System Guide

Automatic Mexfunctions Generator 1.0

By Atiyah Elsheikh
Mathemodica.com
atiyah.elsheikh@mathemodica.com

Table of Contents

1. Introduction to MEX API

To understand the capabilities of AMG, a minimal understanding of basic terminologies of MEX-files is required. This section provides a quick overview of ***MEX-files*** (MATLAB Executable files) and their functionalities, that allow MATLAB programs to interface C/Fortran subroutines.

a) Terminology

MATLAB provides an interface to routines written in C or Fortran. Using this capability, C or Fortran subroutines can be called from MATLAB, as if they were MATLAB functions. The user or developer won't be aware that the function is written in C/Fortran. MATLAB-callable C and Fortran programs are referred to as ***Mexfiles*** (MATLAB Executable Files). A ***Mexfile*** is a routine that results from the compilation of a kernel C/Fortran routine and a corresponding gateway function. Inside the gateway function, the high-level data structures of MATLAB objects are created from variables of basic data types. These variables can then be passed as parameters by a call to the desired kernel C/Fortran routine. Such a gateway function is referred to as ***Mexfunction***. By compiling the Mexfunction and the kernel routine, a ***Mexfile*** is produced.

Mexfiles have the following applications:

- Large pre-existing C/Fortran programs and standard library routines can be called from MATLAB without being rewritten as M-files
- Bottleneck computations that do not run fast enough in MATLAB can be recoded in C or Fortran for efficiency

Moreover, there are some other, not less useful applications, such as:

- Providing interactive matrix construction and visualization possibilities by MATLAB in developing and testing phases of a numerical application written in C/Fortran
- Creating a shared memory parallelzed environment for Matlab by enabling parallelized C/Fortran subroutines to be called within Matlab

b) Mexfunctions

A Mexfunction has always the following prototype:

```
mexFunction(int nlhs, mxArray *plhs[ ],
            int nrhs, const mxArray *prhs[ ])
```

where

- "nlhs","nrhs" stand for the number of output and input parameters respectively
- "prhs" is an array of pointers to constant input parameters, where these objects are not supposed to be altered
- "plhs" is an array to the output parameters, which are supposed to be created inside the mexfunction.

All MATLAB's objects are of a single data type, namely a matrix object (declarations in MATLAB are not required). For example, an object is implicitly declared when it appears in the left hand side of an assignment. In mexfunctions, Input and output parameters are

represented as objects of type ***mxArray***.

Every MATLAB's object can be regarded as a matrix. The mxArray is a special structure that contains the MATLAB data of this matrix. It is the C representation of a MATLAB array:

The MATLAB language works only with a single object type, the mxArray. All MATLAB variables, including scalars, vectors, matrices, strings, cell arrays, and structures are stored as mxArrays. The mxArray declaration corresponds to the internal data structure that MATLAB uses to represent arrays. The MATLAB array is the C language definition of a MATLAB variable. The mxArray structure contains, among other things:

1. The MATLAB variable's name
2. Its data
3. Its dimensions
4. Its type
5. Whether it is a sparse matrix
6. Whether the variable is real or complex

If the variable contains complex numbers as elements, the MATLAB array includes vectors containing the real and imaginary parts. Matrices, or m-by-n arrays, that are not sparse are called ***full***. In the case of a full matrix, the mxArray structure contains parameters called "pr" and "pi". "pr" contains the real part of the matrix data; "pi" contains the imaginary data, if there is any. Both "pr" and "pi" are one-dimensional arrays of double-precision numbers. The elements of the matrix are stored in these arrays column-wise.

c) The mex.h interface

To handle mxArrays, MATLAB provides a special interface, "mex.h", to the user to handle and manipulate mxArrays. The most interesting functions are mx* functions which are used to access data of mxArrays, perform some memory management and create and destroy mxArrays. Some useful routines are:

Array creation	mxCreateNumericArray, mxCreateDoubleMatrix, mxCreate*
Array access	mxGetPr, mxGetPi, mxGetM, mxGetData, mxGet*
Array modification	mxSetPr, mxSetPi, mxSetData, mxSetDimensions, mxSet*
Memory management	mxMalloc, mxCalloc, mxFree, mexMakeMemoryPersistent, mxDestroyArray, memcpy
State checking	mxIsChar, mxIsComplex, mxIsSparse, mxIs*

