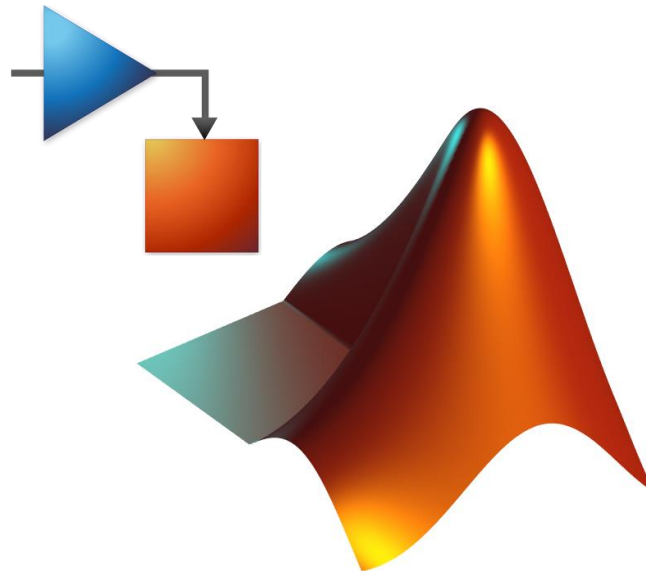


MATLAB / Simulink Lab Course

Code Generation



Objectives & Preparation “Code Generation”

- Which MathWorks products are covered?
 - ⇒ MATLAB Coder
 - ⇒ Simulink Coder

- What skills are learnt?
 - ⇒ How to generate C/C++ code from MATLAB code
 - ⇒ MATLAB code design considerations for automatic code generation
 - ⇒ How to generate C/C++ code from Simulink models
 - ⇒ Modelling considerations for automatic code generation and Simulink Model Advisor

- How to prepare for the session?
 - ⇒ Tutorials for the MATLAB Coder:
 - <https://de.mathworks.com/help/coder/getting-started-with-matlab-coder.html>

 - ⇒ Tutorials for the Simulink Coder:
 - <https://de.mathworks.com/help/rtw/getting-started.html>



Outline

1. Introduction

2. MATLAB Code Generation & Deployment

- MATLAB Coder GUI
- MATLAB Coder Command Line

3. MATLAB Coder: Requirements & Best Practices

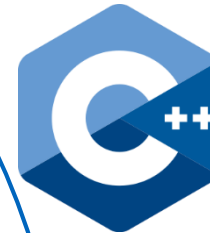
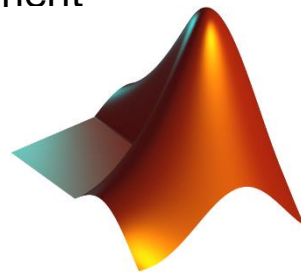
- Supported Features and Code Design Considerations
- Differences Between Generated Code and MATLAB Code

4. Simulink Code Generation & Deployment

- Targets and Target Files
- Code Generation / Build

5. Simulink Coder: Model Architecture & Design

- Modeling Considerations
- Model Advisor



Jeremy Kratz
(https://commons.wikimedia.org/wiki/File:ISO_C++_Logo.svg),
„ISO C++ Logo“

SparkFun Electronics
from Boulder, USA
(https://commons.wikimedia.org/wiki/File:Arduino_Uno_-_R3.jpg),
„Arduino Uno - R3“,
<https://creativecommons.org/licenses/by/2.0/legalcode>

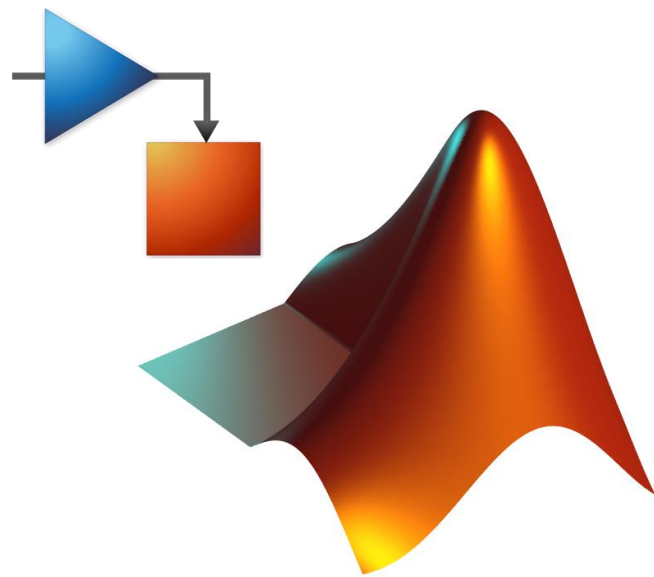


Altera Corporation
(https://commons.wikimedia.org/wiki/File:Altera_StratixIV_GX_FPGA.jpg),
„Altera StratixIVGX FPGA“,
<https://creativecommons.org/licenses/by/3.0/legalcode>



Monkhiker
(https://commons.wikimedia.org/wiki/File:B&R_Box_PC.jpg),
„B&R Box PC“,
<https://creativecommons.org/licenses/by-sa/3.0/legalcode>

1. Introduction



Jeremy Kratz
(https://commons.wikimedia.org/wiki/File:ISO_C++_Logo.svg),
„ISO C++ Logo“

SparkFun Electronics
from Boulder, USA
(https://commons.wikimedia.org/wiki/File:Arduino_Uno_-_R3.jpg),
„Arduino Uno - R3“,
<https://creativecommons.org/licenses/by/2.0/legalcode>



Altera Corporation
(https://commons.wikimedia.org/wiki/File:Altera_StratixIV_GX_FPGA.jpg),
„Altera StratixIVGX FPGA“,
<https://creativecommons.org/licenses/by/3.0/legalcode>



Monkhiker
(https://commons.wikimedia.org/wiki/File:B&R_Box_PC.jpg),
„B&R Box PC“,
<https://creativecommons.org/licenses/by-sa/3.0/legalcode>

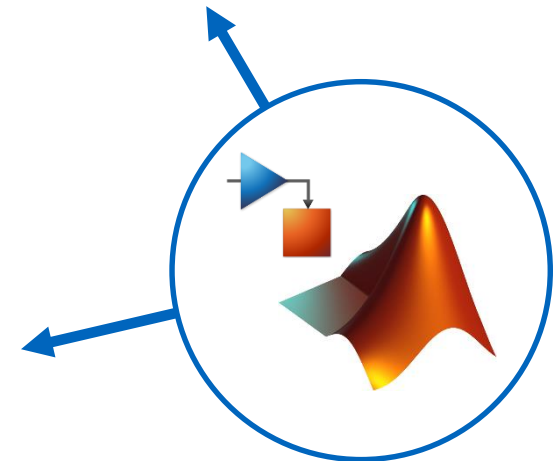
Introduction – Why Code Generation?

Why should one generate C/C++ code from MATLAB code or from Simulink models?

- To deploy algorithms developed in Simulink on embedded hardware
 - For instance, flight control laws implemented in Simulink can be deployed on a **flight control computer (FCC)**.
 - This approach is faster than programming the control laws by hand in the embedded software language and it reduces the risk of errors and bugs.
- To accelerate the execution of a program
 - **MATLAB code is generally slower than C/C++ code**
- To generate production code



SparkFun Electronics from Boulder, USA
(https://commons.wikimedia.org/wiki/File:Arduino_Uno_-_R3.jpg),
„Arduino Uno - R3“,
<https://creativecommons.org/licenses/by/2.0/legalcode>

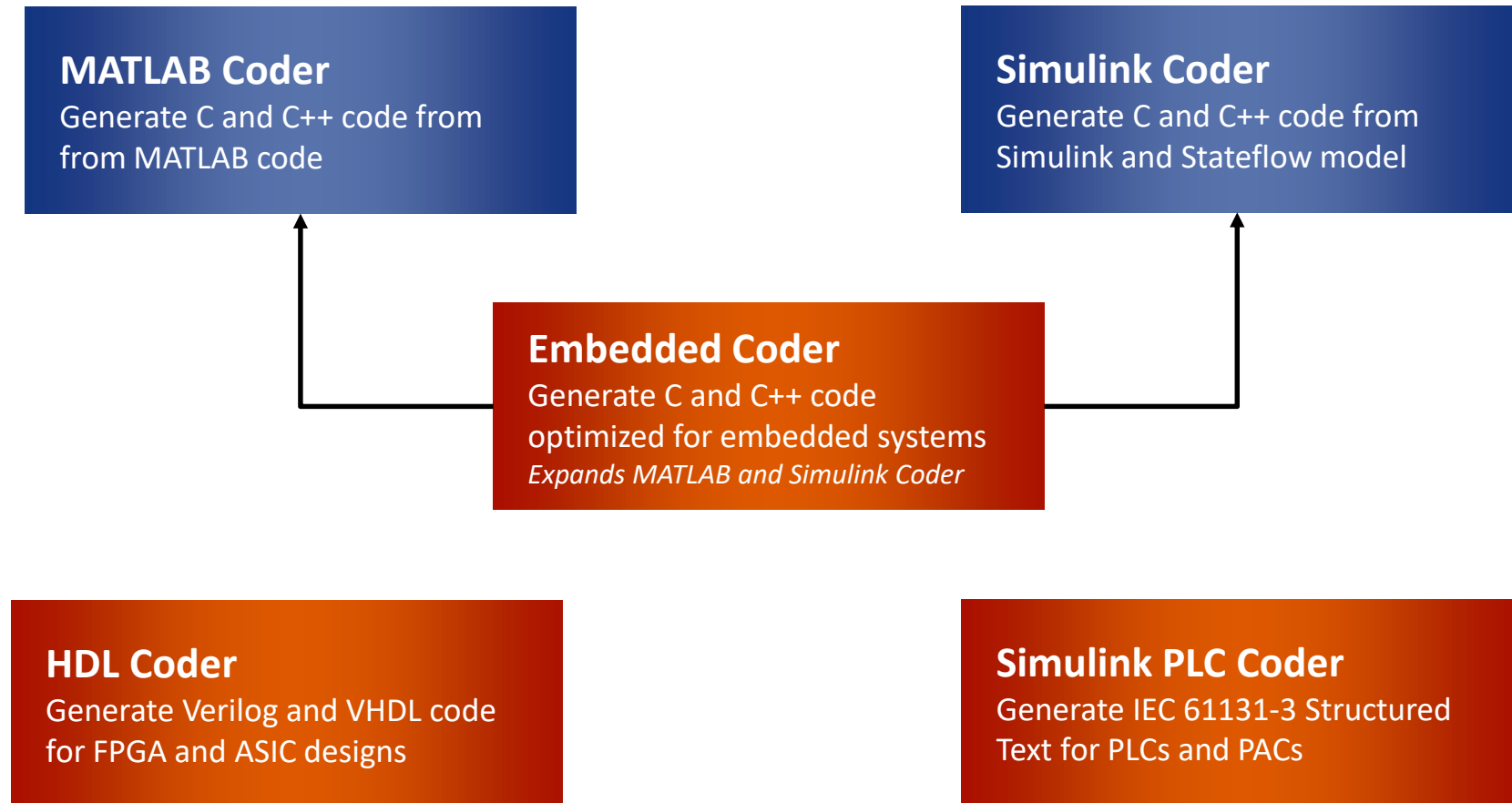


Introduction – Why Code Generation?

