

Using Matlab to Generate C & Speeding up Matlab Using MEX's

Since R2011a, Matlab has been capable of generating C code.

- This code can be used as a stand-alone executables, or
- This code can be compiled as subroutines for use within Matlab called 'Matlab EXecutables' (.MEX files).

Why C?

- You may need C for a given application.
- Matlab is painfully slow. MEX files can substantially increase computational speed.

Getting Started

You must set up Coder to work with your choice of compiler.
Enter

```
mex -setup
```

At the command line.

Enter y to see the list of installed compilers.

You will be presented with a choice of compilers, depending on what is installed on your machine.

Select a supported compiler.

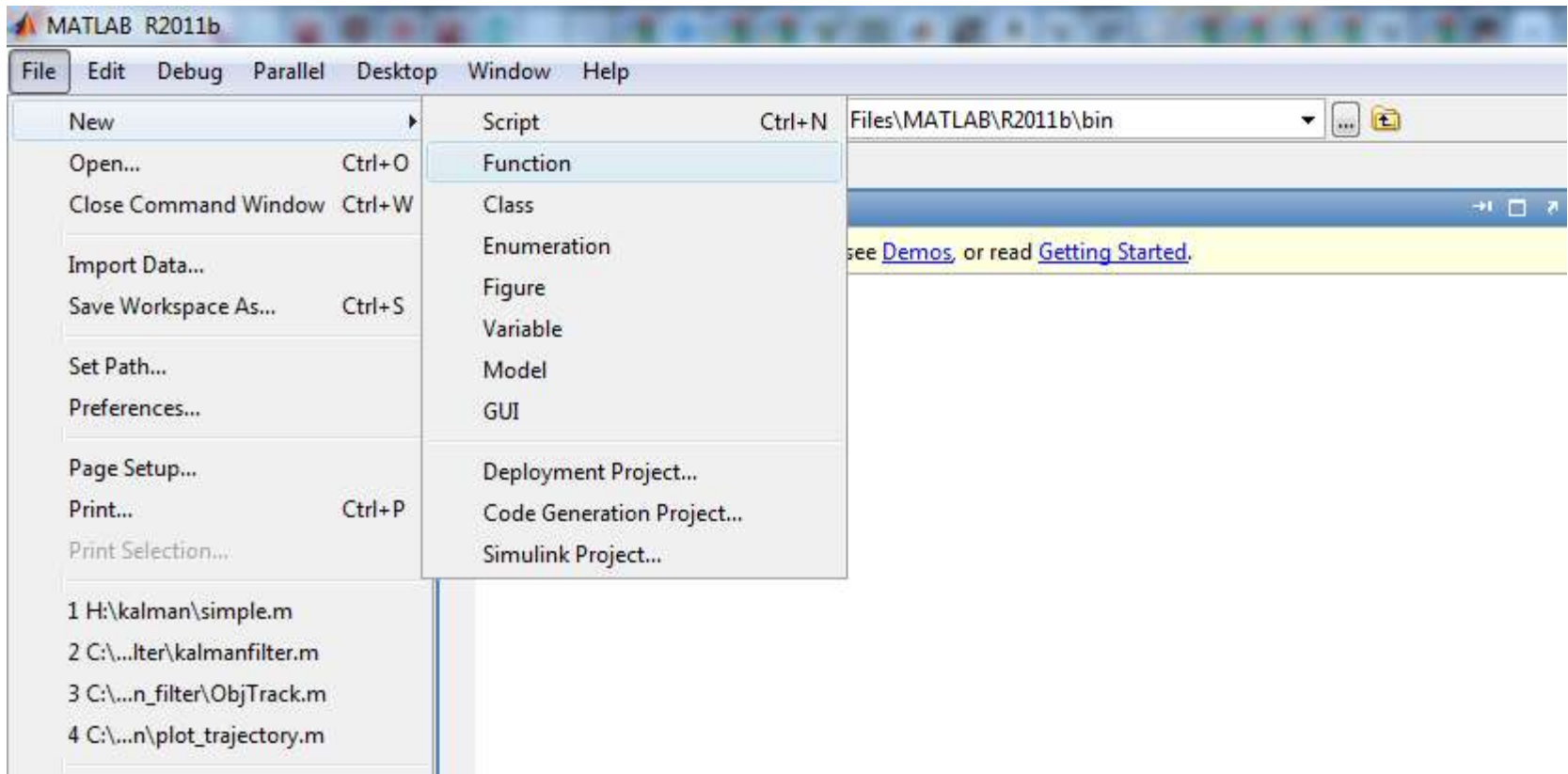
Enter y to verify your choice.

Note: “The default compiler that MathWorks supplies with MATLAB for Windows 32-bit platforms is not a good compiler for performance.” From MathWorks.