"mxGetPr" and "mxGetPi" return pointers to their real part and imaginary part data respectively. To change the values in the array, it is necessary to directly change the value in the array pointed at, or use a function like memcpy from the C Standard Library. Other type of routines provided by mex.h are mex* functions. These can be used to perform operations back in MATLAB. Some useful routines are:

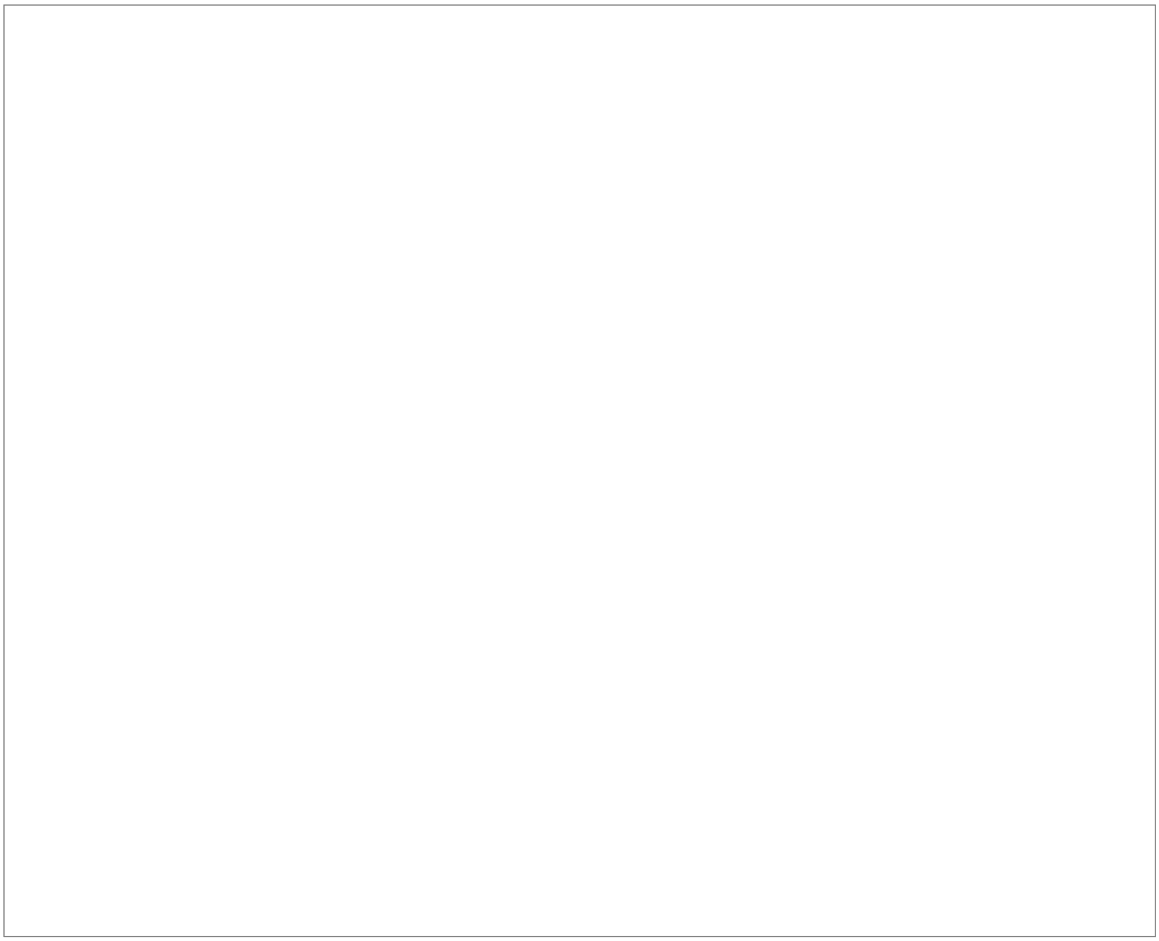
mexFunction	Entry point to C MEX-file
mexErrMsgTxt	Issue error message and return to MATLAB
mexEvalString	Execute MATLAB command in caller's workspace
mexCallMATLAB	Call MATLAB function or user-defined M-file or MEX-file
mexGetArray	Get copy of variable from another workspace

mexPrintf	ANSI C printf -style output routine
mexWarnMsgTxt	Issue warning message

d) Typical Implementation of a mexFunction

A typical implementation of a “mexFunction” can be as follows: Initially, a mexFunction begins with a phase for extracting the data address and characteristics of input parameters of the MATLAB function into pointers. This might be followed by a call to a computational routine where the kernel numerical computations are computed. Finally, a corresponding MATLAB object(s) for the output parameter(s) of the MATLAB calling function should be created and linked to the output variable(s) which has been computed in the kernel routine.