- Terminology and some more example applications...

Application	Target Environment	
Intellectual Property Protection	Host Computer	The same computer that runs MATLAB. Typically a desktop PC with a non-real-time operating system.
Accelerated Simulation		
Rapid Simulation		
System Simulation		
Rapid Prototyping	Real-Time Simulator	A different computer that uses a real-time operating system.
On-target Rapid Prototyping		
Production Code Generation	Embedded Microprocessor	A computer that you eventually disconnect from a host computer and run as a standalone computer as part of an electronics-based product.
Software-in-the-loop (SIL) simulation		
Processor-in-the-loop (PIL) simulation		
Hardware-in-the-loop (HIL) testing		

Introduction – Coder Overview



Introduction – MATLAB Coder and Simulink Coder

MATLAB Coder

- Generates standalone C and C++ code, libraries, DLLs and executables (*.exe) from MATLAB code.
- The generated source code is portable and readable.
- Supports most MATLAB language features, incl. functions and matrix operations.
- Can generate MATLAB executables (*.mex)
 - Accelerate computationally intensive portions of MATLAB code.
 - Verify the behavior of the generated code.

and

Simulink Coder

- Generates and executes C and C++ code from Simulink diagrams, Stateflow charts and MATLAB functions.
- The generated source code can be used for real-time and non-real-time applications, including simulation acceleration, rapid prototyping, and hardware-in-the-loop testing.
- The generated code can be tuned and monitored using Simulink.

Introduction – Compiler Setup

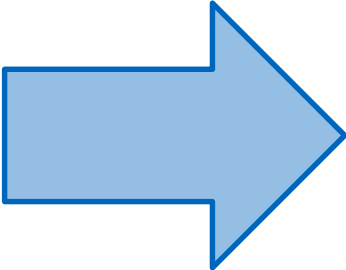
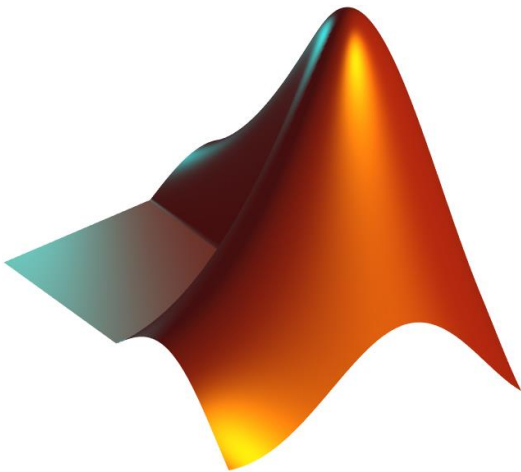
- A C or C++ compiler is required for both MATLAB and Simulink Coder.
- The MATLAB help provides a full list of supported compilers.
 - Microsoft Visual C++, for example, is compatible with most coding functionalities.
 - Alternatively, to download the MATLAB Support for MinGW-w64 C/C++ Compiler Add-On, visit: <https://de.mathworks.com/matlabcentral/fileexchange/52848-matlab-support-for-mingw-w64-c-c-compiler>
- MATLAB Coder automatically locates and uses a supported installed compiler.
- To choose a default compiler yourself, type

```
mex -setup
```

or, in case you want to specify a C++ compiler,

```
mex -setup cpp
```
- Then, follow through the short configuration process.

2. MATLAB Code Generation & Deployment



Jeremy Kratz
(https://commons.wikimedia.org/wiki/File:ISO_C++_Logo.svg),
„ISO C++ Logo“

SparkFun Electronics
from Boulder, USA
(https://commons.wikimedia.org/wiki/File:Arduino_Uno_-_R3.jpg),
„Arduino Uno - R3“,
<https://creativecommons.org/licenses/by/2.0/legalcode>



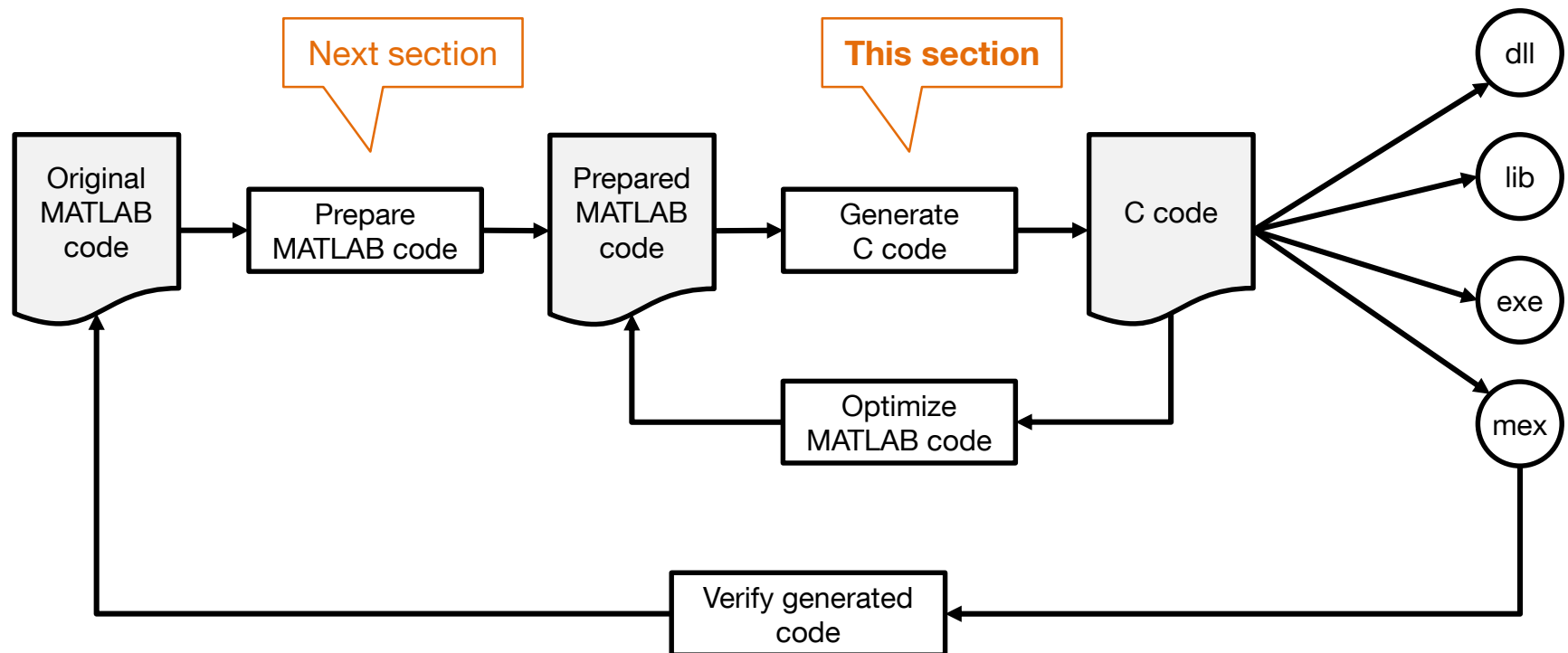
Altera Corporation
(https://commons.wikimedia.org/wiki/File:Altera_StratixIV_GX_FPGA.jpg),
„Altera StratixIVGX FPGA“,
<https://creativecommons.org/licenses/by/3.0/legalcode>



Monkhiker
(https://commons.wikimedia.org/wiki/File:B&R_Box_PC.jpg),
„B&R Box PC“,
<https://creativecommons.org/licenses/by-sa/3.0/legalcode>

MATLAB Code Generation – Workflow

Common Workflow for generating C code from MATLAB

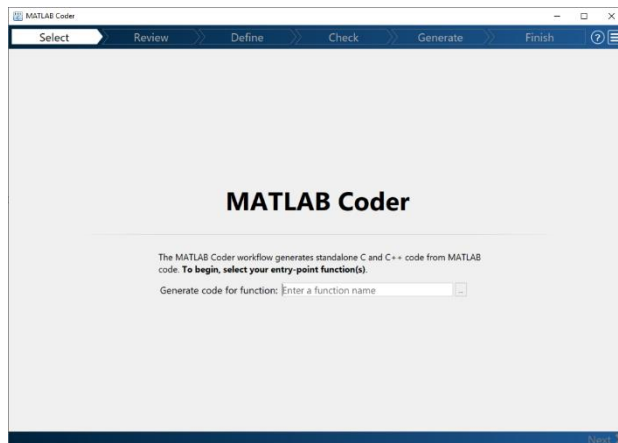


MATLAB Code Generation – MATLAB Coder GUI and Command Line

There are two ways to use the MATLAB Coder:

