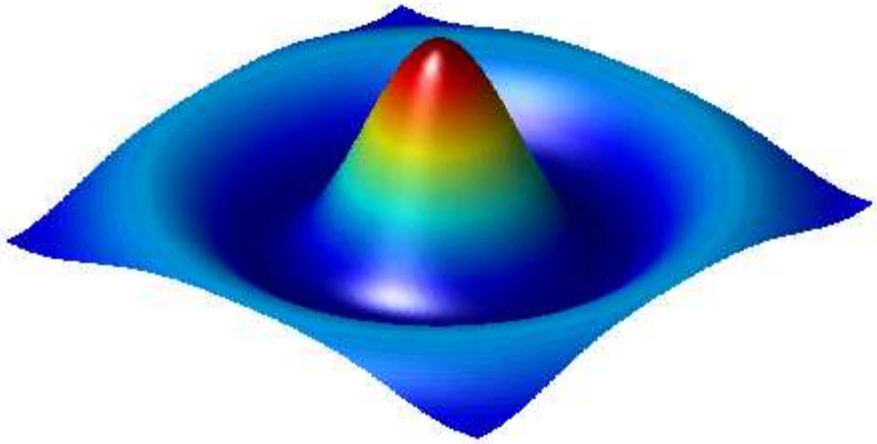

MTIMESX

Fast Matrix Multiply for MATLAB®
With Multi-Dimensional and OpenMP® Support

Version 1.40

October 4, 2010

By James Tursa

Contents

1) Introduction

MTIMESX is a fast general purpose matrix and scalar multiply routine that utilizes BLAS calls and custom code to perform the calculations. **MTIMESX** also has extended support for n-Dimensional (nD, n > 2) arrays, treating these as arrays of 2D matrices for the purposes of matrix operations. BLAS stands for Basic Linear Algebra Subroutines. The BLAS is a library of highly optimized routines for various scalar-vector, vector-vector, matrix-vector, and matrix-matrix linear algebra operations. MATLAB makes calls to their BLAS library in the background whenever you do a matrix multiply. **MTIMESX** links to and calls these same BLAS library routines directly.

"Doesn't MATLAB already do this?" For 2D matrices, yes, it does. However, MATLAB does not always implement the most efficient algorithms for memory access, and MATLAB does not always take full advantage of symmetric and conjugate cases. **MTIMESX** attempts to do both of these to the fullest extent possible, and in some cases can outperform MATLAB by 300% - 400% for speed (yes, you read that right, 3x – 4x faster). For nD matrices (treating them as arrays of 2D matrices), MATLAB does not have direct support for this. One is forced to write loops to accomplish the same thing that **MTIMESX** can do faster (50x - 100x in some cases). NOTE: The MATLAB intrinsic function for matrix multiplication is called ***mtimes***. i.e., when you type the expression A * B, MATLAB actually calls the function ***mtimes***(A,B). In all of the discussions below, ***mtimes*** (without the x) always refers to the MATLAB built-in matrix multiply operation.

2) Operating Modes

MTIMESX has three basic methods for calculating results: BLAS calls, C coded loops, and OpenMP multi-threaded C coded loops. **MTIMESX** has six operating modes that determine which of these methods is used. They are as follows:

'BLAS' mode

Forces **MTIMESX** to do all calculations with BLAS calls. This mode attempts to reproduce the MATLAB intrinsic function ***mtimes*** results exactly by calling BLAS library routines (if available) to do all calculations regardless of speed implications. So all scalar multiplies, vector-vector multiplies, and matrix-vector multiplies are performed by the BLAS routines. All of the calculations involving full double or single matrices have BLAS routines available to do the work. In addition, the (scalar) * (sparse) calculation is done with BLAS calls in this mode. If the BLAS routines are multi-threaded on your machine then you will get the benefit of this multi-threading when you use **MTIMESX** in this mode. Note that **MTIMESX** does not directly support generic sparse matrix multiplies, but instead just calls the MATLAB intrinsic function ***mtimes***.

'LOOPS' mode

Forces **MTIMESX** to do all calculations that have C code loops available to use that code. This mode attempts to reproduce the MATLAB intrinsic function ***mtimes*** results closely but not exactly by performing some of the calculations with C code loops instead of BLAS calls regardless of speed implications. If no C code loop method is available for a particular calculation, such as a generic matrix multiply, then BLAS calls will be used. Note that 'LOOPS' mode never uses OpenMP multi-threading for the C code loops. The operations that have C code loops available to do the calculation are (unless otherwise noted, results will not match MATLAB exactly):

scalar * array (results will match MATLAB exactly)
vector outer product (results will match MATLAB exactly)
vector inner product (i.e., dot product)
vector * matrix (using a series of dot product calculations)
matrix' * vector (using a series of dot product calculations)
matrix.' * vector (using a series of dot product calculations)
(4x4 or smaller matrix) * (4x4 or smaller matrix) (using inline unrolled loops)

'LOOPSOMP' mode

Forces **MTIMESX** to do all calculations that have OpenMP multi-threaded C code loops available to use that code. This mode attempts to reproduce the MATLAB intrinsic function ***mtimes*** results closely but not exactly by performing some of the calculations with OpenMP multi-threaded C code loops instead of BLAS calls. The 'LOOPSOMP' mode is basically the same as the 'LOOPS' mode as far as the basic underlying C code is concerned. However, the 'LOOPSOMP' mode will split up and multi-thread the calculation using OpenMP pragmas. As a result, the order of operations will be different from the 'LOOPS' mode and you should not expect to get exactly the same result as the 'LOOPS' mode. The user can control the number of threads requested up to the number of processors available. **MTIMESX** does not use atomic variable updates nor does it use variable reduction techniques to produce results. Instead, **MTIMESX** will store results of each individual thread separately and then when all threads are complete the final result will be calculated by combining the individual thread results using a pre-defined order. Thus you should be guaranteed to get the exact same result from run to run as long as the inputs are the same and the number of threads used is the same. The result will not depend on the actual order that the individual threads execute. This would not be the case if atomic updates or reduction techniques were used. Note that 'LOOPSOMP' mode is only available if you have compiled **MTIMESX** with an OpenMP compliant compiler such as gcc, Intel, or the latest MSVC compilers. Unfortunately the LCC compiler that is shipped with MATLAB is not an OpenMP compliant compiler, so you will not have this mode available if you use the LCC compiler. The 'LOOPSOMP' mode reverts to the 'LOOPS' mode if you compiled with a non-OpenMP compliant compiler. You can find a list of OpenMP compliant compilers here:

http://openmp.org/wp/openmp-compilers/

Not all of the (4x4 or smaller matrix) loops cases are multi-threaded. The only cases that are multi-threaded are the following, and even these are only multi-threaded for the real nD case and if there is no singleton expansion involved (this may be expanded in future versions):