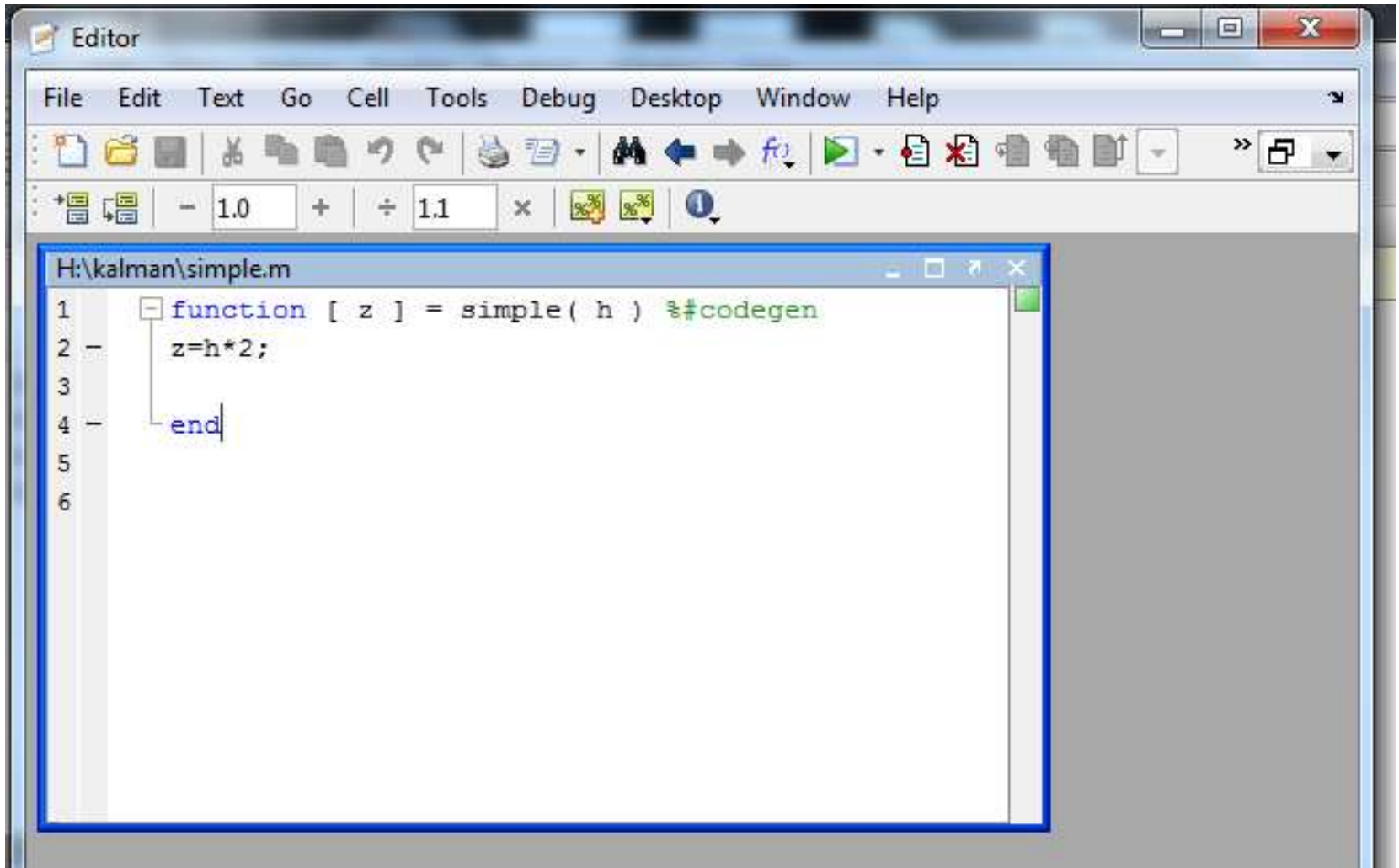
Stand-alone C

- Create a function m-file.
- Create the C source code using Matlab Coder.
- Create a main function to implement your code.
- Compile with your choice of compiler.

Create a function m-file.



This simple m-file multiplies its argument (h) by 2:

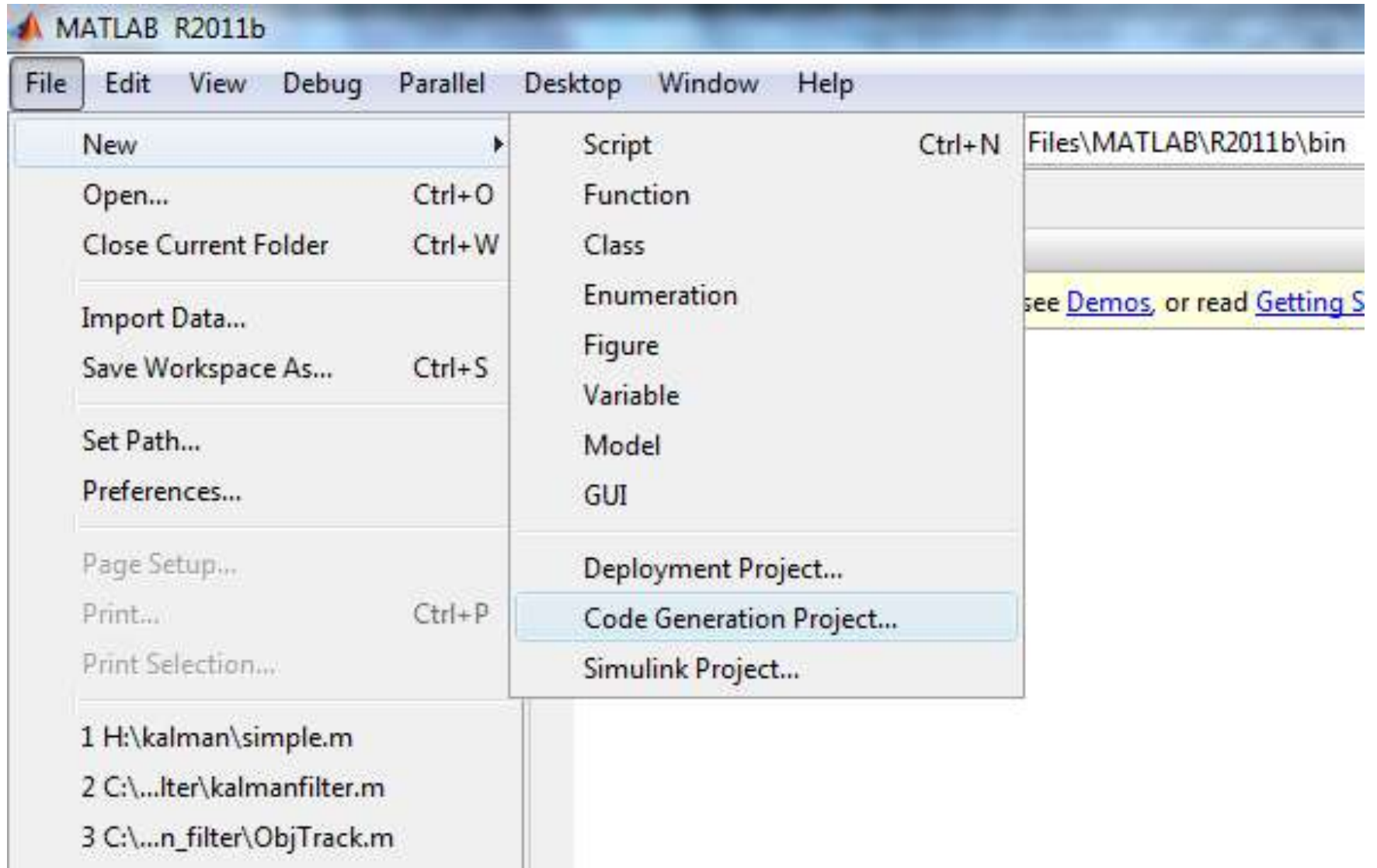


```
H:\kalman\simple.m
1 function [ z ] = simple( h ) %#codegen
2     z=h*2;
3
4 end
5
6
```

