

Walkthrough for Creating a MATLAB DLL That Can Be Called in LabVIEW

Let me start by saying that I am neither a MATLAB nor a C developer, but I do have decent LabVIEW experience. I was recently tasked with compiling some MATLAB code into a DLL that was callable in LabVIEW, and I found the process difficult primarily because documentation was variously nonexistent, incomplete, in far-flung places, or written with the assumption that the coder had MATLAB and C experience and could fill in many blanks in the process. This is not an unrealistic expectation, but the lack of thorough documentation was still frustrating. I did finally find an [example](#) that was the keystone to enabling me to correctly compile my MATLAB function into DLL that could be used in a C environment. This walkthrough presents a trivial, but complete example of how to create such a DLL from a MATLAB script *and* how to call it in LabVIEW. I built the walkthrough based on the above example with some notable changes:

- Much more verbose documentation to help an inexperienced developer
- Clearer variable and file names
- Explanation of concepts and roadblocks I found nonintuitive
- Links to relevant help/forums/other resources
- Explicit LabVIEW integration instructions
- A working solution that interfaces with LabVIEW

There is a complete solution presented along with these instructions, and I would encourage you to follow each step presented here to recreate that solution yourself. The entire process, assuming no hitches, should take around half an hour. Still, you should probably set aside a few hours if you're new to building DLLs. You will likely run into compiler issues, missing installations, or rabbit holes you'll want to explore as you go. You will need MATLAB (I used R2016b), the MATLAB Compiler SDK (which enables building the DLL), a MATLAB-compatible C compiler (I used the Windows SDK 7.1), and a version of LabVIEW (I used 2018 64-bit). From all the various documentation and discussions I've found online, the DLL you create will ONLY be callable in a version of LabVIEW whose bitness is the same as your version of MATLAB. FYI, R2015b was the [last MATLAB version with a 32-bit build](#). If you're developing in MATLAB R2016a or later, you will only be able to call your DLL in a 64-bit version of LabVIEW.

Here we go.

1. Create a function within a script in MATLAB.

- In the trivial exampleScript.m case below, the function exampleScript takes a numeric array as an input, increments all elements by 1, and outputs the incremented array. Write this function and save the M script. The "m" prefix in the parameter names will help distinguish these parameters from similarly named parameters in the wrapper C functions we will build later.

```
1  function [mOut2dArray] = exampleScript(mIn2dArray)
2
3      %Add 1 to all array elements
4  -   mOut2dArray = mIn2dArray + 1;
5
6  -   end
```

- b. Test the function to make sure it returns the expected output.

```
Command Window
>> [mOut2dArray] = exampleScript([0,1;2,3])

mOut2dArray =












     1     2
     3     4
```

2. Compile the script into a DLL.

- a. Make sure that your MATLAB working directory is the same that contains your .m file. Enter the command below in the MATLAB command line and press "Enter".¹ See [here](#) for documentation on the command line option flags if you're curious.

```
>> mcc -v -B csharedlib:exampleScript exampleScript.m
```

- b. Once the DLL has compiled, you should see something like the following in your working directory.

Name	Date modified	Type	Size
 exampleScript.c	1/16/2019 11:26 AM	C File	4 KB
 exampleScript.def	1/16/2019 11:26 AM	DEF File	1 KB
 exampleScript.dll	1/16/2019 11:26 AM	Application extens...	47 KB
 exampleScript.exp	1/16/2019 11:26 AM	EXP File	2 KB
 exampleScript.exports	1/16/2019 11:26 AM	EXPORTS File	1 KB
 exampleScript.h	1/16/2019 11:26 AM	H File	3 KB
 exampleScript.lib	1/16/2019 11:26 AM	LIB File	4 KB
 exampleScript.m	1/14/2019 5:09 PM	MATLAB Code	1 KB
 mccExcludedFiles.log	1/16/2019 11:26 AM	Text Document	2 KB
 readme.txt	1/16/2019 11:26 AM	TXT File	2 KB
 requiredMCRProducts.txt	1/16/2019 11:26 AM	TXT File	1 KB

¹ I used the Windows SDK 7.1 to supply my C Compiler, though other MATLAB-supported C compilers should also work. Note that if you have not used the Windows SDK 7.1 before, you will likely get an error like the following when you compile your DLL. "Warning: Windows SDK 7.1 appears to be installed, but its compiler was not found. Install .NET Framework 4.0, and then rerun the SDK installation to add the "Visual C++ Compilers" component. See <http://www.mathworks.com/support/solutions/en/data/1-IB1G3Q/> for more information. I installed [this](#) patch, which resolved the error.