The following C MEX Cycle figure shows how inputs enter a MEX-file, what functions the gateway routine performs, and how outputs return to MATLAB.



e) Example

In the example below, we will create a MEX-file that takes any number of inputs and creates an equal number outputs. The output values will be twice the input values.

1. The first job of the MEX-file is to create mxArrays to hold the output data. Each output will be the same size as its corresponding input. This is done using `mxCreateDoubleMatrix` (creating a matrix to hold doubles), `mxGetM` (the number of rows the output should be), and `mxGetN` (the number of columns the output should be).
2. After the output mxArray is created, the only things left to do is to multiply the input by two, and to put that value into the output array. This is done with `mxGetPr` (get a pointer to the real part of the input data) and `mxMalloc` (the MEX version of the C function `malloc`).
3. The source code for this example looks as follows:

```
#include "mex.h"
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, const mxArray *prhs[]) {

        int i, j, m, n;
        double *data1, *data2;
        if (nrhs != nlhs)
                mexErrMsgTxt("The number of input and output arguments \
                              must be the same.");

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

                /* Find the dimension of the data */
                m = mxGetM(prhs[i]);
                n  = mxGetN(prhs[i]);


                /* Create an mxArray for the output */
                plhs[i] = mxCreateDoubleMatrix(m, n, mxREAL);

                /* Get the data passed in */
                data1 = mxGetPr(prhs[i]);

                /* Create an array for the output's data */
                data2 = (double *) mxMalloc(m*n * sizeof(double));

                /* Put data in the array */
                for (j = 0; j < m*n; j++)
                        data2[j] = 2 * data1[j];

                /* Assign the data array to the output array */
                mxSetPr(plhs[i], data2);

        }
}
```

f) Performance Note

All mxArray creation functions initialize the data elements to zero. On creation of large arrays, the initialization tends to void the processor caches and block the memory bus. Initialization is not really necessary if the target kernel subroutine doesn't require the outputs data to be initialized to 0. Avoiding initialization can improve the performance of MEX-files in many cases, where the parameters are large

arrays. A typical allocation of a new mxArray can be done as follows:

```
arr = mxCreateNumericArray(ndims,dims,mxDOUBLE_CLASS,mxREAL);
```

To avoid initialization, creation can be done as follows:

```
arr=mxCreateDoubleMatrix(0,0,mxREAL); // does not allocate data
mxSetDimensions(arr,dims,ndims);          // does not touch data
n=mxGetNumberOfElements(arr);             // only if n isn't known
mxSetPr(arr,mxMalloc(n*sizeof(double))); // NO initialisation
```

mxMalloc is the only memory allocation function that doesn't initialize the allocated data elements.

g) Further Reading

For more detailed information about mexFiles, the user is encouraged to consult to the MATLAB user manual via the following links:

- MEX-files guide: http://www.mathworks.com/support/tech-notes/1600/1605.html
- External Interface/API http://support.rz.rwth-aachen.de/Manuals/MATLAB/techdoc/matlab_external/intro_ma.html
- Jason Laska, Writing C functions in MATLAB. http://cnx.rice.edu/content/m12348/latest/

2.Overview

This section gives a quick overview of AMG's capabilities. It introduces a brief idea, how to use it. This is illustrated with some hypothetical and practical examples.

a) Compilation

Currently, amg has been tested with mingw 6.2 (32 bits). It is expected to work correctly with other Linux(-like) environments. One can construct a platform-dependent by building AMG as follows:

```
$ cd /path/to/amg
$ . amg.init # environments variables for amg
$ make build
```

The executable mexgen.exe is generated within $AMG_HOME/bin/$PLATFORM.

b) Minimal Usage

The general way to call AMG can be done as follows:

```
$./amg  input_file.amg
```

In the input file, it is enough to list a MATLAB prototype function to generate a mexfunction:

```
[@1,@2] = foo($1,$2,$3)
```

where @m refers to the m-th output parameter and $n refers to the n-th input parameter, and both are referred as ***ranked parameters***. The ranked parameters should be listed in the correct order.