Notice the `%#codegen` pragma

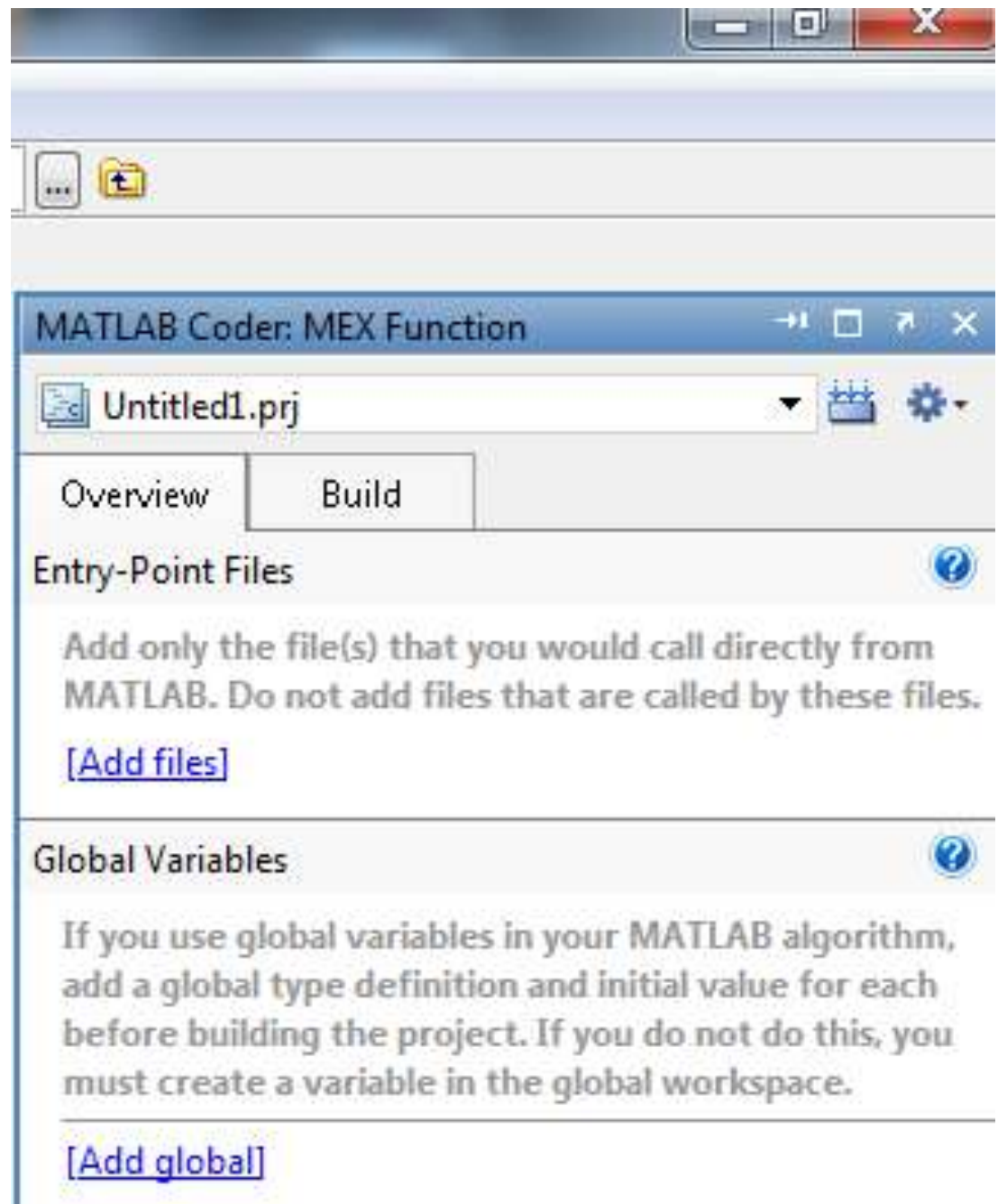
- This notifies the Matlab code analyzer (what normally notifies you of missing semicolons etc. [a.k.a. the bar to the right of the edit window]) that you will be attempting to make C code.
- The analyzer will now notify you of any conflicts that may arise when trying to translate your code to C.