(4x4 or smaller matrix) * (4x1 or smaller vector)
(1x4 or smaller vector) * (4x4 or smaller matrix)

To set the number of requested threads to use in the 'LOOPSOMP' mode, use one of the 'OMP_SET_NUM_THREADS' directives (see next section). If you don't set the number of threads to request, **MTIMESX** will automatically set it to the number of processors available.

The next three modes are special combinations of the above three basic modes.

*'MATLAB' mode (the default when **MTIMESX** is first run)*

Forces **MTIMESX** to use the fastest 'BLAS' or 'LOOPS' method that matches MATLAB exactly. 'MATLAB' mode attempts to reproduce the MATLAB intrinsic function ***mtimes*** results exactly using whichever method is likely to be faster as long as the results match ***mtimes*** exactly. When there was a choice between faster code that did not match the MATLAB intrinsic ***mtimes*** function results exactly vs slower code that did match the MATLAB intrinsic ***mtimes*** function results exactly, the choice was made to use the slower code. Speed improvements were only made in cases that did not cause a mismatch. Caveat: Only tested on a 32-bit WinXP PC with later versions of MATLAB (R2006b - R2010a). This works, but MATLAB may use different algorithms for ***mtimes*** in earlier versions or on other machines that were unavailable for testing, so even this mode may not match the MATLAB intrinsic ***mtimes*** function exactly in these other cases. 'MATLAB' mode is the default mode when **MTIMESX** is first loaded and executed (i.e., the first time you use **MTIMESX** in your MATLAB session and the first time you use **MTIMESX** after clearing it from memory). Also note that the choice of method for (scalar) * (array) and vector outer product may not be the optimal choice for speed on your particular platform. If you find that to be the case you can force a specific method by selecting 'BLAS', 'LOOPS', or 'LOOPSOMP' instead of 'MATLAB' for that particular calculation.

'SPEED' mode

Forces **MTIMESX** to use the fastest 'BLAS' or 'LOOPS' method, regardless of whether or not it matches MATLAB exactly. This mode attempts to reproduce the MATLAB intrinsic function ***mtimes*** results closely, but not necessarily exactly, by selecting the faster of BLAS calls or C code loops methods (if available). When there was a choice between faster code that did not exactly match the MATLAB intrinsic ***mtimes*** function vs slower code that did match the MATLAB intrinsic ***mtimes*** function, the choice was made to use the faster code. Speed improvements were made in all cases that I could identify, even if they caused a slight mismatch with the MATLAB intrinsic ***mtimes*** results. NOTE: The mismatches are the results of doing calculations in a different order and are not indicative of being less accurate. Also note that the speed improvements are highly dependent on the version of MATLAB you are running and the specific compiler you are using. 'SPEED' mode never uses OpenMP multi-threaded C code loops to perform calculations. Also note that the choice of BLAS calls vs C code loops made by **MTIMESX** in 'SPEED' mode may not be optimal for your particular platform. If you find that to be the case you can force a specific method by selecting 'BLAS' or 'LOOPS' instead of 'SPEED'.

'SPEEDOMP' mode

Forces **MTIMESX** to use the fastest 'BLAS', 'LOOPS', or 'LOOPSOMP' method, regardless of whether or not it matches MATLAB exactly. This mode attempts to reproduce the MATLAB intrinsic function ***mtimes*** results closely, but not necessarily exactly, by selecting the faster of BLAS calls or C code loops methods (if available) or OpenMP multi-threaded C code loops (if available). When there was a choice between faster code that did not exactly match the MATLAB intrinsic ***mtimes*** function vs slower code that did match the MATLAB intrinsic ***mtimes*** function, the choice was made to use the faster code. Speed improvements were made in all cases that I could identify, even if they caused a slight mismatch with the MATLAB intrinsic ***mtimes*** results. NOTE: The mismatches are the results of doing calculations in a different order and are not indicative of being less accurate. Also note that the speed improvements are highly dependent on the version of MATLAB you are running and the specific compiler you are using. Note that the choice of BLAS calls vs C code loops vs OpenMP multi-threaded C code loops may not be optimal for your particular platform. If you find that to be the case you can force a method by selecting 'BLAS', 'LOOPS', or 'LOOPSOMP' instead of 'SPEEDOMP'. If you have not compiled with an OpenMP compliant compiler then 'SPEEDOMP' mode reverts to 'SPEED' mode.

Which Mode Is Best?

That depends on your needs. If you want your results to match MATLAB exactly in all circumstances, then use 'MATLAB' mode. If you are willing to give up exactly matching MATLAB and want more speed, then use either 'SPEED' mode or 'SPEEDOMP' mode (if available). Note that in some cases 'BLAS' or 'MATLAB' mode may actually be the fastest mode for a particular calculation. I have attempted to make an educated guess as to the fastest methods to use in 'SPEED' and 'SPEEDOMP' modes, but results can vary quite a bit from machine to machine depending on MATLAB version, computer in use, compiler used, and number of processors. For example, OpenMP with MSVC 9 seems to work great on a 32-bit WinXP Intel Core 2 Duo platform under MATLAB version R2010a, but can perform miserably on the same machine under MATLAB version R2006b. There is apparently some type of OpenMP incompatibility or interference with older versions of MATLAB. Thus, based on these testing experiences, I fully expect that my choices will not always be optimal for speed across all platforms. So you may need to experiment some with any specific calculation to determine which method is fastest. If you do happen to find that MATLAB is significantly faster for any particular calculation, I would appreciate getting an e-mail (see last section) letting me know the particulars so that I can improve **MTIMESX**. You should be very cautious in using 'LOOPSOMP' mode since this may force **MTIMESX** to use OpenMP inappropriately on small size matrices and can in those cases easily *increase* the running time by a factor of 10x or more.