A simplified code of the generated mexfunction will be the following foo.mex.c:

```
// header files and macros declaration.
...
void  mexFunction(int nlhs, mxArray *plhs[],
                  int nrhs, const mxArray *prhs[]) {

  // declarations and initialization of objects for the inputs
       MatObj** arg;

       arg = (MatObj **) mxMalloc( sizeof(MatObj *) * nrhs) ;

       for(int i=0;i < nrhs;i++)
             arg[i] = (MatObj *) mxMalloc( sizeof(MatObj) );

  // arg_1*, arg_2* ,arg_3_par r
epresent the input parameters
          for(int i=0;i<nrhs;i++)
              extract_all(prhs[i],arg[i]);

  // setting the pointers of data to arg_1, arg_2 and arg_3
          double* arg_1 = arg[0]->data ;
          double* arg_2 = arg[0]->data ;
          double* arg_3 = arg[0]->data ;

  // initialization of the data field out_1, out_2 of outputs
          double* out_1;
          double* out_2;

  // out_1,out_2 represent the data of the output parameters
          out_1 = create_matrix_fast(&plhs[0],1,1,mxDOUBLE_CLASS,mxREAL);
          out_2 = create_matrix_fast(&plhs[1],1,1,mxDOUBLE_CLASS,mxREAL);

}
```

mexfunction for "foo"

The above code does nothing except restoring all desired fields of an mxArray into special objects of type MatObj, and creating two scalar variables representing the data of the outputs.

- The object “arg[n-1]” is a special data structure, where all information of the n-th input parameter, such as data, dimension, types, .. etc. are stored. This is done by the function “extract_all”, implemented in “mymex.h”.
- This headerfile provides some simple macros where all these fields can be easily accessed. Some of these macros are “nrows”, “ncols” and “nelem”, which refer to the number of rows, the number of columns, and the number of elements respectively. There are also some other macros to indicate the type, the exact dimensions, and others.

- The “out_n” is a void pointer, which points to the first MATLAB matrix entity of the n-th output parameter. These pointers are supposed to store the result of some values in a later phase. In the example above, AMG assumes that the desired outputs are scalars of type double. This is so, because there is no information provided by the user about their dimensions.

c) Calling a C subroutine from Matlab

Clearly, the last example doesn't reflect a real mexfunction where some complex computations are carried out and stored in matrix objects. Suppose now we have a C function of the following prototype:

```
matvex(double[][] A, int m, int n, double[] x, double[] y,double[] z)
```

This routine computes the product of an m x n matrix “A” with a vector “x” of length n, and add the result with another vector “y” of length m x 1, and return the result as an m x 1 vector “z” (similar to the blas routine dgemv). Suppose we want to generate a mexfunction for that routine, in order to be able to call it from MATLAB. AMG can generate a relevant mexfunction using the following input file:

```
@1 = matvec($1,$2,$3) -size(@1) = $3
-kernel {
        matvec($1,nrows($1),ncols($1),$2,$3,@1);
}
```

Note that a reference to the data field of an input or an output parameter is performed with a reference to its rank. This is done by either $n or @m. Note that the output data and information are not stored in any special data structure as the case with the input parameters. At this stage, it is assumed that necessary information of an ouput parameter can be expressed in terms of that of the input parameters. The size of the output parameter is set by the macro “size”. The above macro let the generated mexfunctions create an output parameter that is identical to the third input parameter in size. The generated mexfunction looks like the mexfunction for “foo”, but enhanced with a call, that sets the size of the MATLAB output object corresponding to the first output parameter, and a call to the “mexvec”routine as follows:

```
        out_1 = create_identical_array_size(arg_3,&plhs[0]);
        matvec(data(arg_1),nrows(arg_1),ncols(arg_1),
                data(arg_2),
                data(arg_3),
                out_1);
```

The function “create_identical_array_size” above creates a MATLAB matrix object corresponding to “z”, that is identical to the vector “y” in size. The type of the data remains double by default as long as the user didn't provide any other information about the required output type.

d) Arguments Checking

Well, now the generated mexfunction performs more useful operations than the mexfunction for "foo". So, by passing the right parameters, the desired computation is performed. What if the passed parameters are not correct? For example, if the dimensions of the parameters are not admissible. Unfortunately, there is no guarantee that the function will be executed without any error massages or complain. A worser case can happen if matrices are passed

instead of vectors. In this case, these matrices will be considered as vectors and the mexfunction will return back a wrong answer. The following statements inside the kernel block, before the procedure call, ensure the admissible parameter shapes:

```
if( ( nrows($1) != nrows($3))  ||
   ( ncols($1) != nrows($2)) )
   error("dimensions don't agree");

if( (ncols($2) != 1) || ( ndims($2) > 2) )
   error("second parameter must be a vector");

if( (ncols($3) != 1) || (ndims($3) > 2))
   error("third parameter must be a vector");
```