- c. Open the exampleScript header file and note that there are several “general use” library functions, which enable the loading and unloading of the script from memory and printing the stack trace for debugging.² We will use some of these later.

```
55 #ifndef LIB_exampleScript_C_API
56 #define LIB_exampleScript_C_API /* No special import/export declaration */
57 #endif
58
59 extern LIB_exampleScript_C_API
60 bool MW_CALL_CONV exampleScriptInitializeWithHandlers(
61     mclOutputHandlerFcn error_handler,
62     mclOutputHandlerFcn print_handler);
63
64 extern LIB_exampleScript_C_API
65 bool MW_CALL_CONV exampleScriptInitialize(void);
66
67 extern LIB_exampleScript_C_API
68 void MW_CALL_CONV exampleScriptTerminate(void);
69
70
71
72 extern LIB_exampleScript_C_API
73 void MW_CALL_CONV exampleScriptPrintStackTrace(void);
```

- d. Note that there are also two implementations of the “user-defined” exampleScript function: mlxExampleScript() and mlfExampleScript().

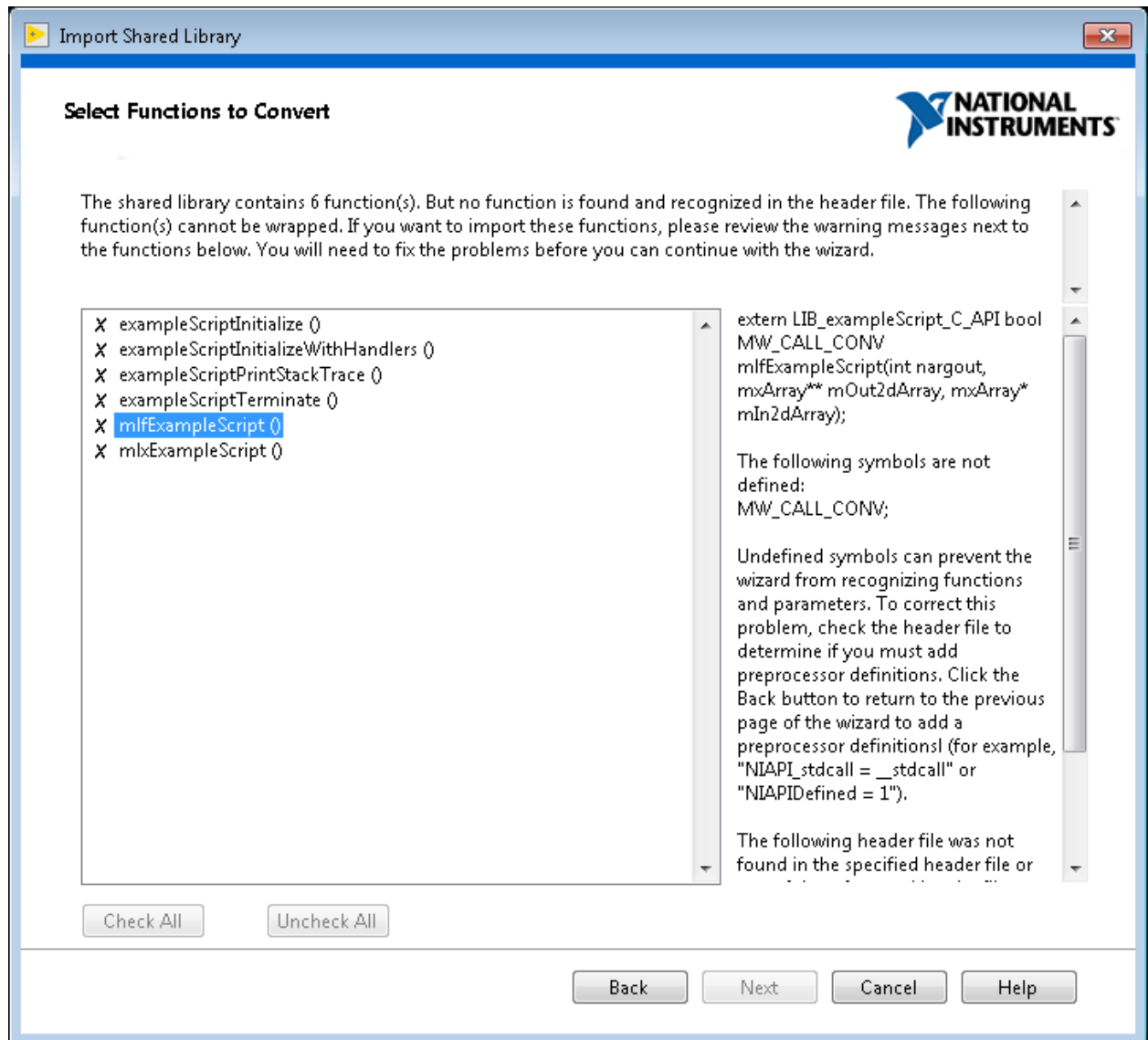
```
76 bool MW_CALL_CONV mlxExampleScript(int nlhs, mxArray *plhs[], int nrhs, mxArray *prhs[]);
77
78
79
80 extern LIB_exampleScript_C_API bool MW_CALL_CONV mlfExampleScript(int nargout, mxArray** mOut2dArray, mxArray* mIn2dArray);
```

