Training Unit 7: Automatic Code Generation

1 INTRODUCTION TO TRAINING UNIT

After starting with Matlab's code generation and getting some hands-on experience with MEX files and EXE files in training unit 6, you will continue to deepen your knowledge of code generation in Matlab and Simulink.

2 PREPARATION

2.1 Get all necessary Add-on packages

- For this unit you will need the following MathWorks Add-ons toolboxes:
  - Matlab Coder
  - Matlab Compiler
  - Simulink Coder
  - Embedded Coder
- If you went through the installation process as described in the last training unit, you should already be good to go for this training unit.
- In case you are having problems with the initialisation, be sure to check out section 2 in the last training unit description (TU6) for an in-depth explanation on how to set-up your Matlab / Simulink installation correctly

3 REPORT

- The report shall be a continuation from the last training unit. Therefore, be sure to continue the documentation within the live script, that you already started last week.
- The report for this training unit is based on your **Matlab live script** and **all your files** that are created during the training unit, that you will start from the first task. (zip your working directory and upload it to Moodle in the **training unit 6** section)
- Make sure to structure your work in the live script using the available editor tools (structure your code in section, add headers to your sections, table of contents, use **bold** or *italic* formats, bullet points, etc.)
- As you will consistently add new features to your live script, make sure to add new sections to each task, to provide a well-documented and structured live script. This will require you to copy-paste some of your code throughout the document most likely. Use the **subtask number** of this training unit description and **header** for your live script as **section header** for a more **consistent documentation**
- Try to make use of UI-controls throughout the script, to generate an interactive script
- Be sure to check the correct function of your scripts before uploading it. If your script **does not work**, you **will not receive any points** for this training unit.
- Make sure, that the script shows **your work** on this training unit task. Using solutions from other colleagues will result in zero points for you and your colleague.
- **Caution (!)** be sure to check the **due date** that is set for uploading the files on Moodle!
- **Additional information regarding the upload: Upload this report to the training unit 6 section. The training unit 7 will not have the possibility to upload any files. Thank you!**

4 LABORATORY WORK

4.1 Generating a simple Matlab function

- Please refer to document "**training unit 6**" for a detailed task description

4.2 Generating a Matlab executable function from your Fibonacci function

- Please refer to document "**training unit 6**" for a detailed task description

4.3 Function performance testing

- Please refer to document "**training unit 6**" for a detailed task description

4.4 Generating a windows executable (*.exe) file from your Fibonacci Matlab function

- Please refer to document "**training unit 6**" for a detailed task description

4.5 Using the command line for MEX code generation

- Even though Matlab's coder App is a very intuitive way to generate code from your Matlab functions, it can be slow. Especially if you perform small changes in your Matlab function, it requires several steps to start the code generation process
- To accelerate the process of code generation, let's use Matlab's command line from now on to initiate the code generation process in Matlab
- You have your Fibonacci m-function still available in your workspace
- Now go into your Matlab Command window and type:

```
coder.screener('YOUR_FIB_FUNC.m')
```

  - Where 'YOUR_FIB_FUNC.m' is the name of your Fibonacci function
- Check the output of this function call (coder.screener) and document your findings
- To generate a MEX function use following command:

```
codegen YOUR_FIB_FUNC.m
```

- Most likely you will receive an error message, as Matlab has problems to determine the data type of the input parameter **n** from your function. There are 2 possibilities to resolve this issue:
  - Specify the functions input parameters using the arguments section block
    - Read the description of this block and add it to your function
  - The second possibility is to specify the input parameters of your m-function using the codegen command with the additional -args option
    - The -args option lets you specifiy all input arguments regarding their size and type
    - Read the description of the codegen command change your function call with parameter specification
- Choose one of the above possibilities to specify the function input parameters and code your Fibonacci function using Matlab's command line
- This way, it is very fast to generate new MEX functions, without the need of going through the Coder App process

4.6 Using the command line for EXE code generation

- To speed-up the code generation process also for the binary file creation, the command line in Matlab can be used here as well
- You have your Fibonacci m-function still available in your workspace
- Now go into your Matlab Command window and type:

```
exe_cfg = coder.config('exe')
```