MATLAB Coder GUI

- + Easy to use
- + Settings stored in project file
- **Not automatable**

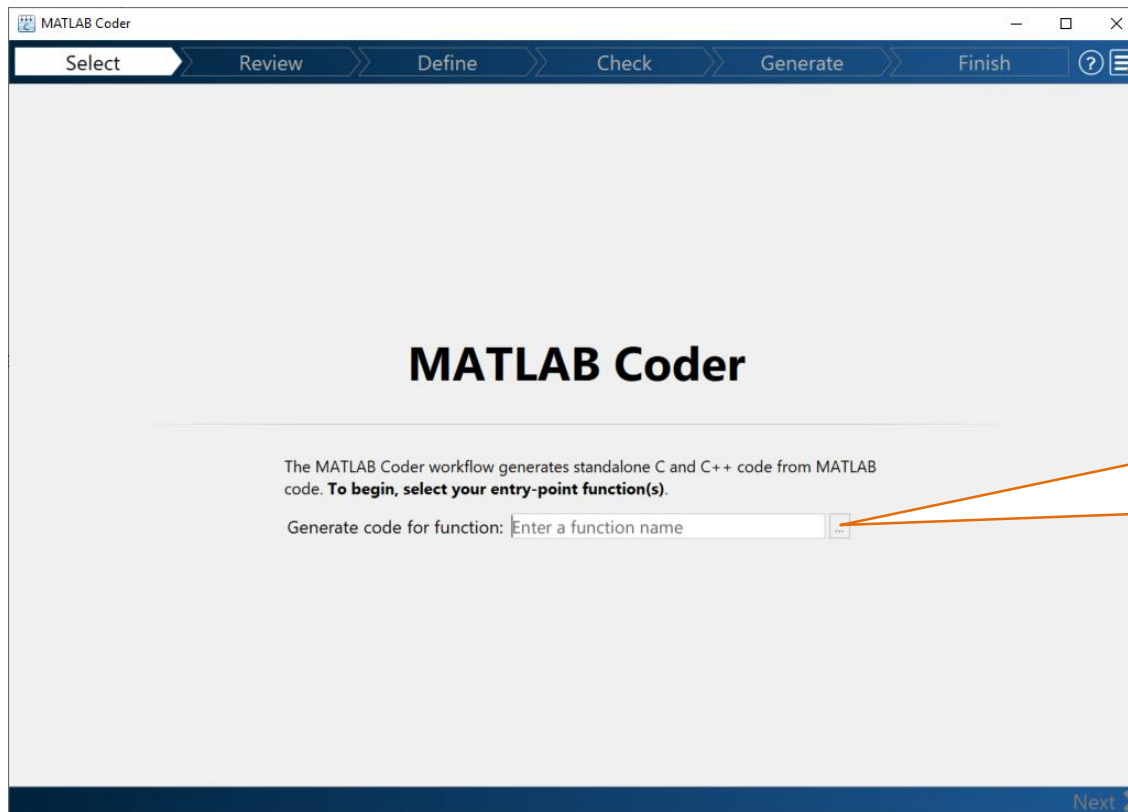


Command line: codegen

- + Faster for simple applications
- + Dynamic compilation within a MATLAB program (automatable)
- Syntax needs to be learned
- Complex function calls may be required for specific problems

MATLAB Code Generation – MATLAB Coder GUI

- On the MATLAB toolstrip tab *Apps*, under *Code Generation*, click the *MATLAB Coder* app icon.



This button leads to a menu that lets you open an existing project.

To create a new project, specify the entry-point function here. (This is the function that you call from MATLAB.) Then click *Next*.

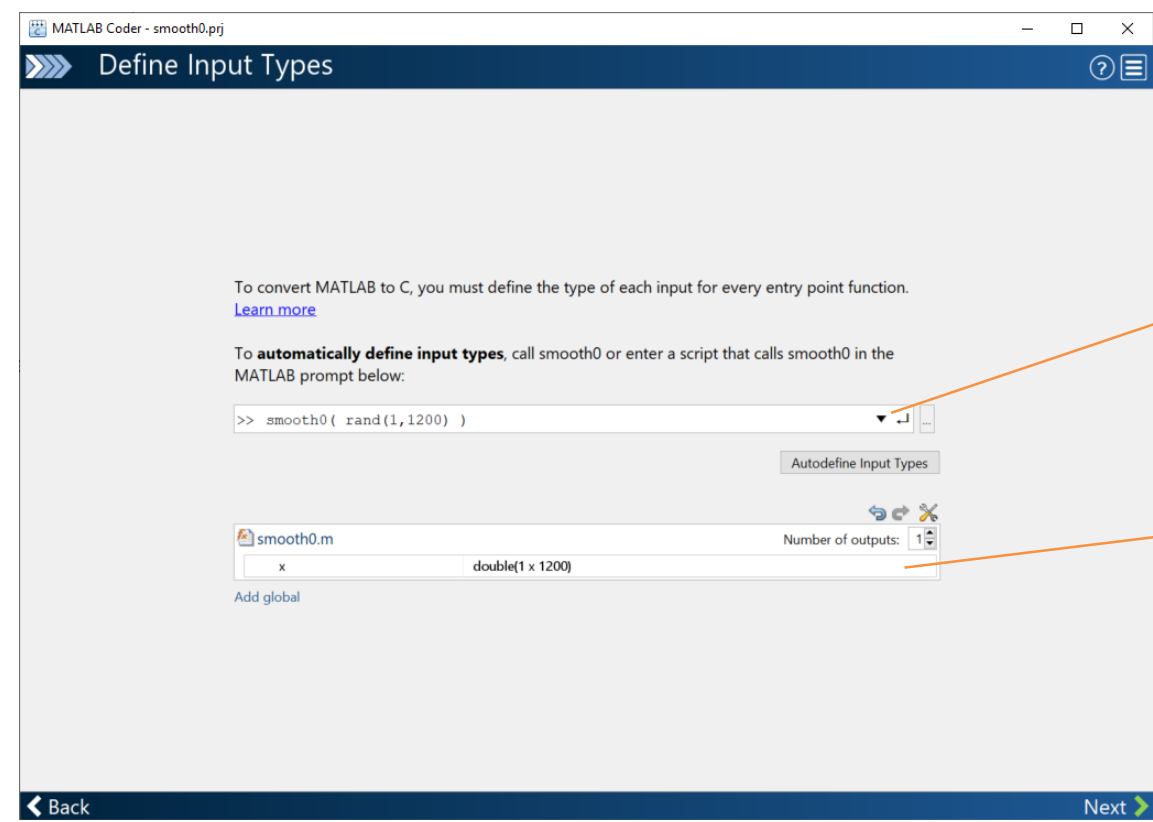
File names may not contain spaces!

MATLAB Code Generation – MATLAB Coder GUI

- Suppose you want to create code from a MATLAB function `smooth0.m`, which implements a simple moving average filter (see right side). This is the following window of the MATLAB Coder GUI:

```
function y = smooth0(x) %#codegen
y = zeros(1,length(x)-16);

% Calculate matrix multiplication
for iter = 1:length(x)-16
    y(iter) = mean(x(iter:iter+16));
end
```



- You can either have the input types defined automatically by calling the function, or
- you can manually specify all inputs:

double

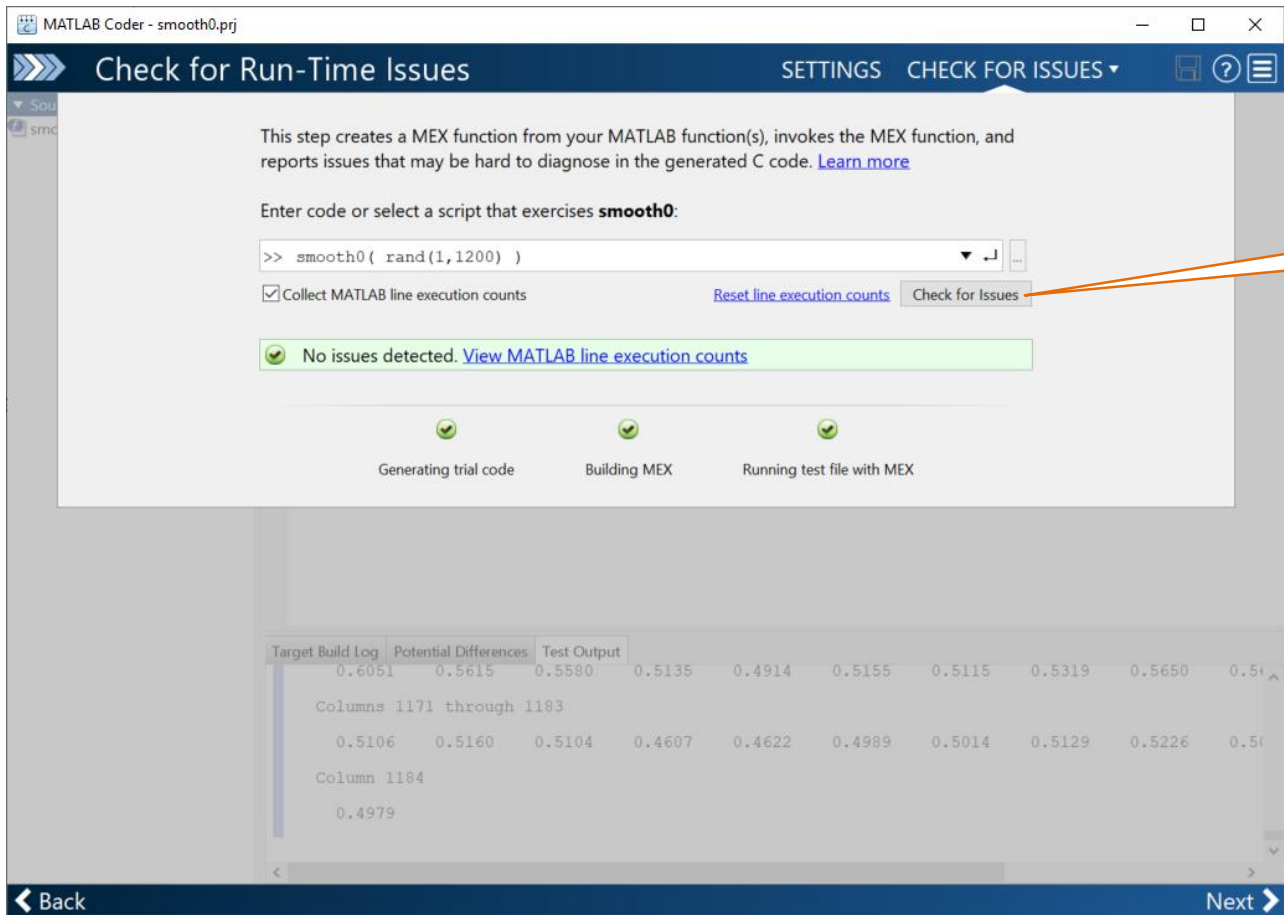
1 x 1	scalar
1 x n	vector
m x 1	vector
m x n	matrix

double 1 x 1200

1200	exactly 1200
:1200	up to 1200
:Inf	unbounded

MATLAB Code Generation – MATLAB Coder GUI

- In the following step, you can check for **run-time issues**.



Starts the check

This check is optional, so you may skip it by clicking Next.