Use Coder to write the C source code

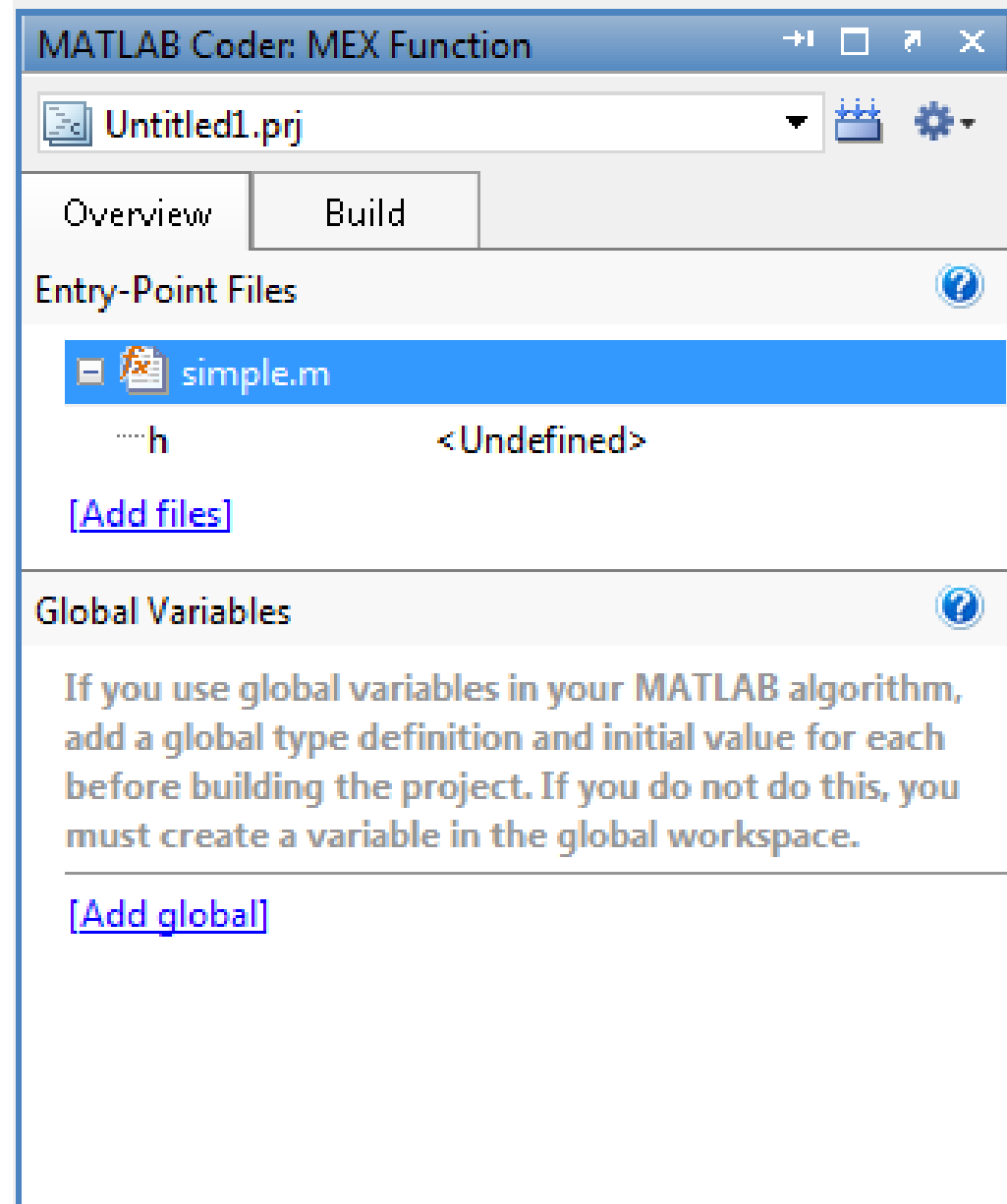


The window shown will open to the right of your screen.

The “Entry-Point” is the function we want to execute, namely, the m-file, simple.m.



Add entry point file



Notice that our input to simple is undefined.

- Because C requires that all variable types be explicitly declared, this is something we need to remedy.
- Click on the gear-like icon to define h within Matlab.
- Matlab will automatically determine how to handle h as well as generate error controls in the resulting C code.

Suitably define all variables

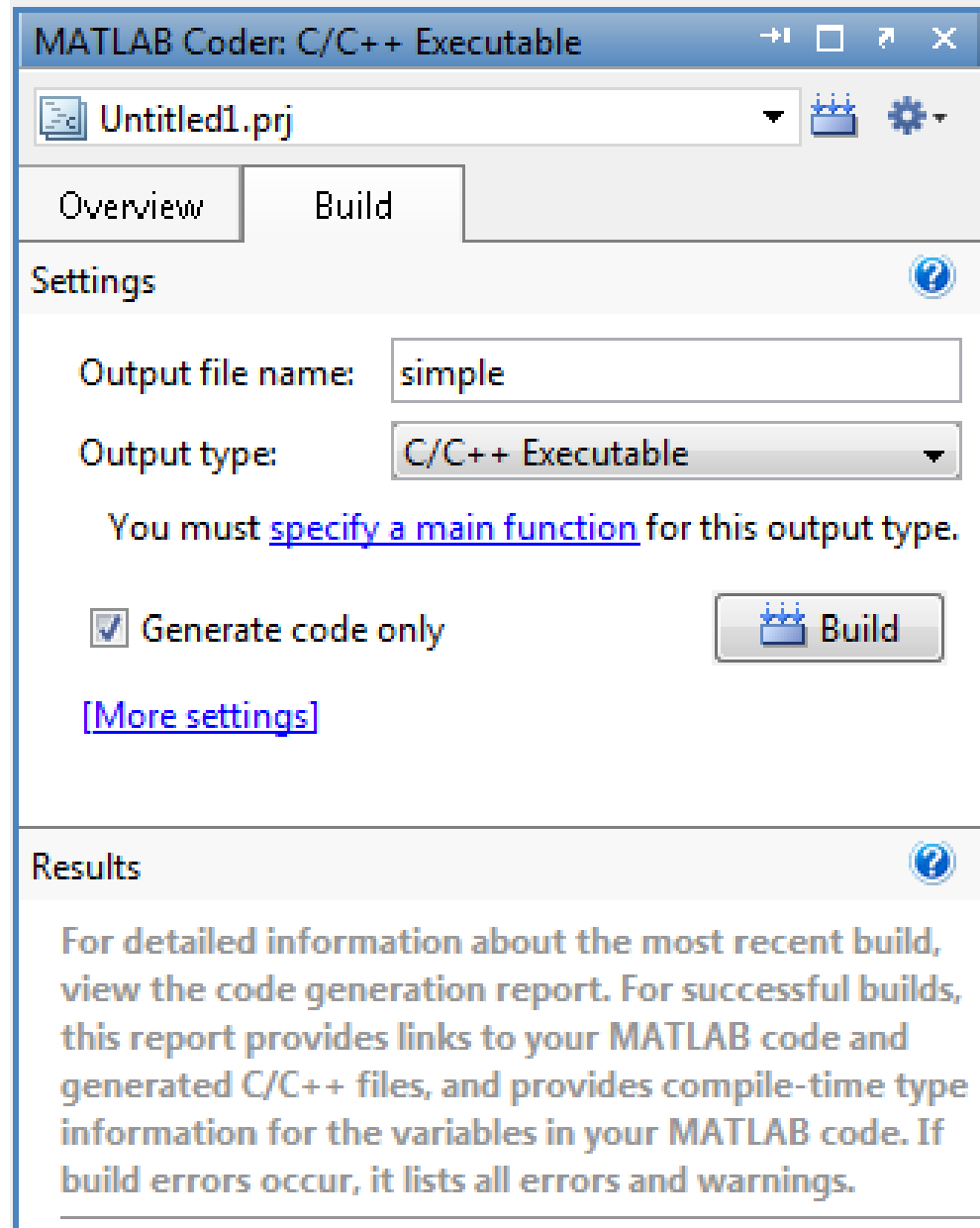
- ‘Define by example’ allows you to enter a Matlab expression and Matlab will guess at an applicable definition. Your mileage may vary.
- The two remaining options let us define a variable by type, or as a run-time constant.
- This part of the tutorial will manually define `h`, by type. We will arbitrarily define `h` as an unsigned 8-bit integer, `uint8 1x1`.