3) Multi-Threading

*Which calculations in **MTIMESX** are Multi-Threaded?*

It depends. There are basically three levels of multi-threading in **MTIMESX**:

1) BLAS Routines.
2) Multi-threading obtained on the basic C loops because the optimizing compiler you are using recognized the parallelism in the calculation and multi-threaded it.
3) Multi-threading obtained explicitly with the OpenMP #pragma parallel constructs applied to the basic C loops.

The BLAS routines used by **MTIMESX** may or may not be multi-threaded. This will depend on your particular version of MATLAB, the BLAS library you linked with, and whether or not you are running MATLAB in a multi-threaded mode. Newer versions of MATLAB are more likely to have the BLAS routines multi-threaded. Whenever **MTIMESX** calls a BLAS routine you will get the benefit of the multi-threading if the BLAS routine is multi-threaded. **MTIMESX** has no control over this. You will either get it or not get it depending on the BLAS library and the MATLAB mode. For example, using the 'OMP_SET_NUM_THREADS' directive (see next section) will have no effect whatsoever on whether or not the BLAS routines called are multi-threaded.

The C compiler you use to compile **MTIMESX** may be an optimizing compiler that is able to recognize existing parallelism in the C loops used by **MTIMESX**. If that is the case, then the compiler might produce multi-threaded code for these loops without the code specifically requesting it. There is no direct control over this at run-time when you call **MTIMESX**. You either get it or you don't depending on how the compiler compiled the C loops. The C loops that **MTIMESX** uses are very basic (e.g., a loop to multiply an array by a scalar) so a good compiler will easily be able to recognize the parallelism and multi-thread the code. However, this is not guaranteed and it needs to be stressed that the speed of the resulting compiled code can vary greatly depending on your computer and the compiler you use. Generally speaking, the calculations that are most likely to benefit from this implicit multi-threading in 'LOOPS' or 'SPEED' mode are scalar multiplies, dot products, outer products, and certain matrix * vector operations that can be performed with a series of dot products.

MTIMESX also has some explicit multi-threaded code using OpenMP #pragma parallel constructs. In order to use the OpenMP parallel processing environment, your compiler must support OpenMP. Note that the LCC compiler supplied with MATLAB does not support OpenMP compiling. Neither do the Microsoft Visual C/C++ compilers prior to version 8 (2005) or any of the Standard versions of the Microsoft Visual C/C++ compilers (you need the Professional version to get OpenMP support). The latest gcc and Intel compilers do support OpenMP. Although Microsoft Visual C/C++ versions may support OpenMP compiling, the associated mexopts.bat files supplied with MATLAB for these compilers do not have the /openmp compiler option selected to enable OpenMP compiling. In this particular case, the mtimesx_build function will automatically create a copy of the mexopts.bat file and add the /openmp compiling option to get this capability enabled. The operations that are explicitly multi-threaded using OpenMP #pragma parallel constructs are (unless otherwise noted, results may not match MATLAB exactly):

scalar * array (results will match MATLAB exactly)
vector outer product (results will match MATLAB exactly)
vector inner product (i.e., dot product)
vector * matrix (using a series of dot product calculations)
matrix' * vector (using a series of dot product calculations)
matrix.' * vector (using a series of dot product calculations)
(4x4 or smaller matrix) * (4x1 or smaller vector) (large nD case, no singleton expansion)
(1x4 or smaller vector) * (4x4 or smaller matrix) (large nD case, no singleton expansion)

Not all of the operations in the above list are always multi-threaded. The actual decision to use a series of BLAS calls vs a single C code loop vs an OpenMP multi-threaded C code loop depends on the variable sizes and on whether or not there are complex variables or conjugates involved in the computation. You can use the 'DEBUG' directive to see exactly what method is actually used in calculations, and you can use the 'OMP_GET_NUM_THREADS' directive to see how many threads were actually used in the most recent **MTIMESX** calculation. Note that it is possible for a calculation in 'LOOPS' mode to run faster than a calculation in 'LOOPSOMP' mode if the optimizing compiler has already multi-threaded the underlying C loops.

*Is **MTIMESX** Itself Thread-Safe?*

The short answer is no, **MTIMESX** is not thread-safe. The there are three main reasons:

1) **MTIMESX** keeps track of the calculation mode related settings in global variables. This is necessary so that the user does not have to enter the calculation mode, number of threads to use, etc. with each and every call. I felt that the convenience of this feature out-weighed the thread-safety issue, so that is how I programmed it. Specifically, the global variables are used to remember the following:

Calculation mode
Number of threads to request for OpenMP methods
Debug print flag
Number of OpenMP threads actually used in the previous calculation

2) **MTIMESX** makes calls to the BLAS library, which may not be thread-safe.

3) **MTIMESX** makes MATLAB API calls, which may not be thread-safe.

That being said, you *might* be able to use **MTIMESX** in a thread-safe manner in your m-code if you adhere to the following restrictions:

1) Do not change any calculation mode related settings in any of your threads. That is, do not use any of the following directives (see next section for details of these directives):

'BLAS'
'LOOPS'
'LOOPSOMP'
'MATLAB'
'SPEED'
'SPEEDOMP'
'DEBUG'
'NODEBUG'
'OMP'
'OMP_SET_NUM_THREADS'
'OMP_SET_NUM_THREADS(N)'

2) Ensure you are linking with a thread-safe BLAS library.