MATLAB Code Generation – MATLAB Coder GUI

- Configuring settings is possible before actual code generation

Type of build to be generated. Here: MATLAB executable

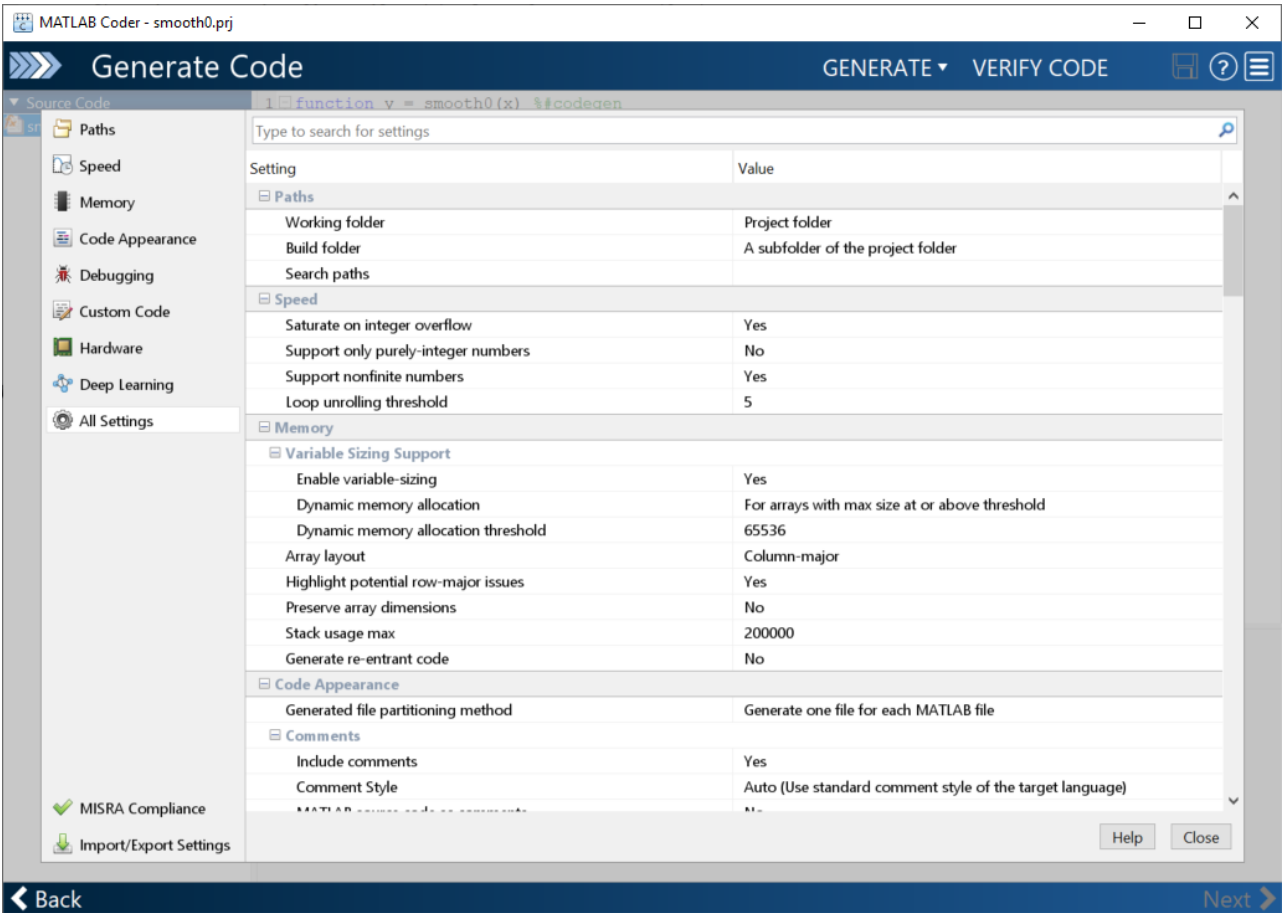
Click on the small arrow to open this settings page

Opens more settings for fine tuning of generated code in a new window
→ see next slide

Generates the code

MATLAB Code Generation – MATLAB Coder GUI

- More available settings depend on the build type



MATLAB Code Generation – MATLAB Coder Command Line

- Instead of using the GUI, the command line can be used as follows:

```
codegen smooth0 -args coder.typeof( rand(1,1200) )
```

- Generally, the (simplified) syntax is as follows:

```
codegen options function arguments
```

Options

MATLAB function name

Cell array of input arguments

- Example options:

```
-config:mex           % Generate MEX file (default)
-config:lib           % Generate static C/C++ library
-d output_folder      % Specify output folder
-o output_file_name    % Specify output file name
-report               % Generate code generation report
```

→ See next slide

MATLAB Code Generation – MATLAB Coder Generated Code

Generated MEX file

```
#include "rt_nonfinite.h"
#include "smooth0.h"
#include "smooth0_data.h"

void smooth0(const emlrtStack *sp, const real_T x[1200],
real_T y[1184])
{
    int32_T iter;
    real_T b_y;
    int32_T k;

    /* Calculate matrix multiplication */
    for (iter = 0; iter < 1184; iter++) {
        b_y = x[iter];
        for (k = 0; k < 16; k++) {
            b_y += x[(k + iter) + 1];
        }

        y[iter] = b_y / 17.0;
        if (*emlrtBreakCheckR2012bFlagVar != 0) {
            emlrtBreakCheckR2012b(sp);
        }
    }
}
```

Generated LIB file

```
#include "smooth0.h"

void smooth0(const double x[1200], double y[1184])
{
    int iter;
    double b_y;
    int k;

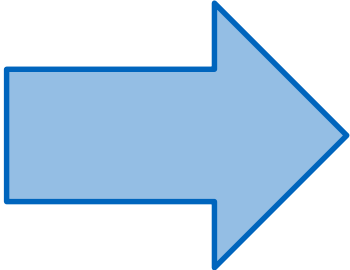
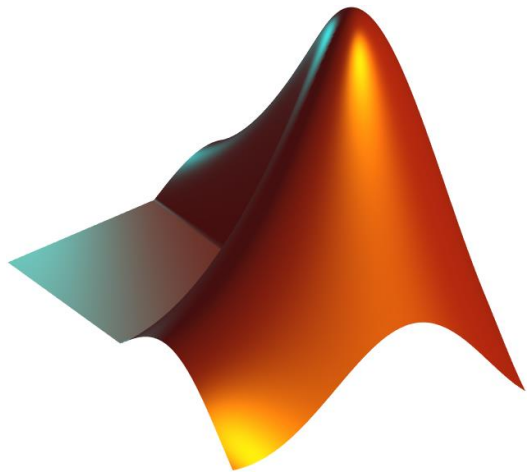
    /* Calculate matrix multiplication */
    for (iter = 0; iter < 1184; iter++) {
        b_y = x[iter];
        for (k = 0; k < 16; k++) {
            b_y += x[(k + iter) + 1];
        }

        y[iter] = b_y / 17.0;
    }
}
```

```
function y = smooth0(x) %#codegen
y = zeros(1,length(x)-16);

% Calculate matrix multiplication
for iter = 1:length(x)-16
    y(iter) = mean(x(iter:iter+16));
end
```

3. MATLAB Coder Requirements & Best Practices



Jeremy Kratz
(https://commons.wikimedia.org/wiki/File:ISO_C++_Logo.svg),
„ISO C++ Logo“

SparkFun Electronics
from Boulder, USA
(https://commons.wikimedia.org/wiki/File:Arduino_Uno_-_R3.jpg),
„Arduino Uno - R3“,
<https://creativecommons.org/licenses/by/2.0/legalcode>



Altera Corporation
(https://commons.wikimedia.org/wiki/File:Altera_StratixIV_GX_FPGA.jpg),
„Altera StratixIVGX FPGA“,
<https://creativecommons.org/licenses/by/3.0/legalcode>



Monkhiker
(https://commons.wikimedia.org/wiki/File:B&R_Box_PC.jpg),
„B&R Box PC“,
<https://creativecommons.org/licenses/by-sa/3.0/legalcode>

MATLAB Coder Requirements & Best Practices – Supported MATLAB Language Features

- N-dimensional arrays
- Matrix operations, including deletion of rows and columns
- Variable-sized data
- Subscripting
- Complex numbers
- Double-precision, single-precision, and integer math
- Fixed-point arithmetic
- Program control statements if, switch, for, while, and break
- Arithmetic, relational, and logical operators
- Persistent and global variables
- Structures, cell arrays, tables and MATLAB classes
- Function handles
- Anonymous, recursive, and nested functions
- Subset of MATLAB toolbox functions
- Variable length input and output argument lists
- ... <https://de.mathworks.com/help/simulink/ug/matlab-language-features-supported-for-code-generation.html>

An incomprehensive list

Not supported:

- Scripts
- Implicit expansion
- GPU arrays
- datetime arrays
- Time series objects
- Java
- Map containers
- try/catch statements



As of MATLAB Version R2019b, refer to

- Data types

- Array sizing

- Memory

- Static memory: + better speed - higher memory usage
- Dynamic memory: + potentially less memory usage - time required to manage memory

- Speed

- Choose a suitable C/C++ compiler
- Consider disabling run-time checks



<https://de.mathworks.com/help/simulink/ug/design-considerations-when-writing-matlab-code-for-code-generation.html>

MATLAB Coder Requirements & Best Practices – Differences Between Generated Code and MATLAB Code

Code generation inevitably includes code optimization, which intentionally causes the generated code to behave differently, i.e., more efficiently. Therefore, the results of the generated code may vary from the results of the original MATLAB code!

To prevent unwanted code behavior, keep the following (possible) differences in mind and address them where necessary.