Click the 'Build' tab and choose the C/C++ Executable option.

The goal of this part of the tutorial is to create stand alone C code, so check the 'Generate code only' option.

Matlab warns us we need a 'main' function, but that is simply how C works.

Click the 'Build' button.



A 'codegen' directory will be created containing the required code

- Within the directory are more directories.
- Continue down the tree to `codegen\exe\simple` where you will find the source code.

What is all this stuff?

- The code generated by Matlab is very particular, defining all parameters as it sees fit.
- Because of this, you **will** need all the header.h files and additional source found within the codegen\exe\simple directory.
- Alternatively, you can redefine variables with conventional C definitions, although this will bypass work already done for you.

Example simple source code

```
/* Include files */
#include "rt_nonfinite.h"
#include "simple.h"

/* Type Definitions */

/* Named Constants */

/* Variable Declarations */

/* Variable Definitions */

/* Function Declarations */

/* Function Definitions */
uint8_T simple(uint8_T h)
{
    uint8_T z;
    if (h > 127) {
        z = MAX_uint8_T;
    } else {
        z = (uint8_T)(h << 1);
    }

    return z;
}

/* End of code generation (simple.c) */
```

In this example, if you wish to use standard headers (for maximum portability) instead of the custom headers generated by Matlab, you could use the standard `inttypes.h` header (`#include <inttypes.h>`) which defines `uint8_t` variable types and the maximum value `UINT8_MAX`.

Note that these are *not* used here, instead `uint8_T` and `MAX_uint8_T` are used, which would need replacement throughout the code in order to use standard headers.

Note that in addition to Matlab's use of custom definitions, the incorporated error handling of values larger than 7 bits, as well as the sophistication of the code—the code does not multiply `h` by two directly, instead it logically shifts `h` left by one bit. This accomplishes our goal (multiplication by two) with less computational expense.

Run the code

- The function we created is now able to be used inside of an arbitrary C routine.
- For the purpose of this tutorial, a simple main function was created solely to execute 'simple'.
- The code was compiled via MinGW using the Eclipse IDE (both free).



Project Explorer

main

main.c

```
/*
=====
Name      : main.c
Author    :
Version   :
Copyright : Your copyright notice
Description : main file utilizing MATLAB function 'simple'
=====
*/

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include "C:\Users\NP\Desktop\codegen\exe\simple\simple.c"

int main()
{
    uint8_t x;
    x=8;
    uint8_t result;
    result = simple(x);
    printf("%d multiplied by two is %d.\n", x, result);
    return 0;
}
```



Problems

Tasks

Console

Properties

Debug

<terminated> main.exe [C/C++ Application] C:\Users\NP\Desktop\codegen\exe\simple\main\Debug\m

8 multiplied by two is 16.

SUCCESS!

Doubling a number is probably not a complex operation you would use C to speed up your Matlab code.

This next example will use C to implement a Kalman filtering routine.