3) Ensure that the MATLAB API functions are thread-safe. (this may not be possible)

In addition, since the number of threads actually used in a call will be subject to a race condition among your threads, the returned value from this directive after your threads complete will be unreliable since there will be no way to know which thread generated the value:

'OMP_GET_NUM_THREADS'

4) Syntax

MTIMESX has full support for transpose ('T'), conjugate transpose ('C'), and conjugate ('G') pre-operations on the input variables. The general syntax to perform the operation op(A) * op(B) is (arguments in brackets [ ] are optional and case insensitive):

```
mtimesx( A [,transa] ,B [,transb] [,directive] [,directive] ... )
```

Where:

A, B = a single, double, or double sparse scalar, vector, matrix, or nD-array

And where `transa`, `transb`, and `directive` are the optional inputs:

`transa, transb` = A character indicating a pre-operation on A and B:
The pre-operation can be any of:
'N' or 'n' = No pre-operation (the default if trans_ is missing)
'T' or 't' = Transpose
'C' or 'c' = Conjugate Transpose
'G' or 'g' = Conjugate (no transpose)

`directive` = A character string indicating calculation mode:
'BLAS'
'LOOPS'
'LOOPSOMP'
'MATLAB'
'SPEED'
'SPEEDOMP'

Note: The 'N', 'T', and 'C' have the same meanings as the direct inputs to the BLAS routines. The 'G' input has no direct BLAS counterpart, but was relatively easy to implement in **MTIMESX** and saves time (as opposed to computing conj(A) or conj(B) explicitly before calling **MTIMESX**).

Note: If you combine double sparse and single inputs, **MTIMESX** will convert the single input to double since MATLAB does not support a single sparse result. If you combine sparse inputs with full nD (n > 2) inputs, **MTIMESX** will convert the sparse input to full since MATLAB does not support a sparse nD result. One exception is a double sparse scalar times a single. In this case **MTIMESX** will convert the double sparse scalar to a single and return a full single result.

Whenever you change modes or requested number of threads, the changes affect the current calculation (if any) *and* all future calculations until other directives are issued. If you clear **MTIMESX** from memory then the next time it runs it will start up in 'MATLAB' mode. You can also issue the following OpenMP related directives. If **MTIMESX** is compiled with a non-OpenMP compliant compiler then these directives are either ignored or produce a fixed result.

'OMP'
Same as issuing the directives 'SPEEDOMP' and 'OMP_SET_NUM_THREADS'

'OMP_SET_NUM_THREADS'
Sets the maximum number of requested threads for OpenMP to the number of processors available.

'OMP_SET_NUM_THREADS(N)'
N is any expression evaluated in caller workspace.
Sets the maximum number of requested threads for OpenMP to N.
Uses the OpenMP method #pragma omp parallel num_threads(N) to request the number of threads to use for OpenMP parallel code. Temporarily creates a variable called OMP_SET_NUM_THREADS in the caller workspace which is used to evaluate the expression N.

'OMP_GET_NUM_PROCS'
Must be the only argument.
Returns the number of processors available.
Returns the value of a OpenMP omp_get_num_procs( ) call.
This directive can be used with non-OpenMP compiled code also.

'OMP_GET_NUM_THREADS'
Must be the only argument.
Returns the number of OpenMP threads that were actually used for the most recent **MTIMESX** calculation. That is, the value of the OpenMP function omp_get_num_threads() called from within the OpenMP parallel section. If the most recent **MTIMESX** calculation did not use OpenMP then the value returned will be 0. This directive can be used with non-OpenMP compiled code also.

'OMP_GET_MAX_THREADS'
Must be the only argument.
Returns the value you set with the OMP_SET_NUM_THREADS directive.
This directive can be used with non-OpenMP compiled code also.

'OMP_GET_WTICK'
Must be the only argument.
Returns the number of seconds between processor ticks.
Returns the result of an OpenMP omp_get_wtick( ) call.

'OMP_GET_WTIME'
Must be the only argument.
Returns value in seconds of time elapsed from some point.
Returns the result of an OpenMP omp_get_wtime( ) call.

There are also additional directives available that do not affect calculations:

'LOGO' (reproduces **MTIMESX** graphic logo)
'DEBUG' (gives feedback on exactly which method is used for a calculation)
'NODEBUG' (turns off 'DEBUG' mode)
'HELP' (prints the **MTIMESX** basic help text from the file mtimesx.m)
'COMPILER' (returns a string with the name of the compiler used)
'OPENMP' (returns 1 if compiled with OpenMP features enabled, 0 otherwise)

You can also call **MTIMESX** with just directives to get or set the calculation mode information etc. without actually doing any calculation. e.g.,

```
M = mtimesx( [directive] [,directive] ... )
```

Where:
`M` = The result of the call. When setting a mode, or if no argument is present, **MTIMESX** simply returns the current calculation mode.
`directive` = Any of the directives listed previously.

Examples:

```
C = mtimesx(A,B)              % performs the calculation C = A * B
C = mtimesx(A,'T',B)          % performs the calculation C = A.' * B
C = mtimesx(A,B,'g')          % performs the calculation C = A * conj(B)
C = mtimesx(A,'c',B,'C')      % performs the calculation C = A' * B'
mtimesx('SPEED')              % sets future calculations to 'SPEED' mode
C = mtimesx(A,'g',B,'c')      % performs the calculation C = conj(A) * B'
C = mtimesx(A,B,'MATLAB')     % performs the calculation C = A * B in MATLAB
                              % mode. All future calculations use MATLAB mode too
mtimesx(C,A,'T',B,'G')        % performs the calculation C = C + A.' * conj(B)
```

5) Multi-Dimensional Support

MTIMESX supports nD inputs. For these cases, the first two dimensions specify the matrix multiply involved. The remaining dimensions are duplicated and specify the number of individual matrix multiplies to perform for the result. i.e., **MTIMESX** treats these cases as arrays of 2D matrices and performs the operation on the associated pairings. The 2D slices can be 2D matrices, 1D vectors, or 1D scalars. For the 1D scalar case the usual scalar multiply is done, not a matrix multiply. For example,

If A is (2,3,4,5) and B is (1,1,4,5), then
mtimesx(A,B) would result in a C(2,3,4,5), where C(:,:,i,j) = A(:,:,i,j) * B(:,:,i,j), i=1:4, j=1:5

which would be equivalent to the MATLAB m-code:

```
C = zeros(2,3,4,5);
for m=1:4
    for n=1:5
        C(:,:,m,n) = A(:,:,m,n) * B(:,:,m,n);
    end
end
```