By adding this code, such errors can be avoided and the user is notified if any thing goes wrong. The routine "error", provided by "mymex.h", is a special routine that performs some kind of clean exception handling mechanism. It can be replaced by the "mexErrMsg" routine. For mexfunctions performing more complex tasks, writing all these exception handling code is a tedious task. The macro "size" can handle such cases. The above input file can be simplified as follows:

```
@1 = matvec($1,$2,$3)
-size(@1) = $1
-size($2) = {ncols($1),1}
-size($3) = {nrows($1),1}
-kernel {
        matvec(data($1),nrows($1),ncols($1),data($2),data($3),@1);
}
```

Setting the size of an input parameter lets AMG generate a code, that checks and ensures that the passed input parameter is of the desired size.

e) Named Parameters

So far, ranked parameters were employed corresponding to the input and output parameters. There is another option which is to use what is called ***named parameters***. With ranked parameter, default macros were generated to refer to the input and output parameters.

An alternative is to employ ***named parameters***. With named parameters, the same names will be used inside the generated code to refer to their corresponding parameters, in every context, wherever this parameter is used. This can enhance readability of the generated code. Therefore, the user is encouraged to choose meaningful names for the parameters. For example, Matrix, Vector, Scalar or a, b, c for scalars; x, y, z for vectors; A, B, C for matrices; etc. Another advantage of using named parameters is that AMG can deduce some information about **identical named parameters**. Identical named parameters informs AMG implicitly, that these parameters should have the same type and shape. In another word, assigning a name to an output parameter that is identical to another input parameter lets AMG create a corresponding MATLAB matrix array mxArray, of which the shape and type are identical to that of the input parameter. This introduces a mechanism of some kinds of exceptions handling.

Applying named parameters instead of ranked parameters, our input file can now be simplified as follows:

```
y = matvec(A,x,y)
-size(x)  = {ncols(A),1}
-size($3) = {nrows($1),1}
-kernel {
        matvec(data(A),nrows($1),ncols(A),data(x),data($3),@1);
}
```

Using the same name "y", as an output parameter and as an input parameter, implies that these parameters

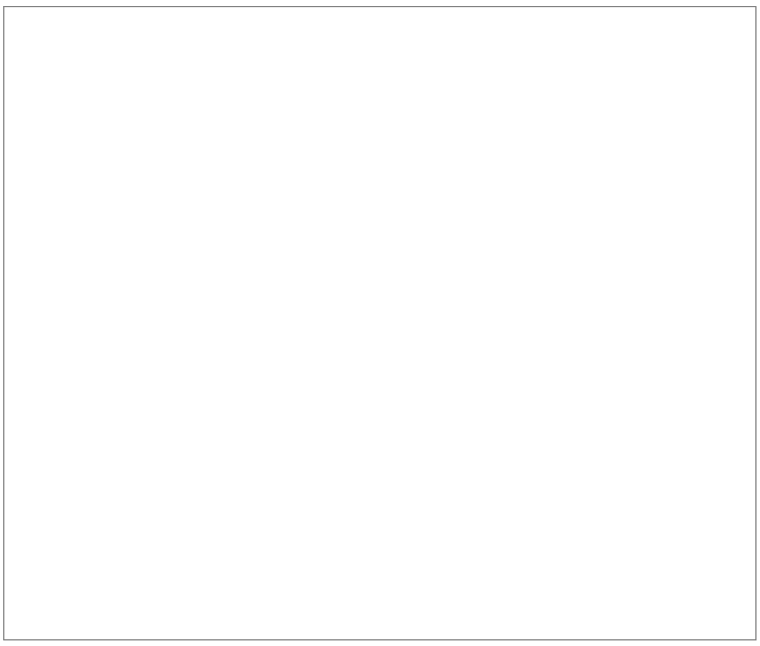
are going to be of identical shape. The generated code doesn't imply by any mean, that the result of the computation is going to be stored in the input parameter corresponding to y. After a call to “matvec”, the third input parameter remains untouched. This is the case unless the user himself manipulate the generated code to store the result of the function in the third input parameter. The names are just dummy variables and can be replaced in reality. By this way, one “size” macro is eliminated in the input file. Note, when named parameters are used, we still can refer to it with its rank. This is useful to avoid confusion that might happen with identical named parameters. Many application can make use of this criteria to generate very useful mexfunctions with minimal macros for setting. The generated mexfunction “matvec.mex.c” is listed in Appendix A.