The files for this example can be found in the

C:\Program

Files\MATLAB\R2011b\help\toolbox\coder\
examples\kalman

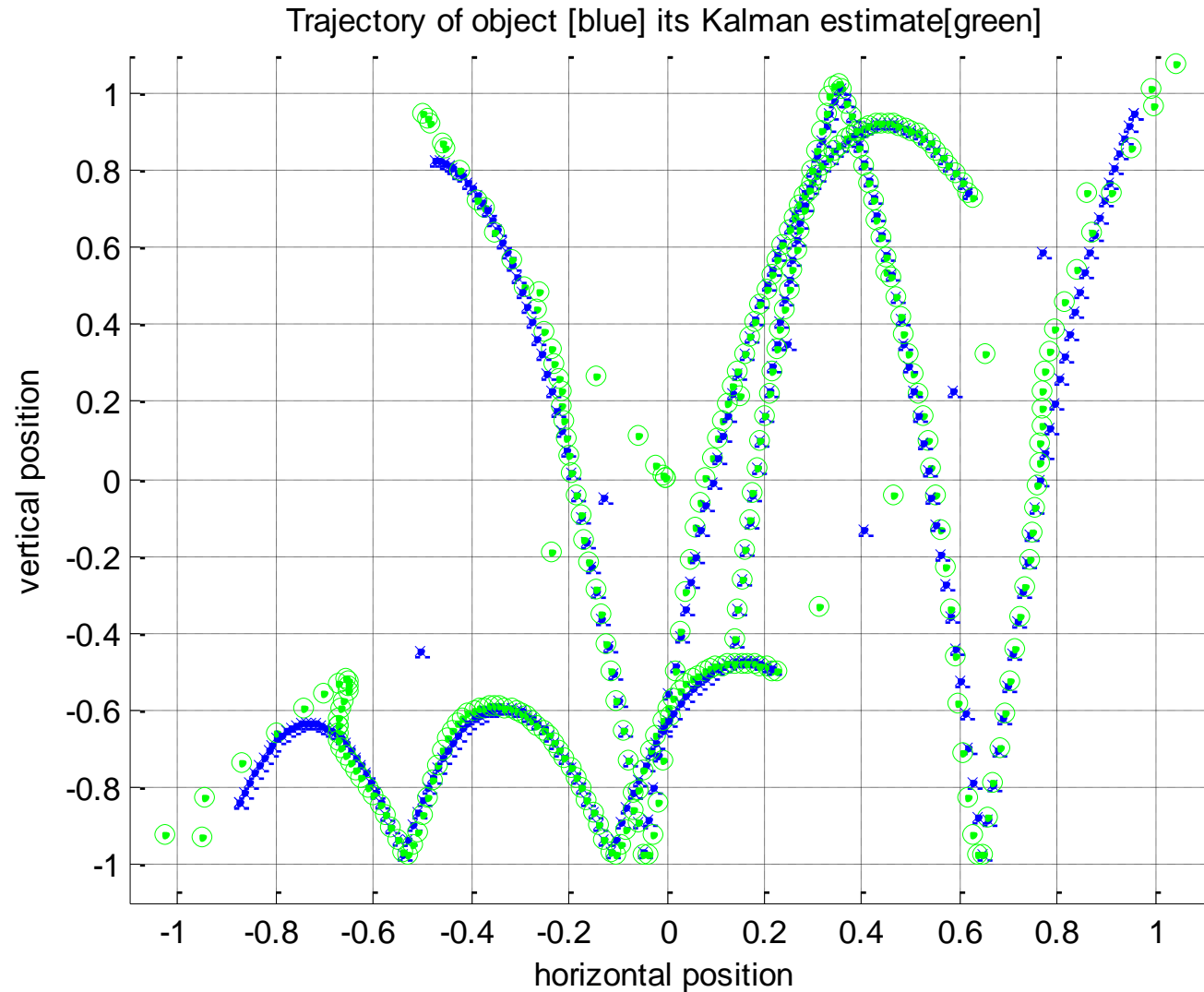
(or equivalent) directory. Note that this is *not* where MATLAB help files instruct you to find them.

The files used in this tutorial:

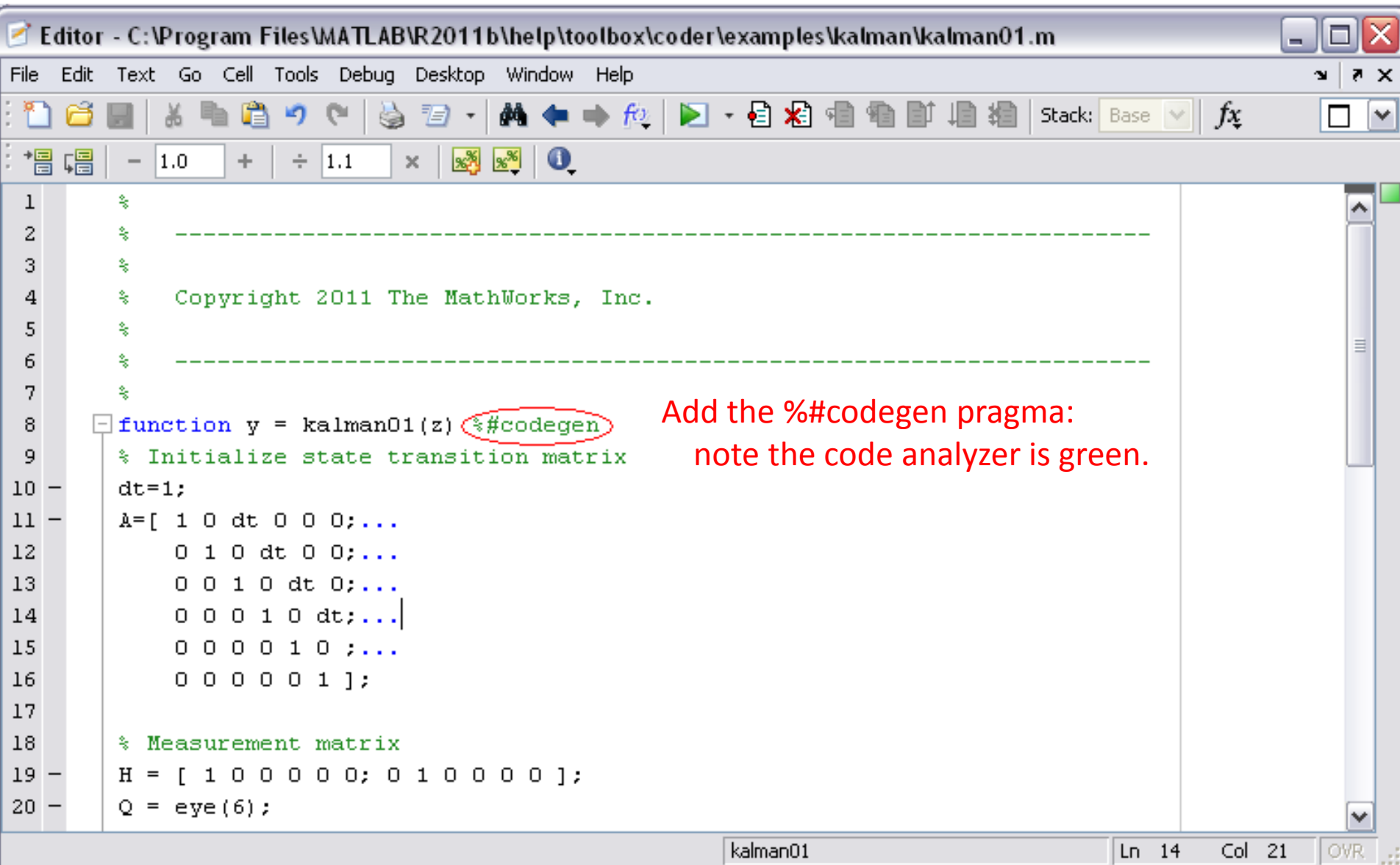
- `kalman01.m`
- `position.mat`
- Test scripts `test01_ui.m` through `test03_ui.m`
- `plot_trajectory.m`

- The m-file test01_ui.m calls on plot_trajectory.m to plot the objects trajectory as well as the estimated position.