Another example:

If A is (2,3,4,5) and B is (3,6,4,5), then
mtimesx(A,B) would result in C(2,6,4,5), where C(:,:,i,j) = A(:,:,i,j) * B(:,:,i,j), i=1:4, j=1:5

which would be equivalent to the MATLAB m-code:

```
C = zeros(2,6,4,5);
for m=1:4
    for n=1:5
        C(:,:,m,n) = A(:,:,m,n) * B(:,:,m,n);
    end
end
```

The first two dimensions must conform using the standard matrix multiply rules (or be scalar) taking the transa and transb pre-operations into account, and dimensions 3:end must match exactly or be singleton (equal to 1). If a dimension is singleton then it is virtually expanded to the required size (i.e., equivalent to a repmat operation to get it to a conforming size but without the actual data copy). This is equivalent to a bsxfun capability for matrix multiplication. For example:

If A is (2,3,4,5) and B is (3,6,1,5), then
mtimesx(A,B) would result in C(2,6,4,5), where C(:,:,i,j) = A(:,:,i,j) * B(:,:,1,j), i=1:4, j=1:5

which would be equivalent to the MATLAB m-code:

```
C = zeros(2,6,4,5);
for m=1:4
    for n=1:5
        C(:,:,m,n) = A(:,:,m,n) * B(:,:,1,n);
    end
end
```

When a transpose (or conjugate transpose) is involved, the first two dimensions are transposed in the multiply as you would expect. For example:

If A is (3,2,4,5) and B is (3,6,4,5), then
mtimesx(A,'C',B,'G') gives C(2,6,4,5), where C(:,:,i,j) = A(:,:,i,j)' * conj(B(:,:,i,j)), i=1:4, j=1:5

which would be equivalent to the MATLAB m-code:

```
C = zeros(2,6,4,5);
for m=1:4
    for n=1:5
        C(:,:,m,n) = A(:,:,m,n)' * conj( B(:,:,m,n) );
    end
end
```

If A is a scalar (1,1) and B is (3,6,4,5), then
mtimesx(A,'G',B,'C') would result in C(6,3,4,5), where C(:,:,i,j) = conj(A) * B(:,:,i,j)', i=1:4, j=1:5

which would be equivalent to the MATLAB m-code:

```
C = zeros(6,3,4,5);
for m=1:4
    for n=1:5
        C(:,:,m,n) = conj(A) * B(:,:,m,n)';
    end
end
```

If A is (3,3) and B is (3,1,1000000), then
mtimesx(A,'T',B) would result in C(3,1,1000000), where C(:,:,i) = A.' * B(:,:,i), i=1:1000000

which would be equivalent to the MATLAB m-code:

```
C = zeros(3,1,1000000);
for m=1:1000000
    C(:,:,m) = A.' * B(:,:,m);
end
```

For this particular case, note that MTIMESX provides inline C loops code for this calculation since the first two dimensions of A and B are no greater than 4. So a fast way to perform this calculation would be:

```
mtimesx('LOOPS');
C = mtimesx(A,'T',B);
```

And, since this matches one of the multi-threaded forms, this calculation can be speeded up even further with use of the OpenMP methods:

```
mtimesx('SPEEDOMP','OMP_SET_NUM_THREADS(4)');
C = mtimesx(A,'T',B);
```

Finally, if you want MTIMESX to pick the number of threads to be the maximum number of processors available automatically instead of hardcoding this in your code, you can do the following:

```
mtimesx('OMP');
C = mtimesx(A,'T',B);
```

For the last form listed above, recall that the 'OMP' directive is the same thing as issuing the 'SPEEDOMP' directive and the 'OMP_SET_NUM_THREADS' directive simultaneously, and if you don't give the number of threads to use (as we didn't in this example) MTIMESX will pick the number of threads to use automatically as the number of processors available. Caution: Some processors have twice as many logical processors as physical processors (e.g., the latest Intel i3, i5, i7 chips and the latest AMD chips). On these machines it is best to set the number of threads to the actual physical number of processors, not the logical number.

6) Other Types

Types other than single or double will not generate errors directly from **MTIMESX**. Instead, they are simply passed on through to the MATLAB intrinsic ***mtimes*** function. If MATLAB supports the operation then that result will be returned, otherwise a MATLAB generated error will result. Note particularly that MATLAB does not support any multi-dimensional capability or bsxfun-like singleton expansion capability for matrix multiplication, so if you attempt this with types other than single or double MATLAB will generate an error.

7) Philosophy

The primary motivation for **MTIMESX** is an improvement on the MATLAB intrinsic function ***mtimes*** for speed whenever possible and to directly support multi-dimensional matrix multiplication without creating 2D array slice copies. The amount of improvement you will get with **MTIMESX** is highly dependent on your particular computer, C compiler, MATLAB version, and what calculation mode **MTIMESX** is using. Older versions of MATLAB often see the most improvement for 'SPEED' mode, while later versions of MATLAB may see little to no benefit in the same cases. Later versions of MATLAB seem to benefit the most from OpenMP calculations, while earlier versions of MATLAB actually seem to impede or interfere with OpenMP calculations. **MTIMESX** directly supports multi-dimensional operations, treating nD arrays as arrays of 2D arrays. See the following test run m-files for comprehensive results on your particular setup (CAUTION: These tests can take several hours to run … best to do it overnight:

```
mtimesx_test_ddequal.m      % A test program for (double) * (double) equality
mtimesx_test_ddspeed.m      % A test program for (double) * (double) speed
mtimesx_test_ssequal.m      % A test program for (single) * (single) equality
mtimesx_test_ssspeed.m      % A test program for (single) * (single) speed
mtimesx_test_dsequal.m      % A test program for (double) * (single) equality
mtimesx_test_dsspeed.m      % A test program for (double) * (single) speed
mtimesx_test_sdequal.m      % A test program for (single) * (double) equality
mtimesx_test_sdspeed.m      % A test program for (single) * (double) speed
mtimesx_test_nd.m           % A test program for multi-dimensional matrix multiplies
```

MTIMESX is also an example of calling BLAS routines from a C-mex routine (non C programmers need not worry: the routine is self-building ... you don't have to know anything about C or mex to use **MTIMESX**... just skip this section). The source code is generously commented so that you can (hopefully) follow why a certain routine was used, or why custom code was used instead of a BLAS routine. Creating a matrix and scalar multiply routine for the BLAS examples gives the opportunity to provide an example that has a practical use and at the same time allows direct comparison with MATLAB results.