3. Software Engineering

This section introduces the software process approach of AMG. Going into the details of all the process gives a formal understanding of the tool and its functionalities. The actual design and implementation of AMG adopted the evolutionary development model. This means that an early version of AMG has been initially implemented and it was later refined many times through functionalities modification and enhancement. This section demonstrates the final version of requirement analysis, specification, design and implementation of AMG.

a) Requirement Analysis

The AMG tool generates a C mexfunction, given some minimal information provided by the operator. The operator can be either a normal user or the ADiMAT tool (a tool for automatic differentiation of Matlab programs). The operator needs a mexfunction to make a certain C/FORTRAN routine callable from MATLAB. The operator, ADiMAT or a user, should supply AMG minimal information about the mexfunction to be generated. Once AMG receives the necessary data about the mexfunction required, it generates the corresponding code for it. MATLAB provides all necessary tools to handle the generated mexfunction. These include an interface that is required to handle MATLAB matrix objects and a script to compile and link the generated mexfunction. The final result of this process is an executable, that represents a MATLAB callable C/FORTRAN routine. The following block diagram presents the AMG's environment.



Graph 1: AMG Block Diagram

The required tool generates a C mexfunction, given a MATLAB function prototype and some other optional macros. The general features and the functional requirements that the tool should support are as follows:

1) Given a MATLAB function prototype, an adequate mexfunction should be generated.
2) All basic intrinsic types (double, float, long, integer, short, string, and logical) of actual arguments should be supported
3) For ADiMAT, cell arrays should be also supported. For every active parameter, there is a corresponding cell array for storing its derivatives
4) The ability to specify the physical characteristics (shapes and types) of output parameters, which are going to be created, in terms of the physical characteristics of input parameters
5) The ability to specify the size and type of output parameters, independent from the physical characteristics of output parameters.
6) The ability to include a kernel routine, expressed in terms of the data and characteristics of the given parameters.
7) The ability to specify the basic characteristics of output parameters in terms of the parameters of the given kernel routine.
8) The ability to pass C code to the tool
9) The generated code should be easy to read, understand and edit (Readability).
10) The generated code shouldn't include non-useful extra code like data copying overhead, non-sensible pre-allocation (Performance)

b) Specification Manual

The corresponding specification derived from the functional requirements are as follows:

(1)

A general way to pass a MATLAB function prototype is done according to the following grammar (the bold words represent non-terminals)

output_list = function_name (**input_list**)
Where :
output_list := [**parameter_list**] | **parameter**
input_list := **parameter_list**

parameter_list := **parameter_list , parameter**
parameter := **named_parameter** | **ranked_parameter**
named_parameter := ID
ranked_parameter :=
ranked_output_parameter | **ranked_input_parameter**
ranked_output_parameter := @ RANK
ranked_input_parameter := $ RANK

The above grammar expresses two different ways to pass a function prototype to AMG, either by giving names (named parameters) or assigning ranks to the parameters (ranked parameters). The grammar doesn't prevent a mixture between both ways, and the tool works fine with that too. Identical named parameters imply that these parameters are of identical shape and size. When ranked parameters are used instead, the ranks of the parameters should be passed in the correct order beginning with 1. To make the above grammar as clear as possible, the former restrictions are not shown by the above simple grammar. A ranked input parameter is distinguished with '$', whereas a ranked output parameter is distinguished with '@'.

Example:

```
[a,@2,@3] = foo($1,a,$3,b)
```

(2)

A macro for type can be used to set the type of parameters.

a - Input parameters :

We don't need to specify the type of MATLAB matrix entries, which can be extracted at runtime. However, if the input parameter type is specified by the user, then the generated code checks and ensures that this input parameter is of the desired type.

b- Output parameters :

There are two two ways of setting the type- either implicitly or explicitly.

– Explicit setting:

type (**parameter_list**) := **input_parameter | {explicit_type}**

Where:

parameter_list :=
parameter_list,input_parameter |
parameter_list , output_parameter
input_parameter := **named_parameter | ranked_input_parameter**
output_parameter := **named_parameter | ranked_output_parameter**
ranked_input_parameter := $RANK
ranked_output_parameter := @RANK
explicit_type := float | long | int | short |
logical | char | cell

Example:

```
-type(a,b) = $2
-type(@1,$2) = $1
-type(@1) = int  (not yet implemented)
```

– Implicit setting:

If a named output parameter is identical to a named input parameter, then the MATLAB output matrix object, that will be created, will be of the same type.