- This will generate an exe config file and outputs all options in your command window
  - Be sure to look at the output of this function call and check the default settings
  - Click on the **exe_cfg** data type in your workspace
    - This will open an UI representation of the config file
    - Does it look familiar? Have you seen this structure before?
- Now you can try to start code generation by using following command:

```
codegen -config exe_cfg YOUR_M_function
```

- Was the code generation successful? If not, why did it not work? Check the code generation report for more details in case of failure
- When starting the code generation, did you receive a warning message in the Matlab command window? Does this warning correlate with the error message from the code generation report?
- When you compare the above steps, with the steps from TU6 where you used the Matlab Coder App, what did you do previously, to successfully generate code? What is required to be able to generate a binary file that can be executed in windows?
- (Hint) Add your modified main.c function to be able to compile your m-function and generate code
  - Go through the "**exe_cfg**" variable "EmbeddedCodeConfig" in your workspace and find the option to specify your main function. Look out for the section "Custom Code"
- After improving your **exe_cfg** file, try your code generation again by:

```
codegen -config exe_cfg YOUR_M_function
```

- Note that you must specify the input argument type as you did for the MEX function coding before (refer to section 4.5).
  - If you chose the first option before (= using the "argument" block within your m-function) you do not need to specify the input arguments when calling the "codegen"-command as the parameter specification is included in the m-function.
  - If you chose the second option (= specifying the function input arguments using the -args option) you need to use this here as well
- After some seconds, the binary file should appear in your workspace
- Now you can generate new binary files, whenever you make changes in your m-function or your main function

4.7 Adapting your main.c function to accept input arguments when calling the function

- To be able to compare the performance between the m-function call, the MEX-function call and the binary-function call, you need to adapt your main.c function to accept input arguments from the calling process, instead of asking the user for the Fibonacci number
- Therefore, make a copy of your modified main.c function, which will need to be adapted now
- Your main.c function should already accept 2 arguments:
  - ***int argc***
  - ***char **argv***
  - check your main.c file, if it already accepts these two arguments
  - ***argv*** and ***argc*** are how command line arguments are passed to the main()-function in C
  - ***argc*** will be the number of strings pointed to by ***argv***
- Until now, they were not used. You asked the user of the binary program to input a number with a scanf() operation. This is not useful when comparing the execution speed of the function.
- Re-write your main.c program and check, which value the variable ***argc*** has
- If ***argc*** is not empty, print the string that ***argv*** holds on the position where ***argc*** is pointing to
  - Hint (!) Read through this example LINK, to get an idea of how to do it
  - Be sure to document the outcome of this small test
- After implementing the new functionality, test it in the windows terminal, if the input argument is accepted successfully by your program and the Fibonacci number is calculated correctly

4.8 Calling the binary file (*.exe) in your Matlab environment

- To be able to compare the function performance within Matlab, you need to be able to call the binary file from your live script
- Matlab needs the system()-command for calling external binary files
- Call the binary file using the system()-command and check if you can pass input arguments to it
- Do you receive the correct result as return value as well?

4.9 Function performance testing with the binary file

- Now you should be able to compare the performance of the binary file with the MEX file and the m-function
- As you already have some test benches from subtask 4.3, this task should be rather simple
- Make sure to start with small values of **n** again and compare the performance between all function calls and then increase the variable **n** and reduce the loop variable to avoid long calculation times.
  - What can you see? How is the performance of the **exe** file at small values of **n** and how does the performance change with higher values for **n**? Can you explain the behaviour?
  - Add some statistical plots that support your argumentation
  - For example, you can add a plot, that shows the process of increasing **n** and maybe the mean timing values or also the relation $\frac{t_{mean\_m\_func}}{t_{mean\_MEX\_func}}, \frac{t_{mean\_m\_func}}{t_{mean\_EXE\_func}}, \frac{t_{mean\_MEX\_func}}{t_{mean\_EXE\_func}}$
    - Add a new part of your script that iteratively increases **n**, while calling all three functions with the timing analysis
    - Start with a low loop_iteration number, just to be sure your code works fine
    - Then increase the number of loop iterations, to receive a good average value
- What can you say to the standard deviation and variance of the function calls (std() and var() in Matlab)?
  - High variances usually show that, there is a high variety in your timings array
  - Does the variance behave differently when considering m-function call, MEX-function call and binary-file call?