- e. See [here](#) for a discussion on the difference between the mlx and mlf functions. The short of it is that both functions will execute the same exampleScript function defined in exampleScript.m, but the mlf function presents a simpler interface since the inputs and outputs are not wrapped in extra arrays, plhs[] and prhs[], that would require more parsing. Thus, we’re going to use the mlfExampleScript() function.

3. Celebrate prematurely until you realize that something is not quite right.

- a. If you now use the Import Shared Library Wizard in LabVIEW, you will see the following function prototypes. (You don’t have to actually do this step unless you really want to. This step is just here to illustrate a point. A complete set of instructions on how to use the wizard is presented later.)

² The exact functions and/or formatting may be different if you’re using a different C compiler.



- b. MATLAB natively exposes all of its function parameters (including scalars, strings, etc.) with the [mxArray type](#), as seen in the mlf function prototype screenshot from before. This is not a native C type, which is what LabVIEW knows how to interface with. LabVIEW does not know what to do with these types, which will pose problems for us later.
 - c. To expose the function parameters as native C types instead of mxArray types, there must be an interface layer that translates between the C types and the mxArray types. A wrapper DLL is a good choice for this interface, as it can abstract away this conversion into a compiled binary that exposes a simple-to-use interface to LabVIEW. The DLL can also wrap and expose functions that load/unload the exampleScript function and its dependencies. (Recall the “general use” functions from exampleScript.c, above.)
4. Create a wrapper DLL.

- a. Write an “exampleWrapper.c” file that exposes the mlfExampleScript() function with native C types.³ This is easier shown than explained, so see exampleScript.c that ships with the full solution to this walkthrough and use it if you’re new to C and don’t know how you’d write this yourself. It is beyond the scope of this document to explain every intricacy of this file. (You wouldn’t want me to anyway. I am NOT a C developer and, as mentioned before, learned how to use and modify the code from an example found [here](#).)
- b. Note that the C file defines functions to load/unload the exampleScript function and its dependencies.

```
6      /*Function to initialize the called DLL*/
7      void loadExampleScript(void) {
8          exampleScriptInitialize();
9      }
10
11     /*Function to terminate the called DLL*/
12     void unloadExampleScript(void) {
13         exampleScriptTerminate();
14     }
```

- c. Note that the wmlfExampleScript() function is the wrapper for our mlfExampleScript() function that is a member of the exampleScript.dll binary. The wmlfExampleScript() function needs to allocate pointers to MATLAB mxArray's, clean up these variables, convert between C and MATLAB data types, and call the M function in exampleScript.dll.

³ This is the crux of this DLL creation process. It is objectively not that difficult, but it does require knowledge of C syntax and how to read/interpret C documentation. (Read: I did not have much of this knowledge, and it was hard for me to initially grapple with and modify despite the simple-looking code.) As you will see, the example presented in this document changes the values of an array in-place and passes a pointer to that same array back to the caller. This is a pretty simple operation. To accept/return any other data types (scalars, strings, or arrays of other types), you will need to use various other functions in the [C Matrix API](#) to translate between C and MATLAB mx types. You will save yourself a lot of time and headache if you consult with a C developer for this process to debug syntax issues and to figure out which functions you’ll need.

```

16  /*Function that wraps the top-level m function and converts C types to/from the MATLAB mxArray type*/
17  int wmlfExampleScript(double* c2dArray, int numRows, int numColumns){
18
19      /*Define variables*/
20      int nargout=1;
21      int result=1;
22
23      /*Define pointers to MATLAB mxArray data*/
24      mxArray *mIn2dArray;
25      mxArray *mOut2Array=NULL;
26
27      /*Convert C data to MATLAB data*/
28      mIn2dArray=mxCreateDoubleMatrix(numRows, numColumns, mxREAL);
29      memcpy(mxGetPr(mIn2dArray), c2dArray, numRows*numColumns*sizeof(double));
30
31      /*Call the M function*/
32      result=mfExampleScript(nargout, &mOut2Array, mIn2dArray);
33
34      /*Convert returned MATLAB data to C data*/
35      memcpy(c2dArray, mxGetPr(mOut2Array), numRows*numColumns*sizeof(double));
36
37      /*Clean up MATLAB variables*/
38      mxDestroyArray(mIn2dArray);
39      mxDestroyArray(mOut2Array);
40
41      return result;
42  }