– Additional requirements:
- The default type of a non-specified output parameter is double.
- Setting an input parameter type, explicitly or implicitly, generates code that checks and ensures that the specified input parameter is of the desired type.

(3)

Cell arrays can be manipulated and handeld. However, because of their different nature, they are treated differently than special arrays. The conventional macros used with normal objects cann't be used with cell arrays. For cell arrays, there are some other macros for them. These are celldata, cellsize, cellnrows, cellncols, cellclasss and others. To refer to the data field of the first cell of a cell arrays, we use celldata($n,0) and so on.

(4)

There are two macros that can be used, one of them "size" to set the size, and another one "identical" to set the size and the type together: The macro "size" is for setting the shape of a parameter. Setting a parameter with the macro "identical" to another parameter implies that both of them are of the same shape and type.

– Explicit setting:

-size (**parameter_list**) = **input_parameter | array_aggregate**
-identical (**parameter_list**) = **input_parameter**

Where:

input_parameter := named_parameter | ranked_input_parameter
output_parameter := named_parameter | ranked_output_parameter
ranked_input_parameter := $RANK
ranked_output_parameter := @RANK
Array_aggregate := {elements_list}
elements_list := elements_list , element
element := number | dim(input_parameter)[RANK]

&

nrows(input_parameter) := dim(**input_parameter**)[0]
ncols(input_parameter) := dim(**input_parameter**)[1]

-Example:

```
-size(a,b) = c
-size(A) = {nrows($1),ncols($2)} (not yet implemented)
-size(A) = {3,3}                 (not yet implemented)
-identical(A) = B
```

– Implicit setting:
If a named output parameter is identical to a named input parameter, then the MATLAB output matrix object, that will be created, will have the same shape.

– Additional requirement:
Setting an input parameter size, explicitly or implicitly, generates code that checks and ensures that the specified input parameter is of the desired shape.

(5)

The macro "kernel" can be used to inline some code, in which the kernel computational routine is supposed to be called.

```
-kernel {
        C code;
}
```

The code can also contain any other C statements, as well as declarations and assignments.

References to the data and the characteristics of the parameters can be performed, using the following simple macros :

- size: the dimension
- nrows: the number of rows
- ncols: the number of cols
- ndims: the number of dimensions
- numel: the number of elements
- class: the class type of the mxArray

- Example:

```
-kernel {
        int n = 2;
        double* ptr;
        kernel_routine($1,nrows($1),ncols($1),
                        $2,numel($2),n,@1,ptr);
}
```

For a cell array object, the above macros cann't be used. We use the macros listed in 3.

(6)

- A configuration file, under the directory "config", can also be used to assist formatting the generated code, using the GNU indent tool.
- To enhance readability, if a named parameter is used, a corresponding macro is also generated within the output code to refer to it, in every context where this parameter is used.
- Input files can be commented.
  Example:
  %* here documentation *%

(7)

Data copying overhead should be avoided, unless it is desired. The macro "abbreviated" enforces the tool not to produce such code. So, a basic physical characteristic is extracted just in the context, where it is desired. The default way of creating MATLAB matrix objects is to create them without pre-allocation. However, if pre-allocation with 0 is desired, then the macro "--nofast" can be used.

c) Design and Implementation

The AMG tool consists mainly of three components: lexical analyzer, parser and generator.