- Order of evaluation in expressions

Generated code does not enforce order of evaluation in expressions. However, for expressions with side effects it is important to retain the order of evaluation (e.g. persistent/global variables, display data, write to files)

```
A = f1() + f2();
```

If `f1()` needs to be called before `f2()` to produce the desired result, rewrite the expression

```
A = f1();  
A = A + f2();
```

- Character size

—————> Do not perform arithmetic with characters

MATLAB: 16 bits; Generated code: 8 bits (standard size for e.g. C)

- Size of variable-size N-D arrays


—————> Use the two-argument form of `size`

Source: Modified from

<https://de.mathworks.com/help/simulink/ug/expected-differences-in-behavior-after-compiling-your-matlab-code.html>



MATLAB Coder Requirements & Best Practices – Differences Between Generated Code and MATLAB Code

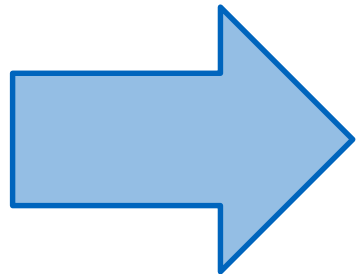
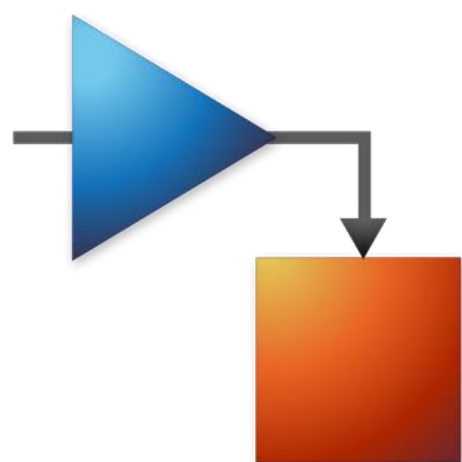
- Size of empty arrays 
 - Instead of `size`, use `isempty`
 - Instead of `x=[]`, create an empty array `x` using `zeros`
- Termination behavior
 - For instance, infinite loops are removed if they do not have side effects.
- Floating-point numerical results may be different
 - When computer hardware uses extended precision registers
 - For certain advanced library functions
 - For implementation of BLAS library functions
- NaN and infinity patterns
 - The generated code might not produce exactly the same pattern of NaN and inf values as MATLAB code when these values are mathematically meaningless.
- Complex numbers, negative zero
- ... For a complete list and possible workarounds, refer to the link below

Source: Modified from

<https://de.mathworks.com/help/simulink/ug/expected-differences-in-behavior-after-compiling-your-matlab-code.html>



4. Simulink Code Generation & Deployment



Jeremy Kratz
(https://commons.wikimedia.org/wiki/File:ISO_C++_Logo.svg),
„ISO C++ Logo“

SparkFun Electronics
from Boulder, USA
(https://commons.wikimedia.org/wiki/File:Arduino_Uno_-_R3.jpg),
„Arduino Uno - R3“,
<https://creativecommons.org/licenses/by/2.0/legalcode>

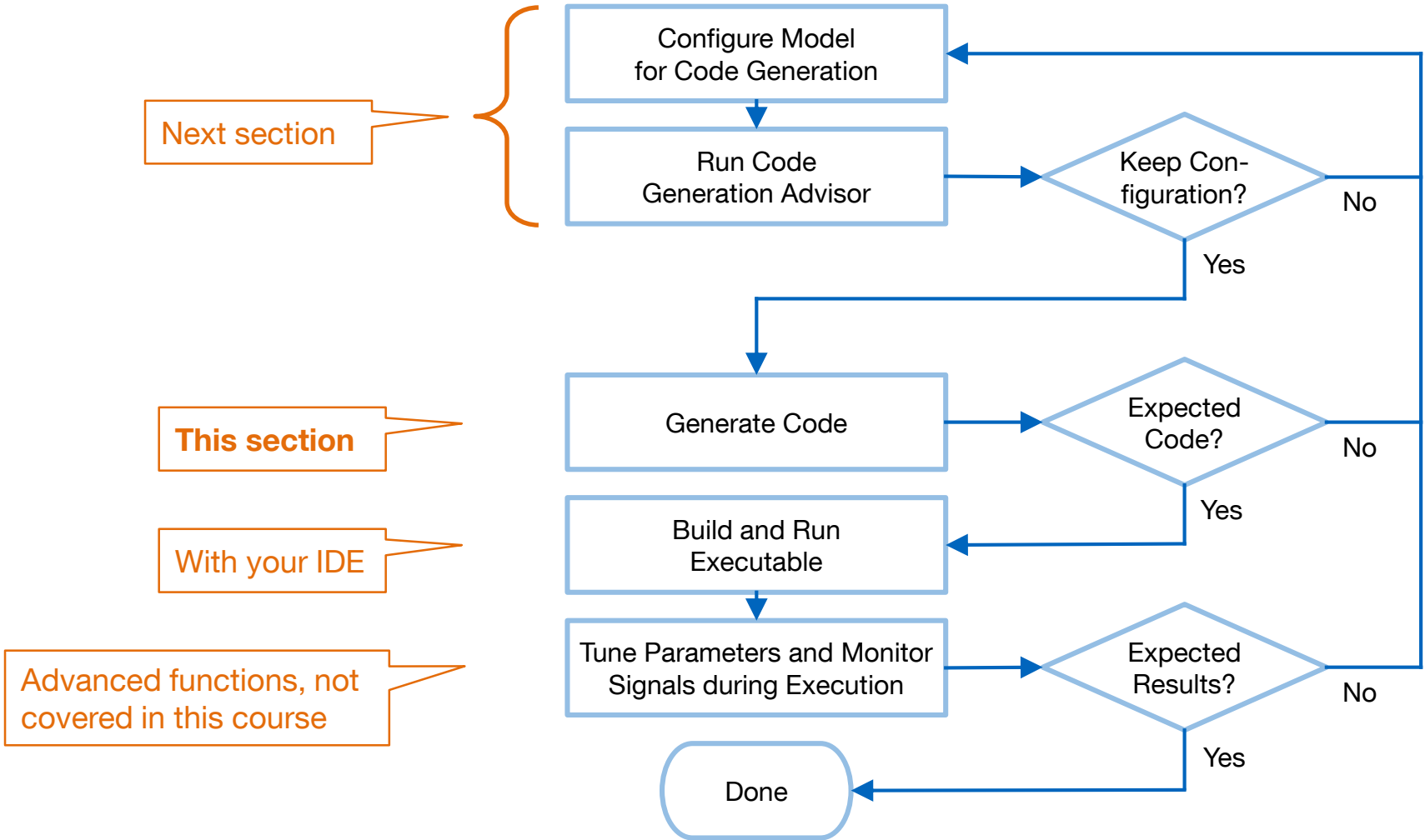


Altera Corporation
(https://commons.wikimedia.org/wiki/File:Altera_StratixIV_GX_FPGA.jpg),
„Altera StratixIVGX FPGA“,
<https://creativecommons.org/licenses/by/3.0/legalcode>



Monkhiker
(https://commons.wikimedia.org/wiki/File:B&R_Box_PC.jpg),
„B&R Box PC“,
<https://creativecommons.org/licenses/by-sa/3.0/legalcode>

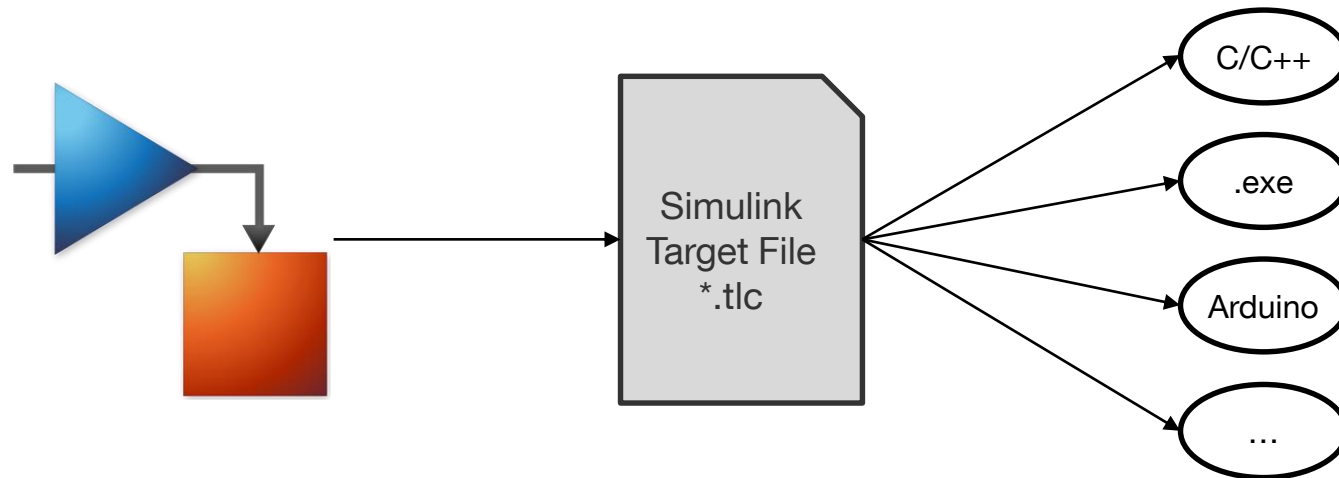
Simulink Code Generation & Deployment – Workflow



Simulink Code Generation & Deployment – Targets

Definition: A **target** is an environment for generating and building code for execution on a certain hardware or operating system platform.

- Target Language Compiler (TLC) files
 - Defines how the model is translated to code and executables
 - Specify the target environment
 - Support for built in / third party / custom targets

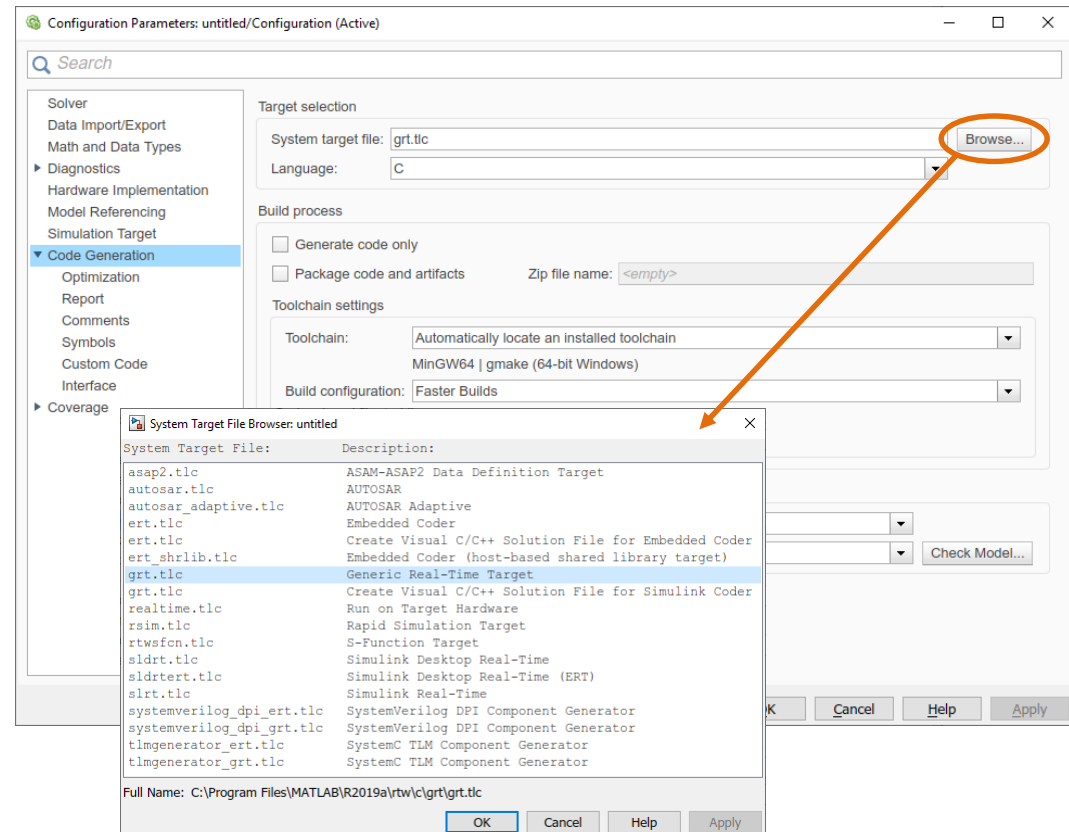


- The **application** determines the target file to be used!