Test01_ui produces the following plot:



Now we can see how suitable the code is for C conversion



Editor - C:\Program Files\MATLAB\R2011b\help\toolbox\coder\examples\kalman\kalman01.m

File Edit Text Go Cell Tools Debug Desktop Window Help

Stack: Base fx

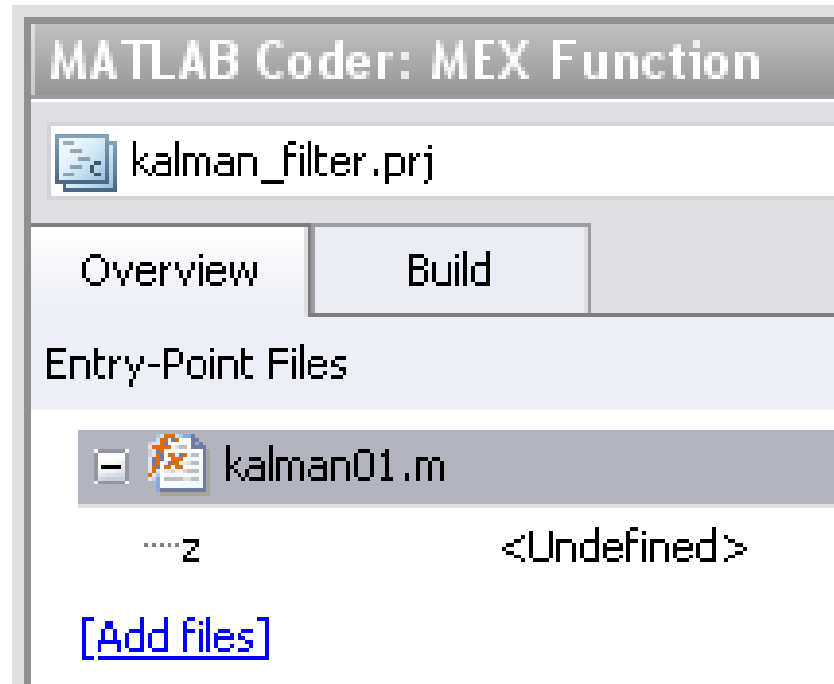
```
1 %  
2 % -----  
3 %  
4 % Copyright 2011 The MathWorks, Inc.  
5 %  
6 % -----  
7 %  
8 function y = kalman01(z) %#codegen  
9 % Initialize state transition matrix  
10 dt=1;  
11 A=[ 1 0 dt 0 0 0;...  
12     0 1 0 dt 0 0;...  
13     0 0 1 0 dt 0;...  
14     0 0 0 1 0 dt;...  
15     0 0 0 0 1 0 ;...  
16     0 0 0 0 0 1 ];  
17  
18 % Measurement matrix  
19 H = [ 1 0 0 0 0 0; 0 1 0 0 0 0 ];  
20 Q = eye(6);
```

Add the %#codegen pragma:
note the code analyzer is green.

kalman01 Ln 14 Col 21 OVR

File > New > Code Generation Project

- Overview tab>add files>kalman01.m



Note that the input variable, z , is undefined.

The latest version of MATLAB (2012a) has an autodefine types link.

This slick option allows you to Autodefine Entry-Point Input Types after adding a test file to the project:

- Select test01_ui.m and click Open.
- The test file, test01_ui.m, calls the entry-point function, kalman01.m, with the expected input types.
- In the Autodefine Entry-Point Input Types dialog box, click the Run button.
- The test file runs, plots the output, and infers that input z is double(2x1).
- Click Use These Types.
- MATLAB Coder assigns this type to z .
- The current documentation can be found online.

We don't have the latest version of MATLAB

This time, we will define z by example.

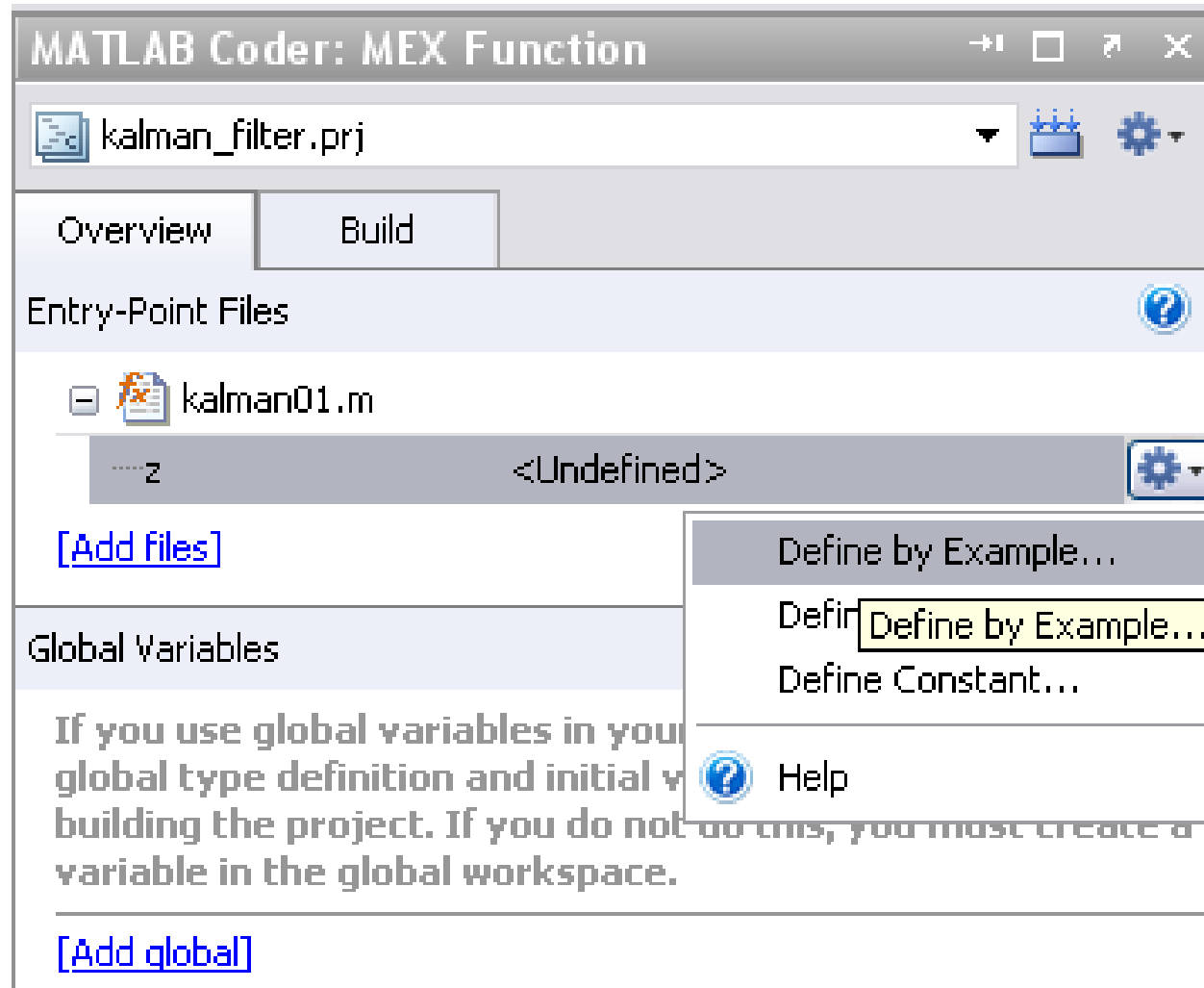
The variable z is a 2-d cartesian coordinate.

