of Delphi there were three routines causing problems when floating point exceptions were turned off: Integer(), Trunc(), and Frac(). These routines changed the exception mask of the FPU and re-enabled floating point exceptions. Each time a NAN or INF value was computed by the FPU (floating point processing unit) an exception flag was set, but no exception was raised, because the exceptions were off. When however the floating point exceptions would get enabled, this set exception flags would immediately raise one of the two exceptions: Invalid floating operation (NAN was a result at least once in the previous calculations) or Floating point overflow operation (INF was a result at least once in previous calculations). The only way to resolve this problem is to reset the exception flags of the FPU by calling ClearFPU routine, before a call to Trunc, Frac and Integer. This routine is located in the Math387 unit. Instead of writing code like this:

```
a := Trunc(b);  //invalid floating operation might get raised here.
```

one should write it like this:

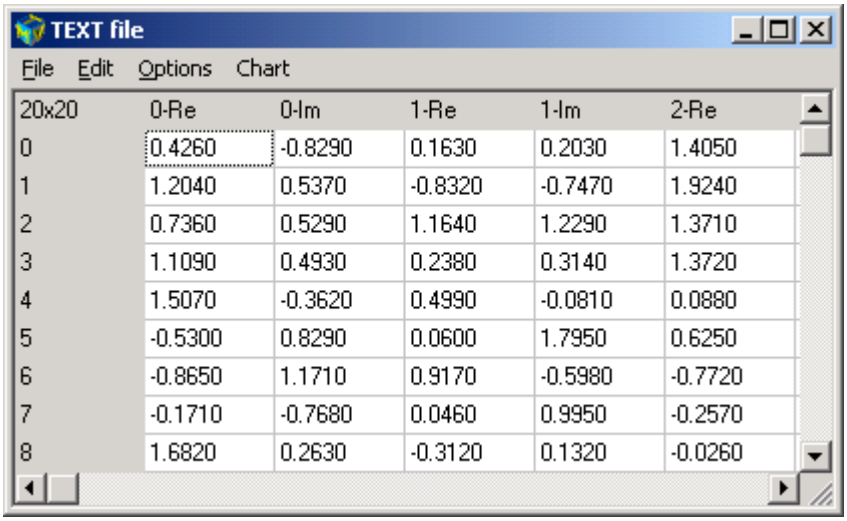
```
ClearFPU;
a := Trunc(b);
```

One important thing to remember with Delphi 4 and Delphi 5 is to always call ClearFPU before calling any code that might potentially use Trunc, Frac or Integer routines. The most common place for the ClearFPU is right before the values are passed to TeeChart for drawing. (TeeChart uses Trunc). If DrawValues routine from MtxVecTee unit is used, the ClearFPU is called automatically and all NAN and INF values are filtered out from the array and only valid values get drawn.

9 View values and working with TeeChart

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 you to examine the values of a TVec or TMtx object. See Figure 1. 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: TVec;
begin
	CreateIt(a);
	try
		a.LoadFromFile('c:\test.vec');
		ViewValues(a); //display a window showing values in "a"
		DrawIt(a); //display a chart of values in "a"
		...
	finally
		FreeIt(a);
	end;
end;
```



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 1

In the top left corner on Figure 1 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 2 the magnitudes of the values stored in the complex matrix can be seen. The layout of the values is the same as in the matrix editor (the left axis labels should be inverted).

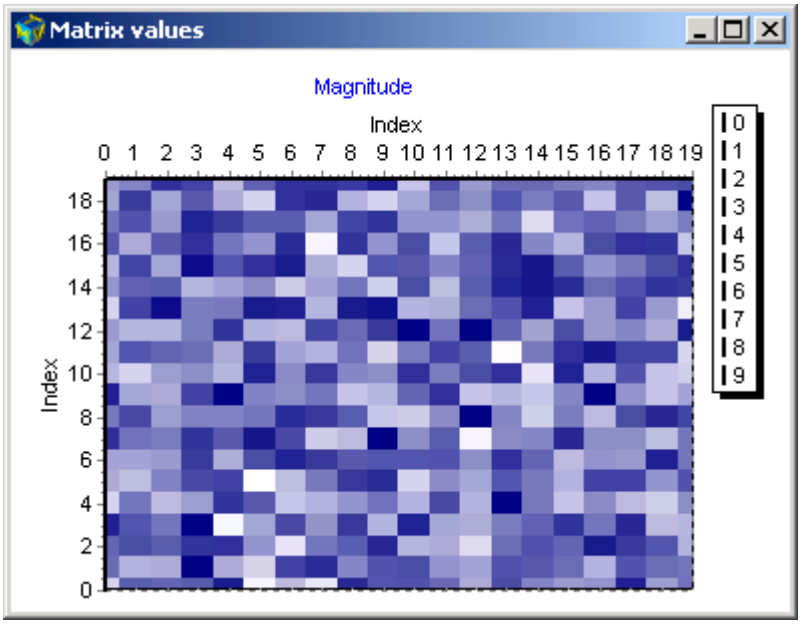


Figure 2

10 Debugging MtxVec

Delphi offers an excellent debugger. However there are a few things that have to be addressed separately.

10.1 Memory leaks

The programer should make sure to always match the Create and Destroy methods of TObjects and CreateIt/FreeIt pairs for TVec and TMtx objects, as already emphasized. MtxVec unit holds two global variables: MtxCacheUsed and VecCacheUsed. Their value will show the number of unfreed objects. After the application has finished using MtxVec routines, these two variables should have a value of 0. This means that all objects for which CreateIt was called, were also passed to the FreeIt routine. For TObject type no such reference counting exists.

10.2 Memory overwrites

When using TVec and TMtx the memory overwrites can be difficult to notice. 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. This however does not help, if you are writing an algorithm which accesses the values directly from TVec.Values/TMtx.Values array. For TVec it helps, if range checking offered by the Delphi compiler is turned on. (Project - > Options - > Compiler - > Run time errors - > Range checking). When working with matrices, the range checking mechanism offered by the compiler raises false alarms unless you work with TMTx.Values1D array. This false alarm occur, because the memory for matrices is allocated in a single block, which is not in compliance with the Delphi dynamic arrays. The memory for matrices must be allocated in a single block so that it is compatible with Fortran style memory allocation

and LAPACK. Anyhow, to protect yourself against memory overwrites when working with matrices, you can use TMtx.Values1D array for which the range checking mechanism works OK and you can disable the memory preallocation feature. The memory preallocation is disabled by calling:

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

The first parameter defines the number of TVec/TMtx objects created in advance and the second parameter defines the number of elements for which to preallocate the memory. By setting memory preallocation to zero you will get AV's much closer to the actual cause of the problem.

10.3 Intel SPL error checking

MtxVec makes heavy use of Intel Signal processing library (Intel SPL) to support the latest CPU architectures (P4, Itanium etc...). The erorr checking mechanism of Intel SPL does not allow "silent" error reporting without some kind of memory corruption. This means that once an invalid parameter is encountered, the application calling the dll will be halted without any error message. Because MtxVec does its own checking of valid parameter combinations, this problem is minimized. In some cases however the user might pass an invalid parameter combination, which was not predicted. In such cases the application will simply terminate. You can use step by step debugging to see which routine caused the problem and prevent invalid parameters being passed to the dll routine.

11 Loading text files

TMtx and TVec objects have SaveToStream/LoadFromStream and SaveToFile/LoadFromFile methods. But this methods only save and load data in the binary format. (MtxVec also supports Matrix Market text file format.) However, sometimes it is neccessary to read a text file. Here is how this can be done:

11.1 Write a text file

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

11.2 Read from a text file

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