CEE598WP

Topic 17: Advanced Topics in MATLAB

James Bittner

Introduction

The purpose of this chapter is to introduce various features and techniques available for data processing through MATLAB. The author of the chapter does not present expertise in computer science, but rather experience with practical solutions to common engineering problems. The following three sections cover general introductions to the topics of advanced data types, code optimization and graphical user interfaces. Further information on each topic is available through the excellent documentation provided by Mathworks.

Advanced Data Types

Advanced data types leverage optimal design characteristics in order to reduce the complexity of a section of code. Ultimately every bit of data is stored in a memory address, but humans need more abstract methods of indexing and organizing these bits. Standard Data Types (Integers, Doubles, Characters, Vectors/Strings) all exist to hold bits of information in an addressable location. As the volume of variables increases, another layer of abstraction is needed to maintain logical flow of code. Cells and Structs provide this additional layer of abstraction (Gdeisat and Lilley 2012).

Cells

In MATLAB, cells are a dynamic vector that points to memory locations. The primary difference between a vector and a cell is that a vector is an ordered section of memory, while a cell is a listing of a unique memory addresses. Upon changing size or type of a value in a cell, the listing can just point to a new memory address. The dynamic addressing ability allows a cell to accept multiple data types, and sizes without disrupting the remaining portion of the vector.

Structs

Structs present a great data type for organizing structured datasets. While cells are based on an underlying principal of rows and columns, structs are based on a tree layout. Structures begin with a starting base name, and then branch out to sub-structs or unique variables. The resulting information tree allows grouping based upon the intended use of each variable. For example if a pump is being modeled, the entire physical dimensions and hydrologic flow data can be stored all within the same overall structure. Then later if a function is written, the full pump struct can be passed as a single argument into the function, making available all the sub-variables.

From the programmer perspective, a struct behaves exactly like a single variable. Therefore, arrays of structs can easily be generated to keep track of similar data structures. For example if a fleet of pumps are available, the previous function can easily run code using any pump structure present in an array of structs. This allows for code to be written once and applied many times without any modification.

Speed Optimization

The execution time of any analysis is critical to determining the flexibility a researcher has to modify the parameters of the analysis. If a code takes 1 hour to run, a parametric analysis is limited to less than 100 iterations per work week. While, if the same code takes 1 minute to run the analysis is limited to 6000 iterations per work week. In the second case, considerably more factors can be statistically analyzed over the course of a research period.

Time Trade-Off

Since execution speed is a great ally in discovery, logic defines that we should use the fastest programming language available. Unfortunately, the speed of analysis is also limited by the speed of the researcher to generate (program) the analysis. In modern times, the cost of labor far exceeds the cost of computation.

MATLAB is a tool that provides a balance between highly optimized execution and simplified code construction. Any tool can be implemented in inefficient methods, thus it is the programmers responsibility to use engineering judgment to determine the optimum time trade-offs. This premature optimization concept was once summarized by a famous computer scientist when he stated:

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil" – Donald E. Knuth, 1974

Identifying Inefficient Code

Several methods of localization of slow code are available. Two very common methods are performing a simple stop-watch procedure or performing an integrated execution profile (Gdeisat and Lilley 2012). The simplest method is a stop-watch procedure through the use of the 'tic'/'toc' commands. The 'tic'/'toc' commands, when placed around a section of code will record the raw time elapsed between function calls. This rudimentary procedure allows sub-second inspection of algorithm efficiency.

Another method is the built-in MATLAB profiler. The easiest way to invoke the profiler is to select the 'Run & Time' instead of the 'Run' button in the Editor Window. MATLAB will run the code along with a detailed trace and analysis procedure. Note, the procedure will slow the overall execution time due to recording all the internal timing details. At completion of the run, a profile window will be displayed with results. The results will include the time spent on each line of code, and the number of times each line of code was executed. This timing evaluation method allows for inspection of complex code that may include multiple script files or functions.

Common Solutions to Inefficient Code

Many solutions to inefficient code exist across the spectrum of computation. With a direct focus on MATLAB programming the preferred methods of optimization, in order, are matrix math, build-in functions, and parallelization. As computing advances, these methods of optimization are apt to shift in priority due to new distributed approaches.