The file position.mat contains 310 of them.

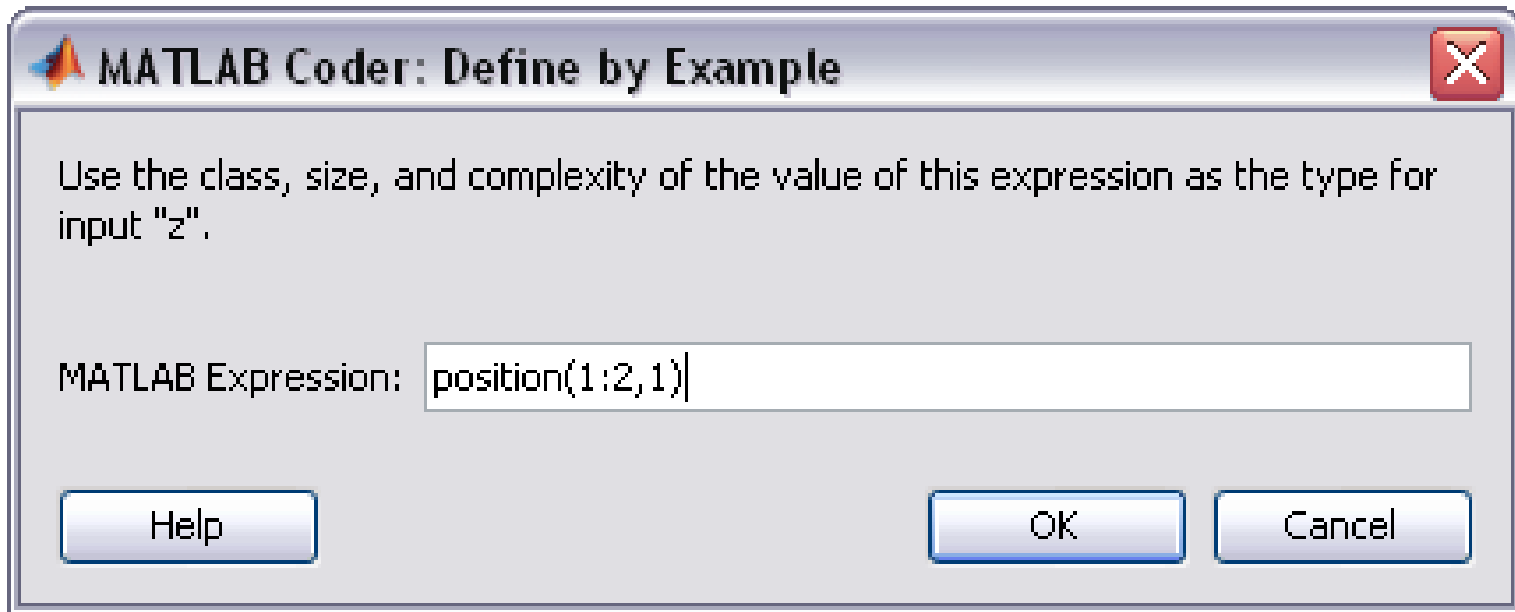
Load position.mat into your workspace:

```
>>load position.mat
```

Nothing new here



Feed Matlab one of the elements of positon.mat for the example



z is now a double precision 2 by 1 array,
`double(2x1)`

Now click the build icon

MEX is the default output type.

Once the build is complete, simply replace the function in your matlab code with the MEX function.

There you have it.

C:\Documents and Settings\AJ\Desktop\Work

```
1      % Figure setup
2 -    clear all;
3 -    load position.mat
4 -    numPts = 300;
5 -    figure;hold;grid;
6
7      % Kalman filter loop
8 -    for idx = 1: numPts
9          % Generate the location
10 -         z = position(:,idx);
11
12          % Use Kalman filter to
13 -         y = kalman01(z);
14
15          % Plot the results
16 -         plot_trajectory(z,y);
17 -     end
18 -     hold;
19
```

C:\Documents and Settings\AJ\Desktop\Workspace\test02_ui.m

```
1      % Figure setup
2 -    clear all;
3 -    load position.mat
4 -    numPts = 300;
5 -    figure;hold;grid;
6
7      % Kalman filter loop
8 -    for idx = 1: numPts
9          % Generate the location data
10 -         z = position(:,idx);
11
12          % Use Kalman filter to estimate the location
13 -         y = kalman01_mex(z);
14
15          % Plot the results
16 -         plot_trajectory(z,y);
17 -     end
18 -     hold;
19
```