```

- d. Note that the “m” and “c” prefixes to parameter names are used to designate which variables are MATLAB mxArray types and which are C types. This is the convention I chose for this example and is by no means required. Note also that while the MATLAB array variable names are “mIn2dArray” and “mOut2Array”, the c2dArray has no “In” or “Out” designation. This is because c2dArray is a pointer to the array, and we are going to modify the array’s elements in-place. In other words, LabVIEW will pass the exampleWrapper DLL a pointer to the array, the DLL will call the underlying code to modify the array elements, and LabVIEW will then look at the array at that same pointer location to observe the modified array.
- e. Create an “exampleWrapper.def” file and an “exampleWrapper.h” file in the same working directory. You will need the DEF file to compile your DLL. It tells the compiler the name of the target library and which DLL functions to expose. We’ll expose all three functions we wrote.

```

exampleWrapper.def
1  LIBRARY exampleWrapper
2  EXPORTS
3  loadExampleScript
4  unloadExampleScript
5  wmlfExampleScript

```















You will need the H file to import your DLL into LabVIEW using the Import Wizard. The H file defines the function prototypes that LabVIEW will use to interface with the DLL.

```

exampleWrapper.h
1  void loadExampleScript(void);
2  void unloadExampleScript(void);
3  int wmlfExampleScript(double* c2dArray, int numRows, int numColumns);

```


















Your working directory should now look like this:

Name	Date modified	Type	Size
 exampleScript.c	1/16/2019 11:26 AM	C File	4 KB
 exampleScript.def	1/16/2019 11:26 AM	DEF File	1 KB
 exampleScript.dll	1/16/2019 11:26 AM	Application extens...	47 KB
 exampleScript.exp	1/16/2019 11:26 AM	EXP File	2 KB
 exampleScript.exports	1/16/2019 11:26 AM	EXPORTS File	1 KB
 exampleScript.h	1/16/2019 11:26 AM	H File	3 KB
 exampleScript.lib	1/16/2019 11:26 AM	LIB File	4 KB
 exampleScript.m	1/14/2019 5:09 PM	MATLAB Code	1 KB
 exampleWrapper.c	1/15/2019 2:54 PM	C File	2 KB
 exampleWrapper.def	1/15/2019 1:51 PM	DEF File	1 KB
 exampleWrapper.h	1/15/2019 2:54 PM	H File	1 KB
 mccExcludedFiles.log	1/16/2019 11:26 AM	Text Document	2 KB
 readme.txt	1/16/2019 11:26 AM	TXT File	2 KB
 requiredMCRProducts.txt	1/16/2019 11:26 AM	TXT File	1 KB

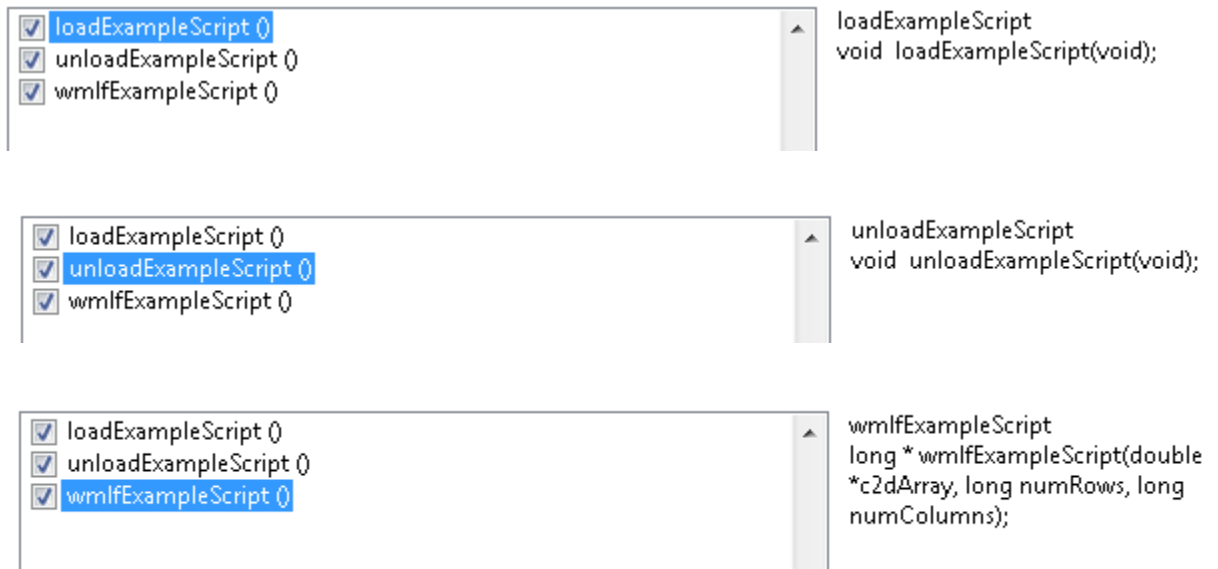
- f. Make sure that your MATLAB working directory is set correctly. Enter the command below in the MATLAB command line and press “Enter”.

```
>> mbuild -v exampleWrapper.c exampleScript.lib LINKFLAGS="$LINKFLAGS /DLL  
/DEF:exampleWrapper.def" LDEXT=".dll" CMDLINE250="mt -outputresource:$EXE';2 -  
manifest $MANIFEST"
```