MTIMESX is not an attempt to necessarily reproduce the exact BLAS calling sequences that MATLAB uses. For some of the general matrix-matrix and matrix-vector operations, I would not be surprised to find that **MTIMESX** was using the exact same BLAS calling sequences, but this is not the goal and is not the claim. Indeed, this would have been a conflict with the speed goal in many cases. Keep this important point in mind ... **MTIMESX** will not in general reproduce MATLAB results exactly in any mode other than 'MATLAB' mode because that is not the main goal of those modes. But these other modes will produce results that are just as accurate as the MATLAB result ... and in some special cases will run many times faster than the MATLAB instinsic function ***mtimes***. For certain 2D operations involving complex conjugates **MTIMESX** can run 3x - 4x faster. For certain nD operations **MTIMESX** can run 50x - 100x faster than the equivalent MATLAB loop (yes, you read that correctly, fifty to one hundred times faster).

8) BLAS Routines Used

The BLAS routines used are DAXPY, DDOT, DGER, DGEMV, DGEMM, DSYRK, and DSYR2K for double variables, and SAXPY, SDOT, SGER, SGEMV, SGEMM, SSYRK, and SSYR2K for single variables. These routines are (description taken from www.netlib.org):

DAXPY and SAXPY:

* Computes constant times a vector plus a vector.

DDOT and SDOT:

* Forms the dot product of two vectors.

DGER and SGER: Performs the rank 1 operation

* A := alpha*x*y' + A,
*
* where alpha is a scalar, x is m element vector, y is n element vector and A is m by n matrix.

DGEMV and SGEMV: Performs one of the matrix-vector operations

* y := alpha*A*x + beta*y, or y := alpha*A'*x + beta*y,
*
* where alpha and beta are scalars, x and y are vectors and A is an m by n matrix.

DGEMM and SGEMM: Performs one of the matrix-matrix operations

* C := alpha*op(A)*op(B) + beta*C,
*
* where op(X) is one of
*
* op(X) = X or op(X) = X',
*
* alpha and beta are scalars, and A, B and C are matrices, with op(A) an m by k matrix,
* op(B) a k by n matrix and C an m by n matrix.

DSYRK and SSYRK: Performs one of the symmetric rank k operations

* C := alpha*A*A' + beta*C,
*
* or
*
* C := alpha*A'*A + beta*C,
*
* where alpha and beta are scalars, C is an n by n symmetric matrix and A is an n by k
* matrix in the first case and a k by n matrix in the second case.

DSYR2K and SSYR2K: Performs one of the symmetric rank 2k operations

* C := alpha*A*B' + alpha*B*A' + beta*C,
*
* or
*
* C := alpha*A'*B + alpha*B'*A + beta*C,
*
* where alpha and beta are scalars, C is an n by n symmetric matrix and A and B are n by k
* matrices in the first case and k by n matrices in the second case.

9) Supported Operations

Custom code is used to minimize memory access times and take full advantage of symmetric cases in 'SPEED' and 'SPEEDOMP' modes, resulting in substantial time savings in some cases, particularly for complex operations on older versions of MATLAB. Hence you will find extensive custom code for these cases, particularly the scalar multiplies and contiguous dot products.

MTIMESX supports the following operations (arguments in either order):

1D vector * scalar
2D matrix * scalar
nD array * scalar
vector * vector (inner or outer product)
matrix * vector
matrix * matrix
nD matrix * nD matrix (treated as arrays of 2D matrices with singleton expansion)

Nearly all operations support nD arrays with the understanding that only the first two dimensions are used in any transpose operations. The main restriction on nD arrays comes from MATLAB itself ... you can't combine sparse and nD (n > 2) variables and get a sparse result because MATLAB does not support a sparse nD result (**MTIMESX** will return a full result).

10) Speed Improvements

It bears repeating that the **MTIMESX** timing results are highly dependent on your particular computer, C compiler, and version of MATLAB. **MTIMESX** uses custom code in many places instead of BLAS calls in an effort to minimize memory access times. The effectiveness of this custom code can vary quite a bit when compared directly to the intrinsic MATLAB ***mtimes***, particularly with respect to the MATLAB version involved and the number or cores used by the BLAS routines. Nevertheless, here are some sample timings with huge variables (e.g., 100MB) using R2008a with the lcc compiler on 32-bit Intel Core 2 Duo WinXP machine:

SPEED MODE

(% faster mtimesx over mtimes)	Real*Real	Real*Cplx	Cplx*Real	Cplx*Cplx
conj(Scalar) * Vector.'	-3%	60%	-2%	60%
conj(Vector) * Scalar.'	-4%	-3%	58%	51%
conj(Array) * Scalar.'	76%	36%	83%	70%
conj(Vector) i Vector.'	-4%	7%	172%	169%
conj(Vector) o Vector.'	-8%	6%	6%	26%
conj(Vector) * Matrix.'	-0%	0%	0%	-0%
conj(Matrix) * Vector.'	-0%	-0%	360%	142%
conj(Matrix) * Matrix.'	-0%	0%	3%	2%

Many of the operations offer no speed improvement, but some of them offer a substantial improvement. For example, the table above shows that **MTIMESX** is 360% faster than MATLAB ***mtimes*** for the computation conj(complex matrix) * (real vector).' in 'SPEED' mode. A quick examination of the table reveals that this is not an isolated incident. There are several calculations in the sample table above where **MTIMESX** is 10's and 100's percent faster than MATLAB ***mtimes***. Similar speed improvements are obtained with other calculations not shown.

Double sparse matrix operations are supported, but not always directly. For matrix * scalar operations, custom code is used to produce a result that minimizes memory access times. All other operations, such as matrix * vector or matrix * matrix, or any operation involving a transpose or conjugate transpose, are obtained with calls back to the MATLAB intrinsic ***mtimes*** function. Thus for most non-scalar sparse operations, **MTIMESX** is simply a thin wrapper around the intrinsic MATLAB function ***mtimes*** and you will see no speed improvement.