Simulink Code Generation & Deployment – Target Files

The target file can be selected in the Configuration Parameters window.
Some examples...

- **Generic Real Time (grt.tlc)**
 - Simulate model as standalone on computer → confirm that generated code runs correct
 - Run Simulink model in external mode
 - Use grt target file as template for custom *.tlc files
- **Rapid Simulation Target (rsim.tlc)**
 - Run on remote computer
 - Distributed computing (e.g. Monte Carlo)
 - Stand-alone (pseudo) realtime simulation
 - Variable-step solver possible!




Usually, a fixed-step solver must be used for code generation.

Simulink Code Generation & Deployment – Target Files

- Embedded Real Time Target (**ert.tlc**)
 - Embedded code applications
 - Highly optimized code (RAM and ROM)
 - Runs on virtually any processor
- AUTomotive Open System ARchitecture (**autosar.tlc**)
 - Automotive applications
- Shared Library Target (**ert_shrlib.tlc**)
 - Generate shared library
 - Windows: Dynamic Link Library DLL
 - Unix: Shared Object SO
- A comparison of the supported system target files can be found in the Simulink documentation, refer to (requires login):
 - <https://de.mathworks.com/help/rtw/ug/compare-system-target-file-support.html>
 - <https://de.mathworks.com/help/rtw/ug/customizing-system-target-files.html>

Simulink Code Generation & Deployment – Code Generation / Build Settings

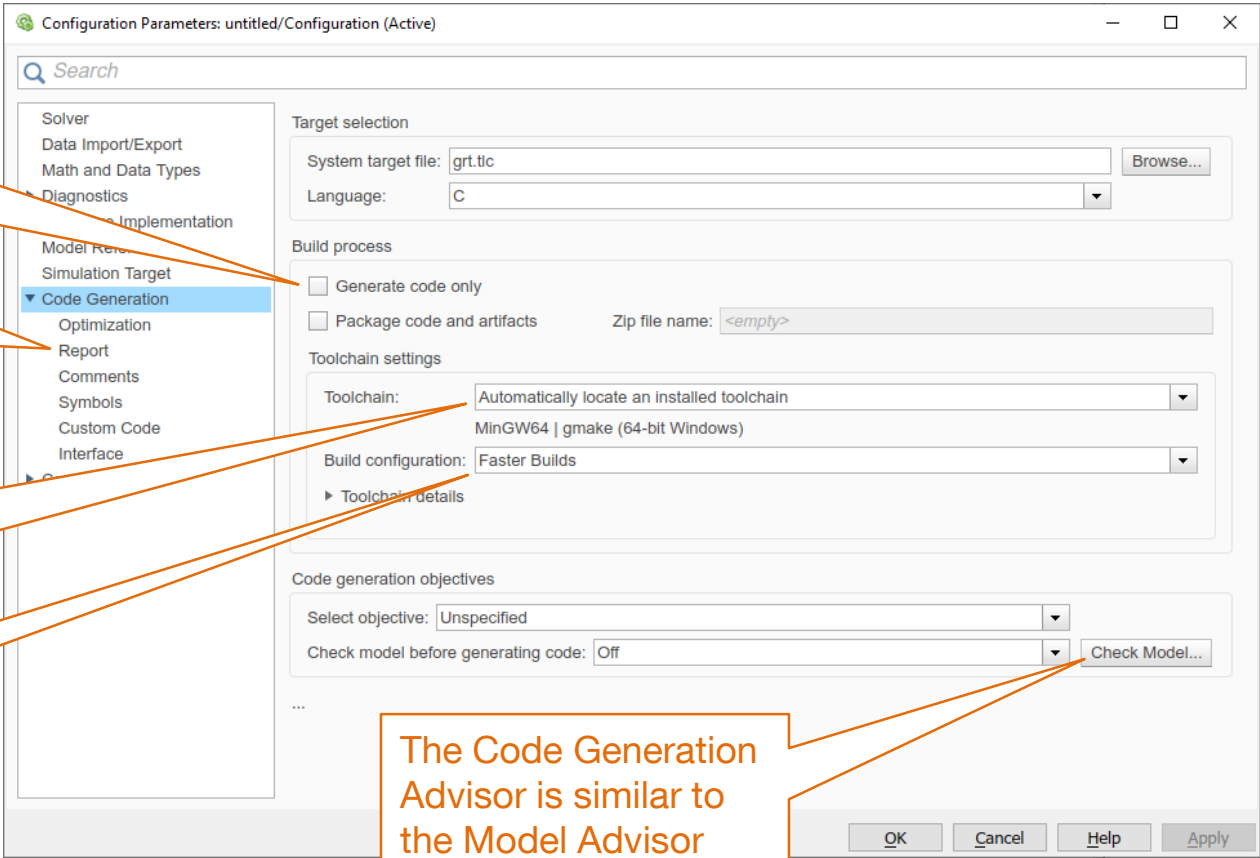
- Build generation can be triggered in the model window with this button: 

You can choose to generate code only (or also execute the make file / build) ...

... and generate a code generation report.

You can specify and validate the code generation toolchain, and ...

... you can prioritize build time, execution time, etc.



The Code Generation Advisor is similar to the Model Advisor (see next section).

Simulink Code Generation & Deployment – Code Generation / Build Settings

- Model configuration parameters: Available options depend on selected TLC file.

Configuration Parameters: untitled/Configuration (Active)

Search

Solver

Data Import/Export

Math and Data Types

▶ Diagnostics

Hardware Implementation

Model Referencing

Simulation Target

▼ Code Generation

Optimization

Report

Comments

Symbols

Custom Code

Interface

▶ Coverage

Default parameter behavior: Tunable

Configure...

☒ Use memcopy for vector assignment

Memcopy threshold (bytes): 64

Loop unrolling threshold: 5

Maximum stack size (bytes): Inherit from target

▼ Advanced parameters

☐ Inline invariant signals

☒ Signal storage reuse

☒ Enable local block outputs

☒ Reuse local block outputs

☒ Eliminate superfluous local variables (expression folding)

☒ Remove code from floating-point to integer conversions with saturation

☒ Use memset to initialize floats and doubles to 0.0

☐ Remove code from floating-point to integer conversions that wraps out-

☒ Buffer for reusable subsystems

Stateflow

☐ Use bitsets for storing state configuration

☐ Use bitsets for storing Boolean data

Base storage type for automatically created enumerations: Native Integer

Configuration Parameters: untitled/Configuration (Active)

Search

Solver

Data Import/Export

Math and Data Types

▶ Diagnostics

Hardware Implementation

Model Referencing

Simulation Target

▼ Code Generation

Optimization

Report

Comments

Symbols

Custom Code

Interface

▶ Coverage

Software environment

Code replacement library: None

Shared code placement: Auto

☒ Support non-finite numbers

Code interface

Code interface packaging: Nonreusable function

Data exchange interface

Array layout: Column-major

External functions compatibility for row-major code generation: error

Generate C API for:

☐ signals

☐ parameters

☐ states

☐ root-level I/O

☐ ASAP2 interface

☐ External mode

▼ Advanced parameters

Standard math library: C99 (ISO)

Maximum word length: 2048

Buffer size of dynamically-sized string (bytes): 256

☒ Generate full file banner

Configuration Parameters: untitled/Configuration (Active)

Search

Solver

Data Import/Export

Math and Data Types

▶ Diagnostics

Hardware Implementation

Model Referencing

Simulation Target

▼ Code Generation

Optimization

Report

Comments

Symbols

Overall control

☒ Include comments

Auto generated comments

☒ Simulink block comments

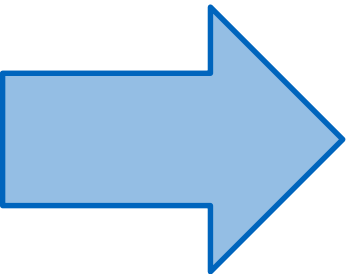
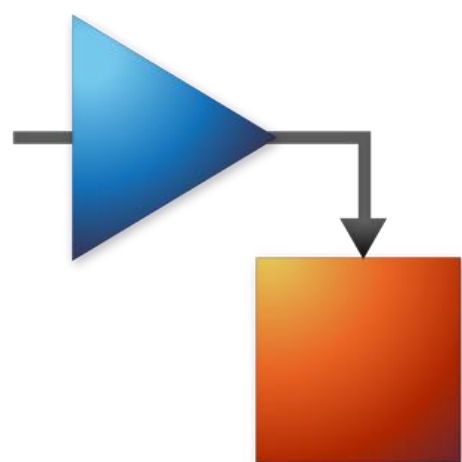
☐ Stateflow object comments

☐ MATLAB source code as comments

☐ Show eliminated blocks

☐ Verbose comments for 'Model default' storage class

5. Simulink Coder Model Architecture & Design



Jeremy Kratz
(https://commons.wikimedia.org/wiki/File:ISO_C++_Logo.svg),
„ISO C++ Logo“

SparkFun Electronics
from Boulder, USA
(https://commons.wikimedia.org/wiki/File:Arduino_Uno_-_R3.jpg),
„Arduino Uno - R3“,
<https://creativecommons.org/licenses/by/2.0/legalcode>



Altera Corporation
(https://commons.wikimedia.org/wiki/File:Altera_StratixIV_GX_FPGA.jpg),
„Altera StratixIVGX FPGA“,
<https://creativecommons.org/licenses/by/3.0/legalcode>



Monkhiker
(https://commons.wikimedia.org/wiki/File:B&R_Box_PC.jpg),
„B&R Box PC“,
<https://creativecommons.org/licenses/by-sa/3.0/legalcode>

Simulink Coder: Model Architecture & Design – Basic Modeling Considerations

- Products and blocks used: Do they support code generation?
- Signal naming: Which signals should carry a meaningful name?