- g. Your working directory should look like this:

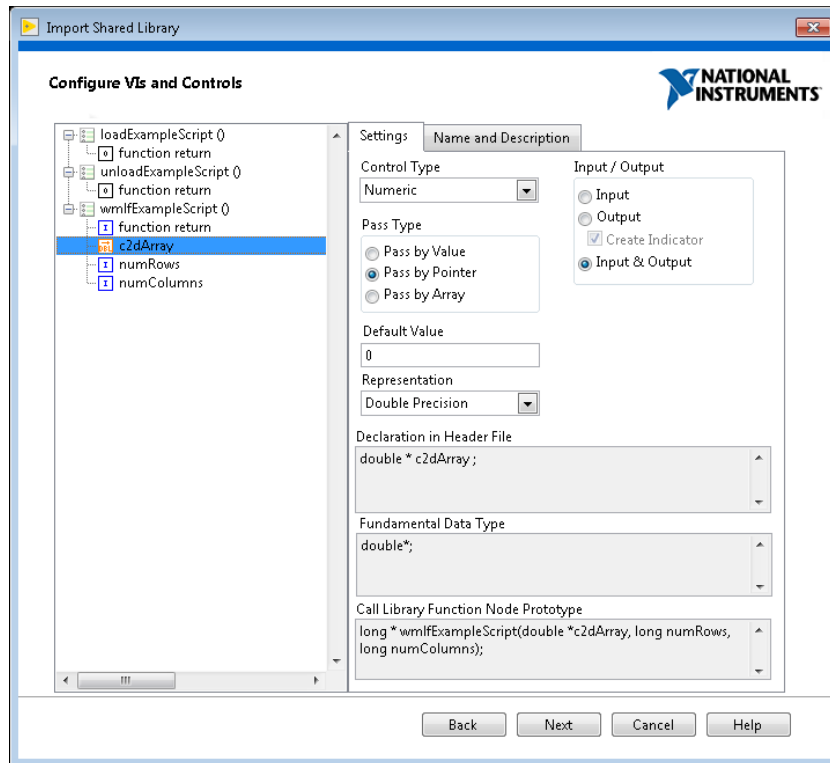
Name	Date modified	Type	Size
 exampleScript.c	1/16/2019 11:26 AM	C File	4 KB
 exampleScript.def	1/16/2019 11:26 AM	DEF File	1 KB
 exampleScript.dll	1/16/2019 11:26 AM	Application extens...	47 KB
 exampleScript.exp	1/16/2019 11:26 AM	EXP File	2 KB
 exampleScript.exports	1/16/2019 11:26 AM	EXPORTS File	1 KB
 exampleScript.h	1/16/2019 11:26 AM	H File	3 KB
 exampleScript.lib	1/16/2019 11:26 AM	LIB File	4 KB
 exampleScript.m	1/14/2019 5:09 PM	MATLAB Code	1 KB
 exampleWrapper.c	1/16/2019 11:46 AM	C File	2 KB
 exampleWrapper.def	1/15/2019 1:51 PM	DEF File	1 KB
 exampleWrapper.dll	1/16/2019 11:56 AM	Application extens...	8 KB
 exampleWrapper.exp	1/16/2019 11:56 AM	EXP File	2 KB
 exampleWrapper.h	1/15/2019 2:54 PM	H File	1 KB
 exampleWrapper.lib	1/16/2019 11:56 AM	LIB File	3 KB
 mccExcludedFiles.log	1/16/2019 11:26 AM	Text Document	2 KB
 readme.txt	1/16/2019 11:26 AM	TXT File	2 KB
 requiredMCRProducts.txt	1/16/2019 11:26 AM	TXT File	1 KB

5. Import the shared library into LabVIEW. You can manually configure Call Library Function Nodes to call the functions or use the Import Wizard to build a basic API. For simplicity, these instructions use the latter method.
 - a. Open LabVIEW (of the same bitness as the version of MATLAB). At the splash screen, select Tools>>Import>>Shared Library.
 - b. Make sure "Create VIs for a shared library" is selected. Click "Next".
 - c. Click the browse button for the "Shared Library (.dll) File" field, navigate to your working directory, and select "exampleWrapper.dll". The "exampleWrapper.h" header file should auto-fill in the lower field since it is in the same directory as the DLL. Click "Next".
 - d. Click "Next" again since we don't need to add any "Include Paths".
 - e. Ensure that all three function prototypes are correct. If they are not, you likely have a mistake in your exampleWrapper.h file. Fix those mistakes, save the .h file, and restart the import process. There's no need to recompile the exampleWrapper DLL.