How does **MTIMESX** get speed improvements over the MATLAB intrinsic ***mtimes*** function?

By reducing memory access times and taking full advantage of conjugate and symmetric cases. The MATLAB intrinsic ***mtimes*** function is excellent, but it doesn't optimize memory access for all operation combinations and sometimes uses memory inefficient algorithms to calculate the result. This is particularly true for older versions of MATLAB. For many cases, what I have done is to call a different set of BLAS routines or create custom code. The MATLAB sequence of calling the BLAS routines for some of these operations, particularly for complex operations, involves accessing the input variables twice. Custom code can sometimes avoid this and access the input variables only once, reducing the total memory access time to perform the operation. Some examples:

Example 1: Large complex matrix conjugate transposed times a "thin" complex matrix.

```
>> A=rand(3000)+rand(3000)*1i;
>> B=rand(3000,3)+rand(3000,3)*1i;
>> tic;A' * B;toc
Elapsed time is 0.440585 seconds.
>> tic;mtimesx(A,'c',B,'SPEED');toc
Elapsed time is 0.217146 seconds.
>> isequal(A'*B,mtimesx(A,'c',B,'SPEED'))
ans =
    0
```

So how did **MTIMESX** beat ***mtimes*** for this operation? Since the first matrix A is transposed, the actual memory access for this matrix can be done by physical columns, not rows, and the physical column elements are contiguous in memory. Custom code was written to take advantage of this fact, and performs the operation as a series of complex dot product type operations on the columns of A and B. The custom code does the real and imaginary part of the dot product result all within the same loop, so the input array contents are accessed only once. The MATLAB intrinsic ***mtimes*** apparently uses a series of BLAS calls, forcing the input array memory to be accessed twice. This probably accounts for the 2x speed improvement. Because the underlying calculations are done slightly differently, the answers are slightly different. But the **MTIMESX** result is just as accurate as the MATLAB intrinsic ***mtimes*** result. **MTIMESX** uses a custom C implementation of a basic dot product algorithm using loop unrolling to gain a speed and accuracy improvement. A loop block size of 10 for the unrolling was optimum in testing, so that is what is used in the custom code. Custom dot product code, rather than calling the BLAS routines DDOT or SDOT directly, is necessary to avoid accessing the input variables twice.

Example 2: Large complex vector outer product.

```
>> A = rand(3000,1)+rand(3000,1)*1i;
>> B = rand(1,3000)+rand(1,3000)*1i;
>> tic;A*B;toc
Elapsed time is 0.467077 seconds.
>> tic;mtimesx(A,B,'SPEED');toc
Elapsed time is 0.249749 seconds.
>> isequal(A*B,mtimesx(A,B))  % 'SPEED' option not needed since it carries over from the
previous call
ans =
    1
```

Again, the speed improvement is the result of custom code in **MTIMESX** that avoids accessing the input variables twice. Here the results are exactly the same.

Example 3: Large complex sparse matrix times a complex scalar on older version of MATLAB.

```
>> A = sprand(10000,10000,.1);
>> A = A + A*1i;
>> B = rand + rand*1i;
>> mtimesx('SPEED');  % This setting will carry forward to all subsequent calls
>> tic;conj(a)*b;toc
Elapsed time is 7.740495 seconds.
>> tic;mtimesx(a,'G',b);toc
Elapsed time is 0.520425 seconds.
>> isequal(conj(A)*B,mtimesx(A,'G',B))
ans =      1
```

The dramatic speed improvement here is because **MTIMESX** treats the operation as a scalar times a 1D array. No special sparse matrix multiply code is needed. Apparently the MATLAB intrinsic ***mtimes*** function calls special sparse matrix multiply algorithms for this. Here the results are exactly the same. I will quickly point out that this example is for an older version of MATLAB. The latest versions of MATLAB have improved this calculation but even in this case **MTIMESX** is nearly 400% faster (as compared to nearly 1500% faster in the above example).

11) List of Included Files

mtimesx.m	**1.40, 4 October 2010**	**% Used for the help text and first-time building**
mtimesx_build.m	**1.40, 4 October 2010**	**% Building mtimesx.mexext for PC Windows**
mtimesx_sparse.m	**1.00, 27 September 2009**	**% Used for calling back to MATLAB for multiplies**
mtimesx_test_ddequal.m	**1.00, 27 September 2009**	**% A test program for (double) * (double) equality**
mtimesx_test_ddspeed.m	**1.00, 27 September 2009**	**% A test program for (double) * (double) speed**
mtimesx_test_ssequal.m	**1.00, 27 September 2009**	**% A test program for (single) * (single) equality**
mtimesx_test_ssspeed.m	**1.00, 27 September 2009**	**% A test program for (single) * (single) speed**
mtimesx_test_dsequal.m	**1.00, 27 September 2009**	**% A test program for (double) * (single) equality**
mtimesx_test_dsspeed.m	**1.00, 27 September 2009**	**% A test program for (double) * (single) speed**
mtimesx_test_sdequal.m	**1.00, 27 September 2009**	**% A test program for (single) * (double) equality**
mtimesx_test_sdspeed.m	**1.00, 27 September 2009**	**% A test program for (single) * (double) speed**
mtimesx_test_nd.m	**1.40, 4 October 2010**	**% A test program for multi-dimensional multiplies**
mtimesx.c	**1.40, 4 October 2010**	**% C source code for mexFunction interface**
mtimesx_RealTimesReal.c	**1.40, 4 October 2010**	**% C source code for matrix multiplication logic**
mtimesx_20101004.pdf	**1.40, 4 October 2010**	**% The file you are currently reading**

For PC Windows, **MTIMESX** attempts to be self-building. When you first run **MTIMESX**, the file that will execute will be mtimesx.m. If this file executes, then that means that the mex dll routine has not been built yet. So the only code in mtimesx.m is a call to mtimesx_build.m to build the mex dll routine and then call it. mtimesx_build will try to autodetect if the machine is PC Windows. If it is, it will try to self-build the mex dll routine. If not, it will exit with brief instructions on how to manually compile the mex dll routine. There is also some anecdotal advice for building **MTIMESX** on non-Windows systems in the next section.