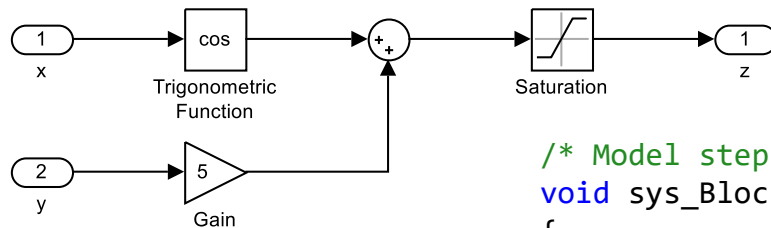
...so they can later be identified in the code!
→ see next slides
- Model componentization: Subsystems, libraries or model referencing?
 - **Subsystems** and **Libraries**: Code generation supported, with limitations.
For detected identical subsystems, the generated code includes only one copy of code for the multiple subsystems. Note: subsystems can be virtual or atomic (see next slides).
 - **Model referencing**: Well suited for code generation.
The generated code reflects the model structure.
- Subsystems: Virtual or atomic?

→ see next slides
- Parameterization: Virtual parameters or parameter objects?

→ see next slides

Simulink Coder: Model Architecture & Design – Basic Modeling Considerations: Signal Naming

- If possible, signals between blocks are automatically resolved during code generation.



Input and output names in code are taken from Simulink model.

Intermediate variable `u0` is created by the code generator.

```
/* Model step function */
void sys_Block_custom(real_T *arg_x, real_T *arg_y, real_T *arg_z)
{
    real_T u0;

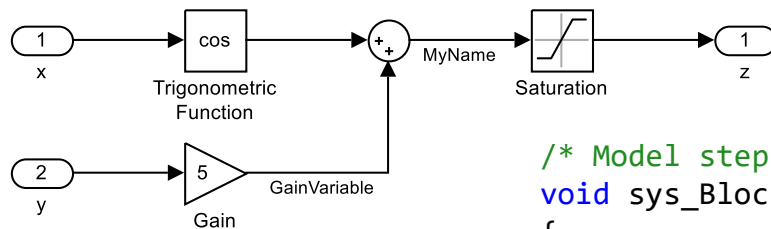
    u0 = 5.0 * *arg_y + cos(*arg_x);

    /* Saturate: '<Root>/Saturation' */
    if (u0 > sys_BlockAndSignal_P.Saturation_UpperSat) {
        /* Outport: '<Root>/z' */
        *arg_z = sys_BlockAndSignal_P.Saturation_UpperSat;
    } else if (u0 < sys_BlockAndSignal_P.Saturation_LowerSat) {
        /* Outport: '<Root>/z' */
        *arg_z = sys_BlockAndSignal_P.Saturation_LowerSat;
    } else {
        /* Outport: '<Root>/z' */
        *arg_z = u0;
    }

    /* End of Saturate: '<Root>/Saturation' */
}
```

Simulink Coder: Model Architecture & Design – Basic Modeling Considerations: Signal Naming

- Variable names can be enforced by specifying signal names.



- Double-click on signal, or
- Right click on signal → *Properties*

Intermediate variables
GainVariable and ...

... MyName are created
by the code
generator.

```
/* Model step function */
void sys_Block_custom(real_T *arg_x, real_T *arg_y, real_T *arg_z)
{
    /* Gain: '<Root>/Gain' incorporates:
     *   Inport: '<Root>/y'
     */
    GainVariable = 5.0 * *arg_y;

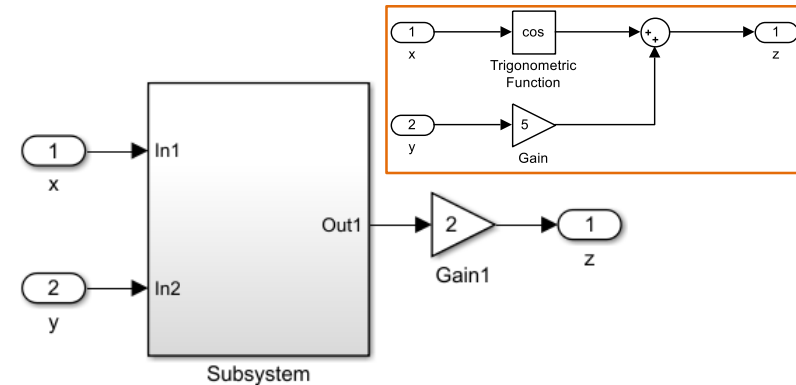
    /* Sum: '<Root>/Sum' incorporates:
     *   Inport: '<Root>/x'
     *   Trigonometry: '<Root>/Trigonometric Function'
     */
    MyName = cos(*arg_x) + GainVariable;

    /* Saturate: '<Root>/Saturation' */
    if (MyName > sys_BlockAndSignalName_P.Saturation_UpperSat) {
        /* Saturation block implementation not shown */
    }
    /* End of Saturate: '<Root>/Saturation' */
}
```

Simulink Coder: Model Architecture & Design – Basic Modeling Considerations: Virtual vs. Atomic Subsystems

- Virtual subsystems are automatically resolved for code generation.

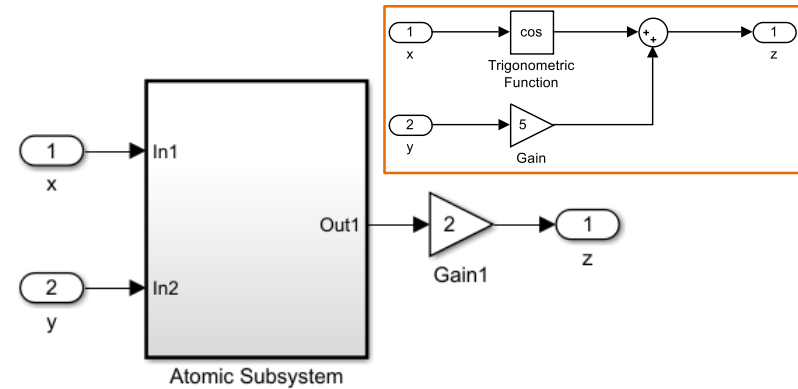
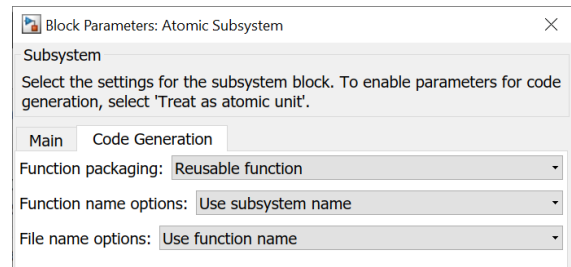
Note: Virtual subsystems or blocks merely change the graphical representation of the model!



```
/* Model step function */
void sys_Block_custom(real_T *arg_x, real_T *arg_y,
real_T *arg_z)
{
    /* Outport: '<Root>/z' incorporates:
    * Gain: '<Root>/Gain1'
    * Gain: '<S1>/Gain'
    * Inport: '<Root>/x'
    * Inport: '<Root>/y'
    * Sum: '<S1>/Sum'
    * Trigonometry: '<S1>/Trigonometric Function'
    */
    *arg_z = (5.0 * *arg_y + cos(*arg_x)) * 2.0;
}
```

Simulink Coder: Model Architecture & Design – Basic Modeling Considerations: Virtual vs. Atomic Subsystems

- By using atomic subsystems, you can enforce the creation of (reusable) functions.



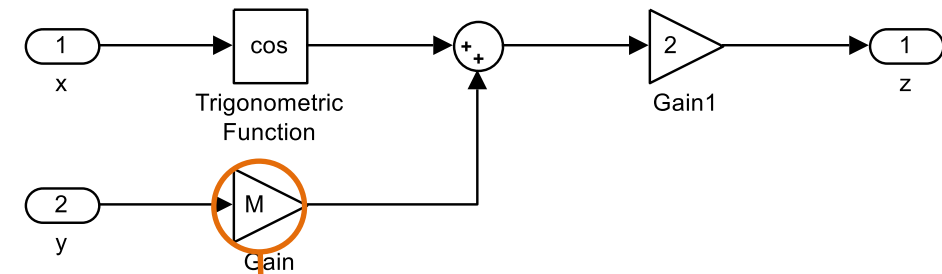
```
/* Output and update for atomic system: '<Root>/Atomic Subsystem' */
void sys_AtomicSubsy_AtomicSubsystem(real_T rtu_In1, real_T rtu_In2,
    B_AtomicSubsystem_sys_AtomicS_T *localB)
{
    localB->Sum = 5.0 * rtu_In2 + cos(rtu_In1);
}

/* Model step function */
void sys_Block_custom(real_T *arg_x, real_T *arg_y, real_T *arg_z)
{
    sys_AtomicSubsy_AtomicSubsystem(*arg_x, *arg_y,
        &sys_AtomicSubsystem_B.AtomicSubsystem);

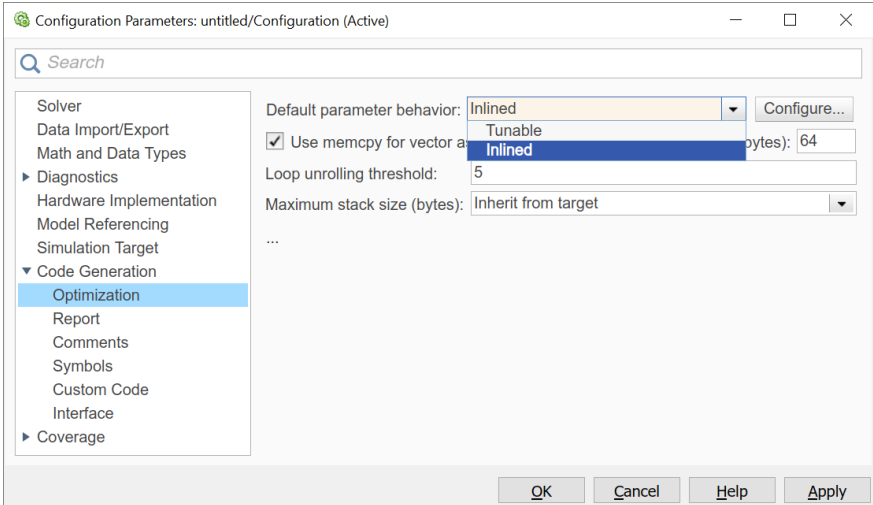
    *arg_z = 2.0 * sys_AtomicSubsystem_B.AtomicSubsystem.Sum;
}
```

Simulink Coder: Model Architecture & Design – Basic Modeling Considerations: Virtual Parameters vs. Parameter Objects

- Virtual parameters: Workspace values are hard-coded into the generated code



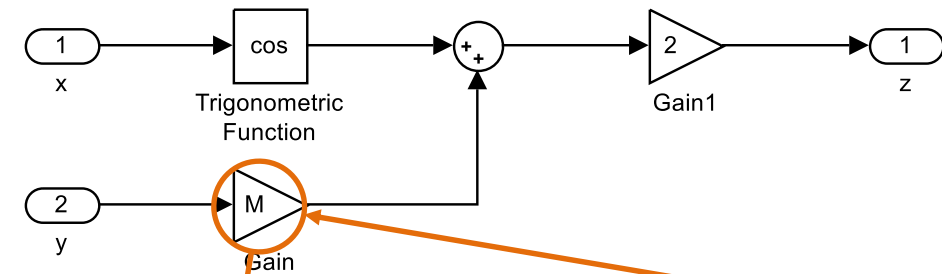
```
/* Model step function */
void sys_Block_custom(real_T *arg_x, real_T *arg_y,
real_T *arg_z)
{
    /* Output: '<Root>/z' incorporates:
    * Gain: '<Root>/Gain'
    * Gain: '<Root>/Gain1'
    * Inport: '<Root>/x'
    * Inport: '<Root>/y'
    * Sum: '<Root>/Sum'
    * Trigonometry: '<Root>/Trigonometric Function'
    */
    *arg_z = (5.0 * *arg_y + cos(*arg_x)) * 2.0;
}
```



In many cases (e.g. model referencing), setting the inline parameter option is required.