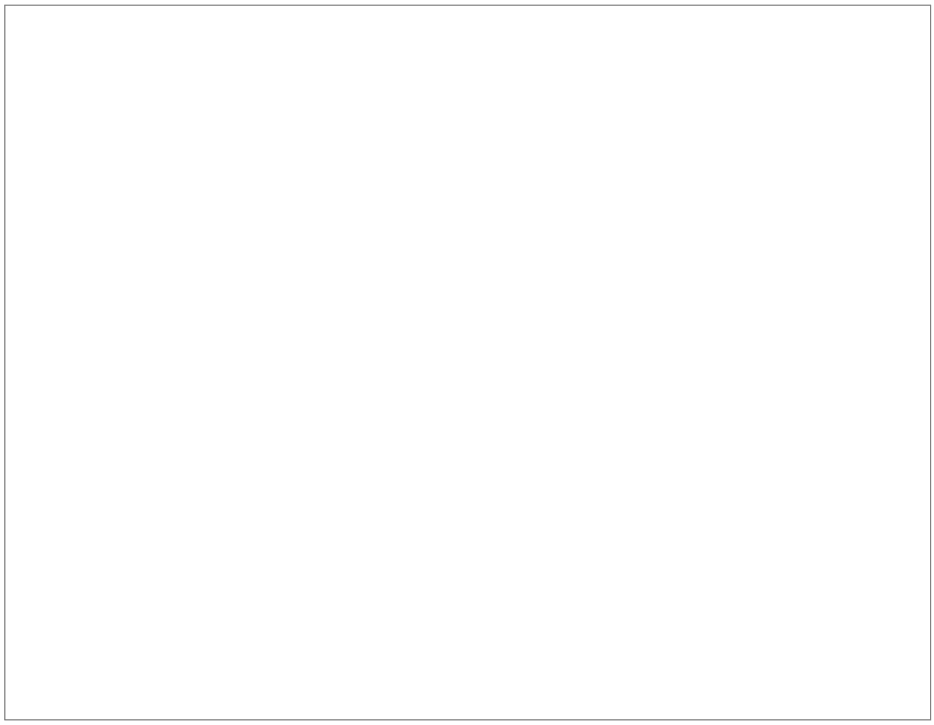
- The lexical analyzer receives the user directives, described in the specification manual, as inputs, and tokenizes them

- The produced tokens are then forwarded to the parser, which checks their syntax. The parser tries to classify these tokens into relevant rules. These rules express necessary data and information about the desired mexfunction and its parameters, that should be provided to the generator unit. These data and information are then

stored into relevant data structure objects

- The generator unit manipulates these objects and generates the desired code accordingly. A seemingly static component, "mymex.h", provides an interface to the generated mexfunction. This interface eases the code generation phase by summarizing related groups of mxArray operations into smaller functional units. By this technique, only calls to these functions are generated, instead of their underlying implementation.

Graph 2 describes AMG tool.



Graph 2: AMG Block Diagram

There are mainly two consistent implementation approaches for mexfunctions:

The first one, the data and the information of the input parameters are extracted into relevant data structure objects, provided by "mymex.h". These objects contain all necessary data and information about an input parameter accessible to the user inside the “kernel” block. Extracting all fields of a mxArray is not really necessary, however, there is no penalty on performance because there is no data copying overhead. Only references to the desired data are extracted. Using the macro “abbreviated” disables this feature and desired reference of data and fields of an mxArray are extracted only in the context where they are mentioned. The interface "mymex.h" provides several macros to access the data and the basic physical characteristic. These are, as described in the specification manual section, “data”, “dims”, “ndims”, “nrows”, “ncols”, “nelem” and “class”.

The difference between both approaches is distinguished by the way the output objects are created. The first approach is to create unpopulated MATLAB matrix objects to the desired shape and type. The desired entities of a MATLAB matrix objects are then computed in the “kernel” block, using a constant reference to the data field. Conversely, in the second approach, all necessary information and data about the output parameters are computed first. Then, together with the information provided by the user, the corresponding MATLAB matrix objects are created and set at the last phase.

Possible future enhancement: The “inline” macro

There is a big difference between the macro “kernel” and the macro “inline. Inside the “kernel” block, setting the data or any other physical characteristic is not possible. The physical characteristics of an output parameter are not stored in any special data structure like those of the input parameters. Using the macro “kernel” lets AMG generate code that creates mxArrays that correspond to the output parameters. On the other side, with the macro “inline”, all known physical characteristics provided by the user are stored in special data structures. Creation of mxArrays corresponding to the output parameters is shifted to last phase. So, if the macro “inline” is used, all fields of the MATLAB matrix arrays can be set at this stage. This is because the output parameters have not been yet created as MATLAB objects. In the last phase, the MATLAB matrix objects corresponding to the output parameters are created.

As a summary, the macros “inline” and “kernel” show two different styles of mexfunction implementation. The simpler style is to extract the information needed from the mxArrays representing the input parameters, create the output parameters in terms of the available information and finally perform the desired computation. The other style is to shift the output creation to the last phase and gather necessary information about the output before that. So, “inline” is useful when:
1- Reference to some fields of some output parameters is desired and it is not possible to express them in terms of the fields of some input parameters.
2- Special setting of some fields of some output parameters is desired, and the available macros don't support them.