Graphical User Interfaces

Graphical User Interfaces (GUIs) have had a major impact with how humans display and interact with computational systems. Often in engineering applications the output numbers are the only product of practical use. Unfortunately, in a static input/output environment, numerical stability is hard to intrinsically judge. While in a graphical environment, sliding an interactive input allows for almost instantaneous feedback. As an example, humans were able to fly planes long before perfecting autopilot. Moving forward, implementing MATLAB in an interactive environment may prove to be a valuable research data exploration tool.

Execution Differences

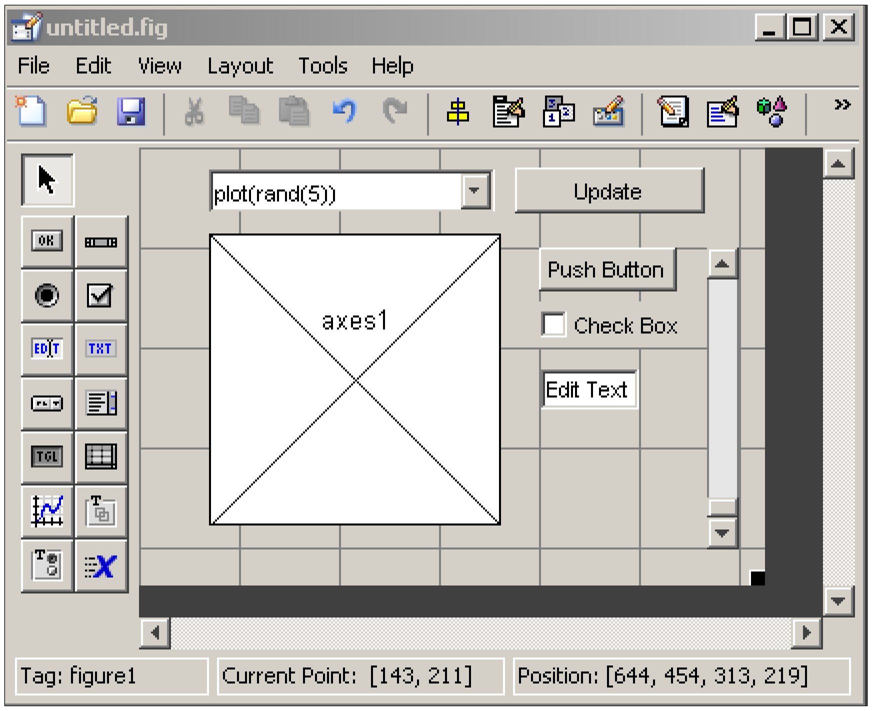
To move into the interactive environment, the layout of a basic MATLAB script must change. First, a common static script works with an input, computation and output workflow, while an interactive script must display the interface then constantly loop waiting for new inputs. Once an input occurs the interface needs to call a chunk of code, run the computation, output the workflow and return to waiting for new inputs.

Now instead of starting and ending, our script has to wait constantly. Luckily, this waiting function is something all computers do and can be handed off to the operating system. However, the programmer must handle the calling a specific chunk of code after the user input. To allow the interface to report an input to our script, we generate *callback* functions.

A *callback* function is linked from a specific interface action and takes care of performing the desired action (e.g. ButtonPush, ButtonReleased). These functions behave exactly like all other MATLAB functions and execute with the same set of locally defined variables. The initial graphical user interface script setup requires quite a bit of layout and function creation. Luckily, the folks at Mathworks have created a toolbox for generating GUIs called Graphical User Interface Design Environment (GUIDE).