Note that `wmIfExampleScript()`'s data types are now our desired C types, and not the `mxArray` types from before when we tried to import the `exampleScript` DLL's functions. Our wrapper has served its purpose! Click "Next".

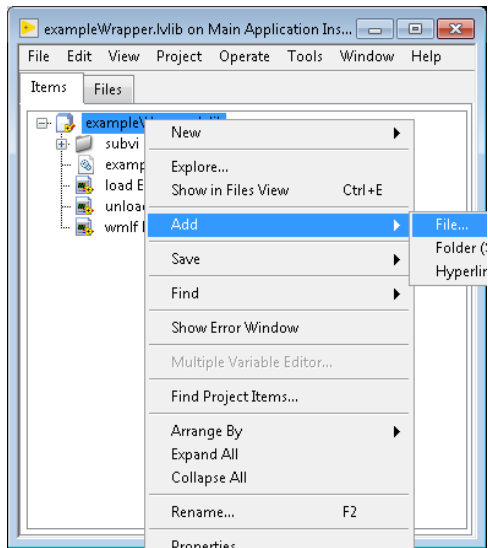
- f. The next screen will allow you to change your project library name and path. By default, the name will be the same as the name of the DLL, and the path will be in `user.lib`. Leave these as default unless you have good reason to change them and click "Next".
- g. Select your error handling strategy and click "Next". (To follow the example instructions exactly, select "Function Returns Error Code/Status".)
- h. Review how all parameters will cast to VI controls and indicators. The `exampleWrapper`'s `c2dArray` variable will be passed as a pointer by default since this is the C data type.



We will want the LabVIEW user to actually pass in/out an array of doubles instead of a pointer to that array because this is how a LabVIEW user expects to pass data to VIs. That said, this Import Wizard does not give you the option to change the dimensionality of the array even if you did change “Pass Type” to “Pass by Array”, so we will just do both steps manually in LabVIEW once the library is built. Leave the defaults and click “Next”.

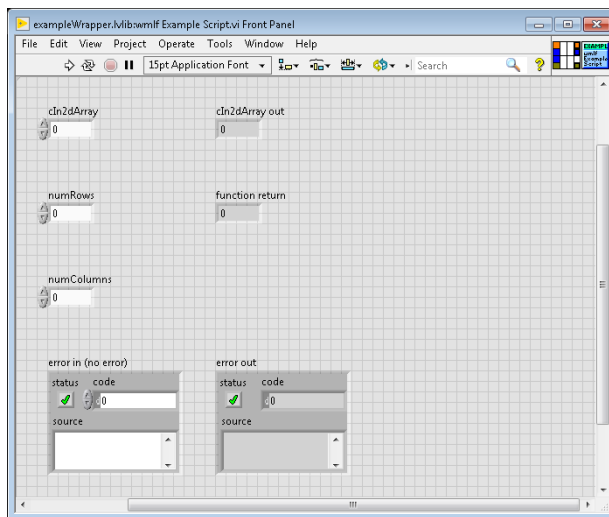
- i. Click “Next” on the “Generation Summary” page to build the library, and then click “Finish”.
6. Examine and finalize the API. The LVLIB containing your wrapped functions should open automatically. If it doesn’t, open it from <LabVIEW 20xx>/user.lib/exampleWrapper.
 - a. Open one of your functions, and you’ll find that there’s a broken run arrow because exampleScript.dll cannot be found, which makes sense since the Import Wizard only knew about exampleWrapper.dll and didn’t know that it depended on another DLL. To fix this, copy exampleScript.dll from your MATLAB working directory into the <LabVIEW 20xx>/user.lib/exampleWrapper directory. Then right click on exampleWrapper.lvlib in the LabVIEW window, select Add>>File, and browse to exampleScript.dll in <LabVIEW 20xx>/user.lib/exampleWrapper.⁴

⁴ Do NOT select the exampleScript.dll in your original working directory. This is a bad practice because you will easily run into dependency issues in the future if you ever change the working directory’s location or contents.

