Automatic code generation from MATLAB & Simulink

Stephan van Beek
Senior Application Engineer
The MathWorks
who uses MATLAB?
and what about Simulink?
The Digital System Design Challenge

- How can we rule out bad algorithms faster?
- How can we implement our algorithms faster?
- How can we ensure that our implementation matches our algorithm?
- How can we collaborate with our hardware and software engineers?
Introduction to Model-Based Design
Problems with Traditional Development

Requirements and Specs

- MS Word
- LaTeX
- Visio
- MS PowerPoint

Design with prototypes

- MATLAB Code
- C/C++ Code
- High-Level Languages

Manual Implementation

- VHDL
- Verilog
- C/C++

Test and Verification

- Simulation
- Test Bench
- Hardware Test Environment

Text-based
Prevents rapid iteration

Prototypes
Incomplete and expensive

Manual coding
Introduces human error

Traditional testing
Errors found too late in the process
Adopting Model-Based Design

Requirements and Specs

Design with Simulation

Automatic code generation

Test and Verification

Continuous Verification

Model Elaboration

Rapid design iterations

Executable models
- Unambiguous
- Only “one truth”

Design with Simulation
- Reduces “real” prototypes
- Systematic “what-if” analysis

Automatic code generation
- Minimizes coding errors

Test with Design
- Detects errors earlier
Model-Based Design Flow at its core

System Design and Simulation

MATLAB, Simulink, Stateflow
Algorithm and System Design

Generate

Digital Electronics
VHDL, Verilog
FPGA / ASIC

Generate

Embedded Software
C, C++
MCU / DSP

Implement

Integration
Introduction to MATLAB & Simulink
MATLAB at a glance

The leading environment for technical computing

- Interactive development environment
- Technical computing language
- Data analysis and visualization
- Algorithm development
Simulink at a glance

*The leading environment for system-level modeling and simulation*

- Block-diagram environment
- Model, simulate and analyze dynamic systems
- Build verification test-benches
- Fully integrated with MATLAB

Discrete filter in Simulink

Discrete filter in C
Why Simulink?

- Model, simulate and analyze

- Dynamic systems
  - Multi-domain
  - Multiple Rates
  - Discrete and continuous
  - Non-linear

- Architecture exploration
  - Rapidly model what-if scenarios
  - Share a design idea
  - Concept to implementation
Why using MATLAB and Simulink?

- To model and simulate complicated systems for design and verification
  - Communications
  - Video and image
  - Audio
  - Embedded
  - Mixed-signal
- Algorithm development
  - Filter design
- For data analysis and visualization
  E.g. data processing including
  - Statistics
  - Curve fitting
Edge Detection Demo
Challenge:

Edge Detection

Bill: System Engineer
Wikipedia: Sobel Algorithm for Edge Detection

Simplified description

In simple terms, the operator calculates the gradient of the image intensity at each point, giving the direction of the largest possible increase from light to dark and the rate of change in that direction. The result therefore shows how "abruptly" or "smoothly" the image changes at that point, and therefore how likely it is that that part of the image represents an edge, as well as how that edge is likely to be oriented. In practice, the magnitude (likelihood of an edge) calculation is more reliable and easier to interpret than the direction calculation.

Mathematically, the gradient of a two-variable function (here the image intensity function) is at each image point a 2D vector with the components given by the derivatives in the horizontal and vertical directions. At each image point, the gradient vector points in the direction of largest possible intensity increase, and the length of the gradient vector corresponds to the rate of change in that direction. This implies that the result of the Sobel operator at an image point which is in a region of constant image intensity is a zero vector and at a point on an edge is a vector which points across the edge, from darker to brighter values.

Formulation

Mathematically, the operator uses two $3\times3$ kernels which are convolved with the original image to calculate approximations of the derivatives - one for horizontal changes, and one for vertical. If we define $A$ as the source image, and $G_x$ and $G_y$ are two images which at each point contain the horizontal and vertical derivative approximations, the computations are as follows:

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * A \quad \text{and} \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

where $*$ denotes the 2-dimensional convolution operation.

The x-coordinate is here defined as increasing in the "right"-direction, and the y-coordinate is defined as increasing in the "down"-direction. At each point in the image, the resulting gradient approximations can be combined to give the gradient magnitude, using:

$$G = \sqrt{G_x^2 + G_y^2}$$

Using this information, we can also calculate the gradient's direction:

$$\Theta = \arctan \left( \frac{G_y}{G_x} \right)$$

where, for example, $\Theta$ is 0 for a vertical edge which is darker on the left side.

More formally

Since the intensity function of a digital image is only known at discrete points, derivatives of this function cannot be defined unless we assume that there is an underlying continuous intensity function which has been sampled at the image points. With some additional assumptions, the derivative of the continuous intensity function can be approximated as a function on the sampled intensity function, i.e. the digital image. It turns out that the derivatives at any particular point are functions of the intensity values at virtually all image points. However, approximations of these derivative functions can be defined at lesser or larger degrees of accuracy.

The Sobel operator represents a rather inaccurate approximation of the image gradient, but is still of sufficient quality to be of practical use in many applications. More precisely, it uses intensity values only in a $3\times3$ region around each image point to approximate the corresponding image gradient, and it uses only integer values for the coefficients which weight the image
Challenge cont’d:

Bill: System Engineer

John: FPGA Engineer

1%?

1 week?
Summary: Sobel Edge Detection Demo

Requirements and Specs

Design with Simulation

Automatic code generation

Test and Verification

Executable models
- Unambiguous
- Only “one truth”

Design with Simulation
- Reduces “real” prototypes
- Systematic “what-if” analysis

Automatic code generation
- Minimizes coding errors

Test with Design
- Detects errors earlier
Automatic Code Generation Tools from The MathWorks
Top-Down Design Flow - Partitioning

System Design and Simulation

Implement

Concept Design

Integration

FPGA / ASIC

Digital Electronics

Generate

Embedded Software

MCU / DSP
Hardware Implementation Flow

MATLAB, Simulink, Stateflow
Algorithm and System Design

Simulink HDL Coder

Filter Design HDL Coder

Generic RTL
VHDL, Verilog

EDA Simulator
Link

Legacy / External Code

HDL Simulation

Synthesis
P & R

FPGA / ASIC

Xilinx ISE
Synplify Pro
DesignCompiler
PrecisionRTL
Leonardo

Altera Quartus II

Mentor Graphics ModelSim
Cadence Incisive
Synopsys Discovery
Targeting Embedded Microprocessors and DSPs

MATLAB, Simulink, Stateflow
Algorithm and System Design

Real Time Workshop
Embedded Coder

C, C++

Embedded IDE
Link

Target Support
Package

Infineon C166
STMicroelectronics ST10
Freescale MPC5xx
TI C2000
TI C6000

Green Hills MULTI
Analog Devices Visual DSP++
TI Code Composer Studio
Altium TASKING

Specific Peripheral S/W Drivers
Automatic Code Generation Summary

- **Converge** on the best algorithm faster
- **Implement** your algorithm faster
- **Ensure** that your implementation matches your system model
- **Collaborate** with your hardware / software engineers

MATLAB, Simulink, Stateflow
Algorithm and System Design

Automatic code generation

Co-simulation

Generate

Verify

HDL

C / C++

FPGA / ASIC

MCU / DSP