Introduction to GUIDE toolbox

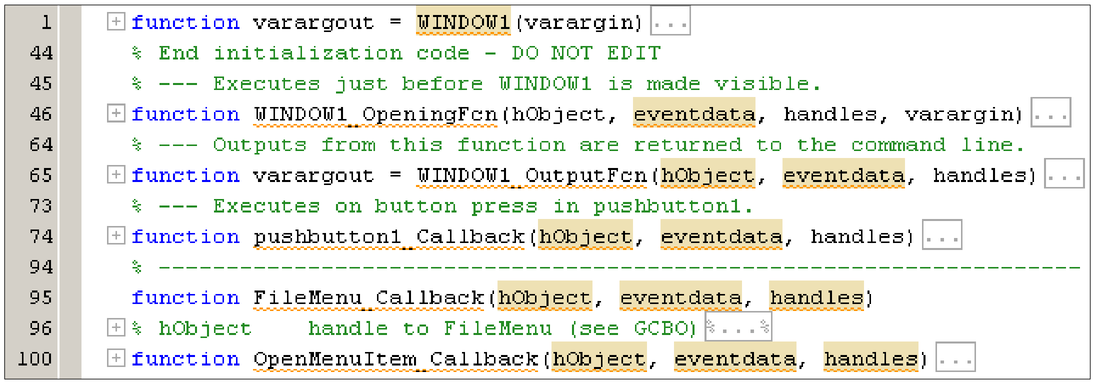
The GUIDE toolbox can be initialized by typing ‘guide’ on the MATLAB command line (Hunt et al. 2014). The guide toolbox contains a few starting templates to start off a project. This section will outline some of the major features of the ‘GUI with Axes and Menu’ starting template.



Tag: figure1 | Current Point: [143, 211] | Position: [644, 454, 313, 219]

Figure 1. MATLAB Graphical User Interface Design Environment

After selecting template, the GUIDE will load as seen in Figure 1. The design environment allows for general window layouts to be defined; additional interface features may be added from the toolbar on the left-hand side. The GUIDE environment can be revisited at any point by opening up the saved figure (.fig) file. The next step is to save the figure and GUIDE will generate a controller script (.m) file based upon the layout.



```
function varargout = WINDOW1(varargin) ...
% End initialization code - DO NOT EDIT
% --- Executes just before WINDOW1 is made visible.
function WINDOW1_OpeningFcn(hObject, eventdata, handles, varargin) ...
% --- Outputs from this function are returned to the command line.
function varargout = WINDOW1_OutputFcn(hObject, eventdata, handles) ...
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles) ...
% ----------------------------------------------------------------------
function FileMenu_Callback(hObject, eventdata, handles)
% hObject    handle to FileMenu (see GCBO) %...%
function OpenMenuItem_Callback(hObject, eventdata, handles) ...
```

Figure 2. MATLAB GUIDE Generated Script

An example script file is displayed in Figure 2, outlining the GUIDE generated *callback* functions. To integrate existing models with the graphical user interface, a model can be initialized in the OpeningFcn and any intermediate results can saved to the handles struct. Further, when user input executes a specific *callback* function, the model can be modified/recalculated and the interface display updated. To run a GUI program, the user must execute the controller script just as any existing MATLAB script.

Innovative teaching

During this topic session as a class we attempted a flipped class style environment (Brame 2013). A flipped class environment provides an introduction to main concepts through reading or viewing material prior to the classroom meeting, while in the classroom meeting focus is directed towards synthesizing the presented information. In one study, student's ability to answer multiple choice questions improved from 41 +/- 1% for a control group, to 74 +/- 1% for a flipped classroom environment (Brame 2013). These results are quite optimistic, but perhaps providing additional tools for material exposure will allow people with multiple learning types to flourish.

References

Brame, C., (2013). Flipping the classroom. Vanderbilt University Center for Teaching. Retrieved Wednesday, December 3, 2014 from http://cft.vanderbilt.edu/guides-sub-pages/flipping-the-classroom/.

Gdeisat, M., & Lilley, F. (2012). MATLAB® by Example: *Programming Basics*. Newnes. Elsevier. Londron, UK. doi:10.1016/B978-0-12-405212-3.00011-6

Hunt, B. R., Lipsman, R. L., & Rosenberg, J. M. (2014). *A guide to MATLAB : for beginners and experienced users*. Cambridge University Press. doi:10.1017/CBO9780511791284

Mathworks (2014). MATLAB. Computer Program. Mathworks, 2014a.