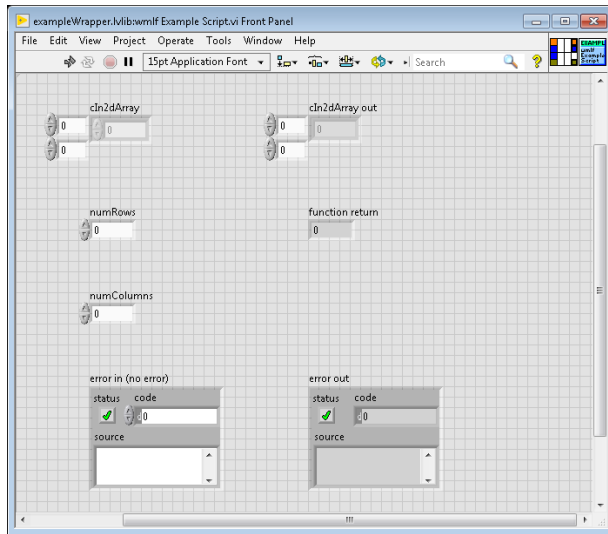


Once the exampleScript DLL is added to the library, save all (Ctrl+Shit+S).

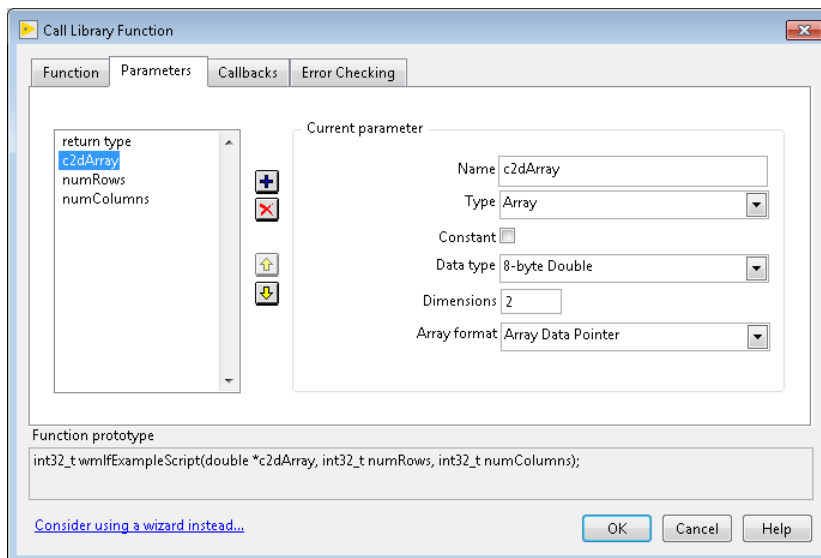
- b. Close any open VIs, and then reopen one of them. LabVIEW should now find exampleScript.dll, and the VI should have an unbroken run arrow.
- c. Open “wmIf Example Script.vi” and note that “cIn2dArray” and “cIn2dArray out” are scalars. This is because the import wizard assumed that we wanted to pass in/out pointers to the array and not the array value.



- d. Replace the scalars with 2d arrays of doubles. This will break the run arrow again since LabVIEW still thinks we should be passing a pointer, not the array itself, to the DLL.

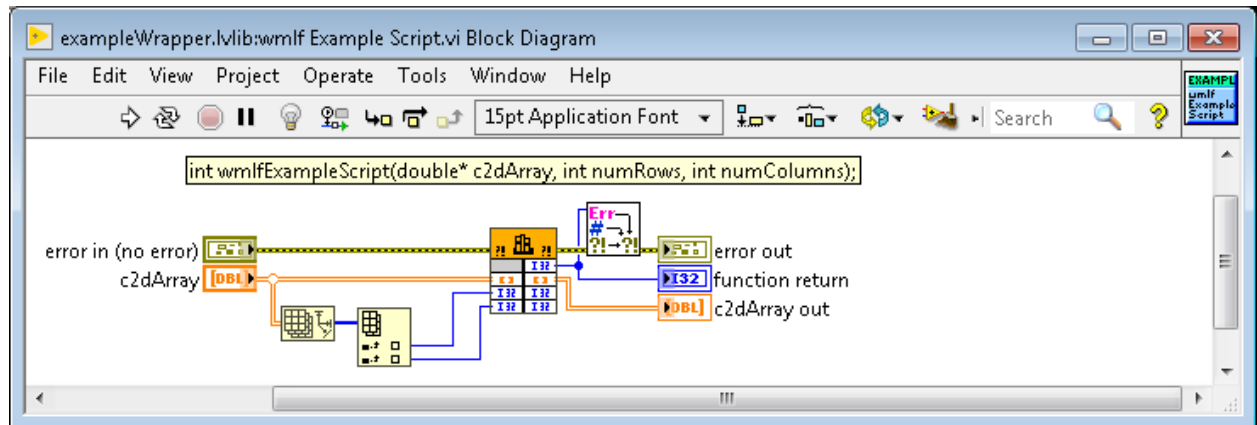


- e. On the block diagram, double-click the Call Library Function Node, and select the “Parameters” tab. Click on c2dArray, change the “Type” to “Array” and the “Dimensions” to “2”.