Once the dll building is complete, you will also have an additional file: mtimesx.mexext

From now on, whenever you invoke **MTIMESX** it is this file that actually executes. The mexext is replaced with the actual mex dll extension for your particular computer and operating system. For example, on 32-bit windows the extension is mexw32 for later versions of MATLAB.

Sparse matrix multiply operations and transpose operations are not directly supported in the C code. Instead, a call-back to MATLAB is done to perform these using the mtimesx_sparse.m function. So you will not see any difference between **MTIMESX** and the MATLAB ***mtimes*** function for these cases. Where you *will* see a difference is in the op(scalar) * op(sparse) cases, where custom code is used to minimize memory access in **MTIMESX**. These cases can yield a significant speed improvement over the MATLAB ***mtimes*** function for older versions of MATLAB or newer versions of MATLAB if at least one of the variables is complex.

12) Testing

The only configurations tested by the author for **MTIMESX** are PC 32-bit Windows XP with various MATLAB versions R2006b - R2010a and the lcc compiler and Microsoft Visual C/C++ 8 (2005) and Microsoft Visual C/C++ 9 (2008) compilers. The author would like to solicit help from the community in getting **MTIMESX** (and the self-building code in mtimesx_build.m) to work under other configurations.

Other configurations tested by the MATLAB community:

User: Fabio Veronese
System: Gentoo Linux (64bit) 2.6.33 on amd64 system
Compiler: gcc 4.3.* (not the deprecated 4.2 as suggested by Mathworks)
Compile Command(s):

```
mex('-DDEFINEUNIX','-largeArrayDims', '-lmwblas','mtimesx.c')
```

User: Joshua Dillon
System: Ubuntu / linux
Compiler: (unknown)
Compile Command(s):

```
libblas='/usr/lib/libblas.so';
if exist(libblas,'file')~=2
   system('sudo aptitude install libblas-dev');
end
mex('CFLAGS=-std=c99-fPIC','-DDEFINEUNIX','-largeArrayDims','-lmwblas','mtimesx.c');
```

User: Jon
System: Debian, 64-bit
Compiler: (unknown)
Compile Command(s):

```
mex CFLAGS="\$CFLAGS -std=c99" -DDEFINEUNIX -largeArrayDims -lmwblas mtimesx.c
```

User: Val Schmidt
System: Snow Leopard 10.6, 64-bit
Compiler: gcc
Compile Command(s):

To do so I edited
/Applications/MATLAB_R2009b.app/bin/gccopts.sh
In the section for "maci64" I changed "SDKROOT" and "MACOSX_DEPLOYMENT_TARGET" to reflect a change in OS from 10.5 to 10.6.
Then I selected this opts file with
mex -setup
selecting "1" at the prompt.
Then I compiled everything with:
mex -v -DDEFINEUNIX -largeArrayDims mtimesx.c -lmwblas -lmwlapack

13) Upgrades

Future upgrades planned:

- Bug fixes, of course.
- More extensive use of OpenMP code (in work, possibly next release)
- More extensive test routines.
- Expanded documentation and examples.
- Support for 64-bit and linux and mac machines, particularly for self building. But this will depend on other users willing to supply this code to me, since I do not have access to these machines for development or testing. (in work, possibly next release)
- Support for in-place operations, particularly with direct access to BLAS routine capabilities (in work, possibly next release)
- Direct access to the BLAS sub-matrix capability and scalar multipliers.
- Cell arrays used for block matrix multiples

Future upgrades that I *may* consider depending on my time availability and the popularity of **MTIMESX**:

- More speed improvements to the current code if I can come up with better algorithms. Does anybody out there have a fast matrix transpose algorithm? Please contact me if you do.
- Custom code for (single * double) and (double * single) multiplies. As currently written, **MTIMESX** simply does a conversion of the input argument(s) first and then does the multiply. This can run quite a bit slower than the MATLAB intrinsic ***mtimes*** function.
- Custom code for sparse transpose and conjugate transpose operations. The goal here again would be to see if a speed improvement can be achieved. Implementation will depend on my ability to find an efficient sparse matrix transpose algorithm.
- Custom code for sparse matrix times sparse vector operations. The goal would be to see if a speed improvement can be achieved. Implementation will depend on my ability to find some spare time to design the algorithm.
- Support for older versions of MATLAB, particularly versions prior to 7. My guess is that some of the MATLAB code and/or API functions I used might not be compatible. Again, this will depend on other users willing to help modify the code, since I do not have access to these older versions of MATLAB for development or testing.

14) MTIMESX Logo

The MTIMESX logo was inspired by a 3D graphing example in the MATLAB documentation and can be generated with the following command:

```
mtimesx('logo')
```

15) Contact the Author

Feel free to post items of general interest to other users (bug reports, performance data, questions about usage or optimizations, etc) directly on the FEX of course. But if you have modified the code for your version of MATLAB (older version, non-PC machine, non-supported C compiler, etc.) please feel free to contact me directly and I will try to incorporate them into future **MTIMESX** upgrades. You can reach me at

a#lassyguy%ho$mail_com (replace # with k, % with @, $ with t, _ with .)

James Tursa

16) Acknowledgements

The omp_get_num_procs() function for non-OpenMP implementations is courtesy of Dirk-Jan Kroon and is based on his FEX submission maxNumCompThreads.

17) Release Notes

2009/Sep/27 --> 1.00
Initial Release.

2009/Dec/10 --> 1.11
Fixed bug for empty transa & transb inputs.

2010/Feb/23 --> 1.20
Fixed bug for dgemv and sgemv calls.

2010/Aug/02 --> 1.30
Added (nD scalar) * (nD array) capability.
Replaced buggy mxRealloc with custom routine.

2010/Oct/04 --> 1.40
Added OpenMP support for custom code.
Expanded sparse * single and sparse * nD support.
Fixed (nD complex scalar)C * (nD array) bug.