Simulink Coder: Model Architecture & Design – Basic Modeling Considerations: Virtual Parameters vs. Parameter Objects

- Parameter objects, which can be created in the Model Explorer, cause the parameter to be defined globally in the code.



The screenshot shows the 'Model Explorer' window. The 'Model Hierarchy' on the left shows the 'SimulinkModel' containing a 'Model Workspace'. The 'Contents of: Model Workspace (only)' pane on the right shows a table with one object, 'M', with a value of 5. The table has columns: Name, Value, DataType, Dimensions, Complexity, Min, Max, Unit. The 'M' row is highlighted with an orange circle, and an orange arrow points from this circle to the 'M' in the code block below.

Name	Value	DataType	Dimensions	Complexity	Min	Max	Unit
M	5	double (auto)	[1 1]	real			

```
/* Definition for custom storage class: Define */
#define M 5.0

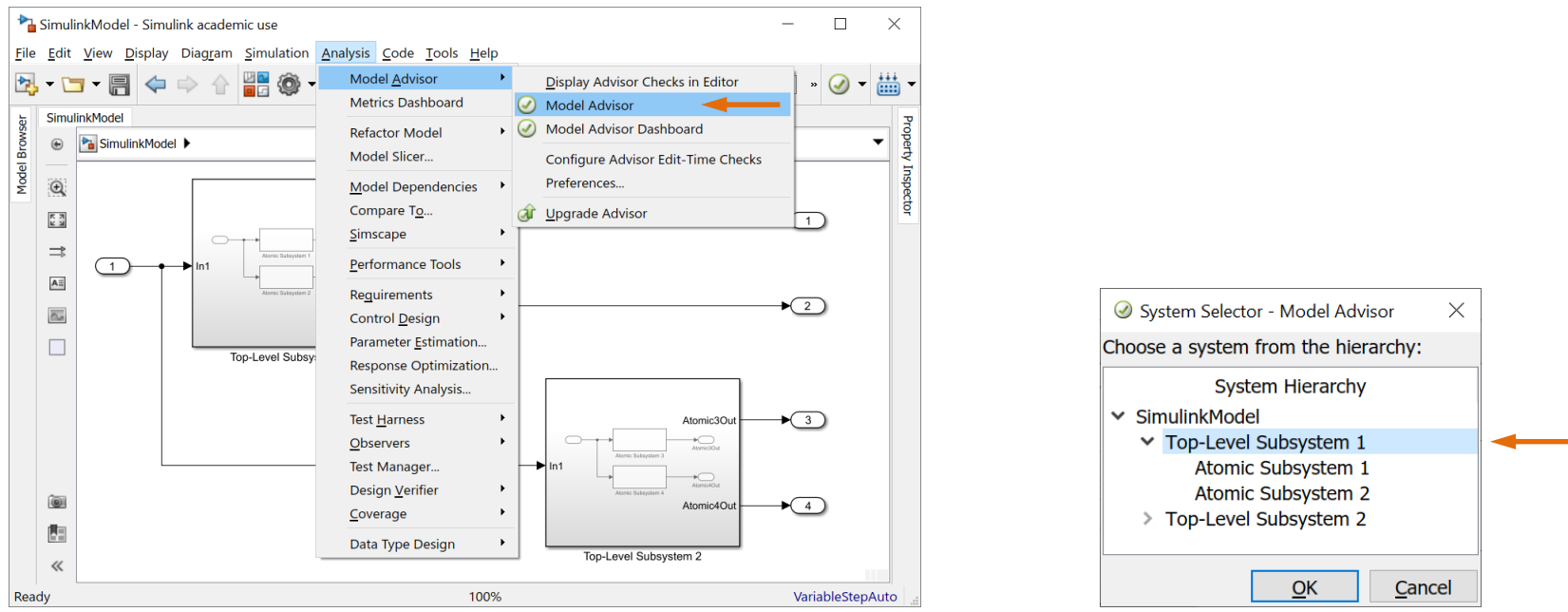
/* Model step function */
void sys_Block_custom(real_T *arg_x, real_T *arg_y, real_T
*arg_z)
{
    *arg_z = (M * *arg_y + cos(*arg_x)) * 2.0;
}
```

Simulink Coder: Model Architecture & Design – Some More Modeling Considerations...

- **Blocks**
 - Zero-based indexing
 - Evenly spaced breakpoints in lookup tables
 - Precalculated signals and parameters
 - Modeling global shared memory using data stores
 - Modeling local shared memory using data stores
- **Modeling Patterns**
 - Redundant Unit Delay and Memory blocks
 - Usage of For, While, and For Each subsystems with vector signals
 - Vector and bus signals crossing into atomic subsystems or Model blocks
 - Signal handling for multirate models
 - Data integrity and determinism in multitasking models
- **Configuration Parameters**
 - Prioritization of code generation objectives for code efficiency
 - Diagnostic settings for multirate and multitasking models

Simulink Coder: Model Architecture & Design – Model Advisor


- A useful tool in verifying that a model is ready for code generation is the Model Advisor.

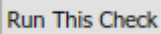


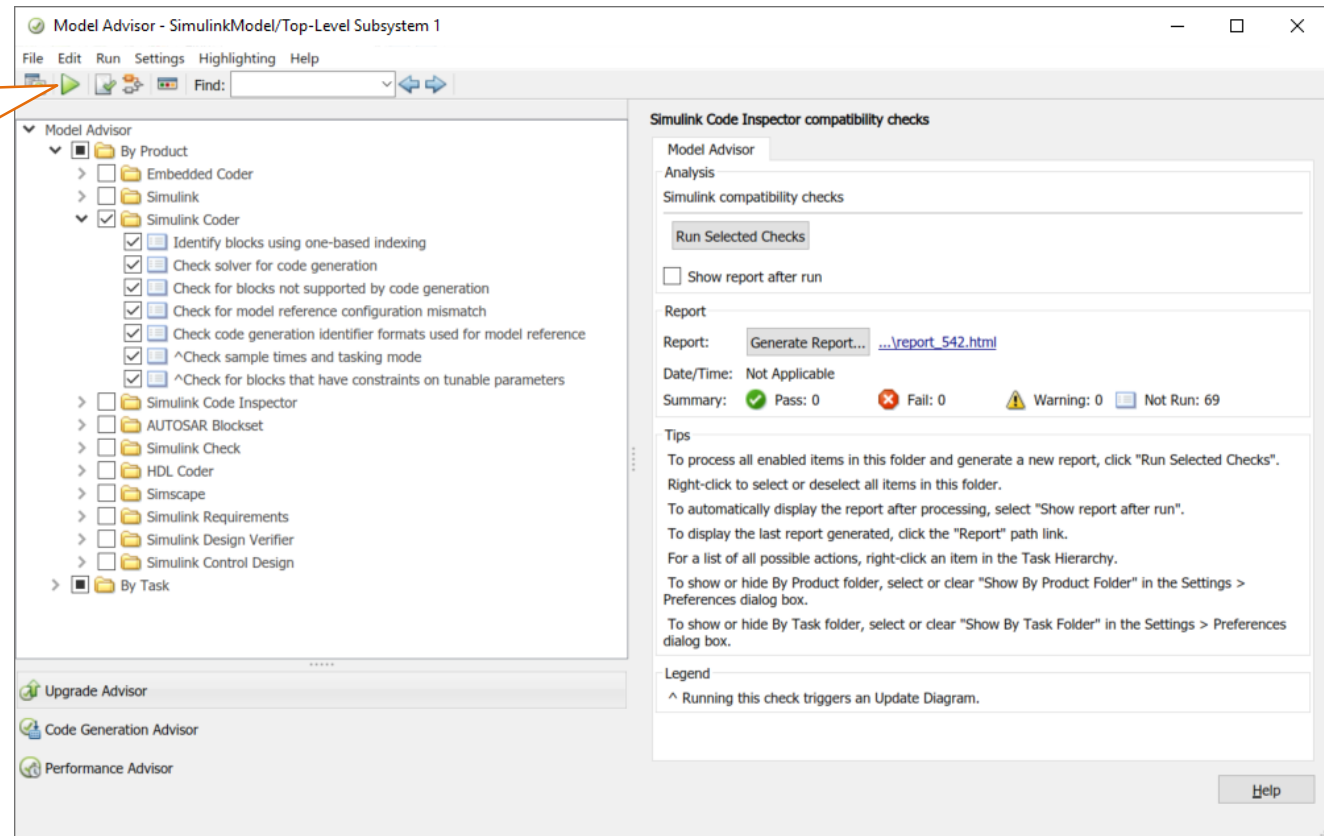
- Before the actual Model Advisor opens, you need to select the system you want to analyze from the system hierarchy

Simulink Coder: Model Architecture & Design – Model Advisor

- In the Model Advisor window, you can select the checks you want to run on the left pane.

You can then run the selected checks by clicking 

Alternatively, you can select one single check and click 



Simulink Coder: Model Architecture & Design – Model Advisor

- After running one or more checks, you can see the result on the right pane.
- Address the issues pointed out by the Model Advisor's relevant checks (most notably those in the category Simulink Coder) to prepare your model for code generation.