4.10 Simulink implementation of Fibonacci function in a Matlab function

- After successfully testing the Matlab implementation of the Fibonacci function, it is time to test the Simulink implementation of it
- Simulink might not be the “tool-of-choice” for the Fibonacci implementation, but for this training unit it is more important to see how equations can be modelled in Simulink and how simulations can be accelerated
- Build the Fibonacci function within a **Matlab function block** in a new Simulink Model
    - You can either choose to implement the function in a recursive manner or by using a for-loop representation within your Matlab function block
    - Got to the models “configuration settings” and set “Type” to “**Fixed-Step**” and the “Solver” to “**Discrete**” and chose a step size of 1 seconds and reduce the simulation time to 0.9 seconds
    - As input to your Matlab function block, consider using a constant block with the workspace variable “**n**” which is the Fibonacci number
    - Use an outport port to give back the Fibonacci numbers to the workspace
    - Test your function, if it works fine within Simulink (Check result with a Display block)
    - If it works, test it within your Matlab Command Window by using the sim(‘YOUR_MODEL’) command and check if you receive the correct Fibonacci numbers in your workspace (output from your Simulink model)
- Now it is time to test your Simulink Model using different “Simulation Modes”
    - Make 2 copies of your Simulink Model and append the name “accelerator” to one copy and “rapid_accelerator” to your other copy
    - Open the two Simulink models and change the Simulation mode to “accelerator” and “rapid accelerator” which is located beneath the “stop time edit filed” by opening the drop-down field
    - Now investigate, if the two modes accelerate the sim()-function call.
    - Make sure to use:
    - sim('YOUR_MODEL_acc.slx','SimulationMode','accelerator','StopTime', num2str(stop_time));
    - This command calls your simulation either in the ‘accelerator’ or ‘rapid’ mode and uses the number you define in your workspace for the variable ‘stop_time’ as stop time for your simulation (0.9)
    - This gives you now 2 parameters to add “computational load” to your simulation
        - Fibonacci number = **n**
        - Stop time = **stop_time**
    - Start with low Fibonacci numbers first and compare the results of different function calls
    - Increase the Fibonacci number to values between 30 and 45 (depending on your hardware)
    - Increase the load with longer runtimes of your simulation by increasing the **stop_time**
    - How is the behaviour of the accelerator and rapid accelerator mode?

4.11 Simulink implementation of Fibonacci function in a Stateflow function

- After successfully testing the Matlab function within a Simulink model, try to implement the Fibonacci function within a Stateflow Chart
- For that, copy your Simulink file, that was created first in task 4.10
- Delete the Matlab function and insert a so-called “Chart” block
- Connect the input and output to your chart and start to implement the Fibonacci function as for-loop implementation
    - This can be done by using a defined pattern like a “if-elseif-else” pattern for the if condition and using a for-loop pattern for the for-loop implementation of your Fibonacci function
    - The Fibonacci function call does not require steady states, as it is represented by an if-elseif-else flow control and a for-loop iteration. Therefore, it can be implemented just by using “junctions” and transitions.
    - Stateflow offers predefined patterns that can be used to help you include the flow control and for-loop iteration (see picture below, to see where to find it)
    - After clicking on one of those predefined patters, a pop-up window asks you for the conditions and actions in your flow chart / loop pattern. Make some tests with this automatic pattern creation and add an external test bench to test your implementation.
    - Whenever you need some variables for your Fibonacci function implementation, you can create “Local” Data stores by using the “Symbols Pane”. This opens a window on the right hand side of Simulink (usually) where all the inputs / outputs and local data stores are visible (refer to picture below). Whenever a variable is unknown / has not been created, a red mark appears besides the variable saying, “Undefined Symbol”. Press on the “Type” Column and select what kind of variable type it is “Input / Output / Local”. As soon as you specified the data type you can start your simulation.
- As soon as your implementation in stateflow works, test it within your Simulink model
- When you are certain that the stateflow representation works, perform the same tests as done before with the Matlab function block implementation. Make copies of your model and use accelerator and rapid accelerator mode and compare the different runs as before
- Compare the results of this implementation to the results of the previous implementation. Which one performs better? Are the differences noticeable? Document it in your live script.