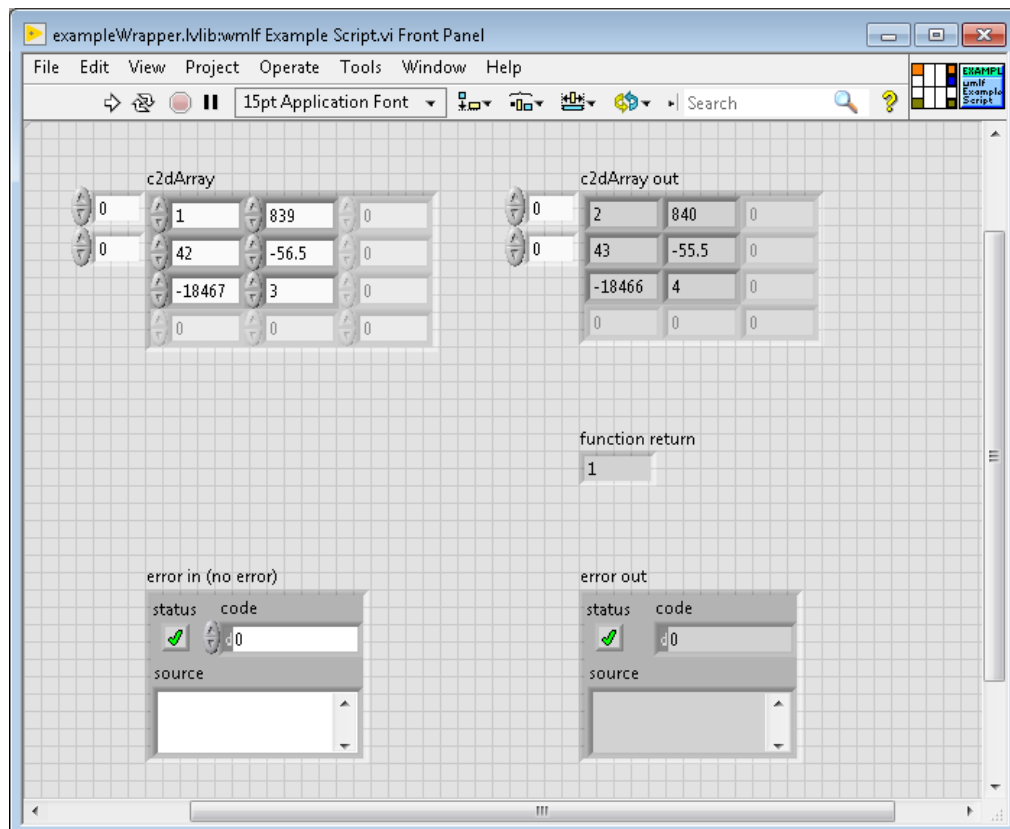
The Call Library Function node will now accept and return 2D arrays of doubles, and internally it will handle passing a pointer to the DLL so the user doesn’t have to worry about this conversion.

- f. Finally, use the “Array Size” and “Index Array” LabVIEW primitives to get the dimensions of c2dArray, and pass those dimensions into the function so the user doesn’t have to enter them manually.



7. Test the library's functionality.

- Open "load Example Script.vi" and run it. This loads the exampleScript function and its dependencies into memory. You will always have to run this VI when you start your program to use the main exampleScript function.
- Open "wmlf Example Script.vi", select some test values for c2dArray, and run the function. If the function returns 1, that means it executed successfully. If it returned 0, that means that it was not successful. If you get 0, make sure that you have run "load Example Script.vi" first.



- c. Note that all input array elements were incremented by 1 as expected, and that the function returned 1 (success). You will probably want to use the Error Converter subVI on the block diagram to convert any return codes you add to your functions to LabVIEW errors to make the API more user-friendly. For example, if the function returns 0, you may want to alert the user that they have likely not yet loaded the library and should make sure to run “load Example Script.vi”.
 - d. Run “unload Example Script.vi”, and note that “wmlf Example Script.vi now returns 0. Again, this is expected because the underlying function has not been loaded.
8. Celebrate and/or despair.
- a. If this worked for you, I hope you found this document useful. Best of luck on building your own MATLAB function into a DLL!
 - b. If this didn’t work for you, please post your full code solution, your LV/MATLAB versions, and the problems you’re facing to get help.