2009-01-0269

Fixed-Point ECU Code Optimization and Verification with Model-Based Design

Tom Erkkinen
The MathWorks

ABSTRACT

When implementing production software for fixed-point engine control units (ECUs) it is important to consider the code optimization and code verification strategies for the embedded algorithms. System and software engineers work together to design algorithms that satisfy the system performance requirements without significant numerical quantization results. Software engineers and suppliers in mass production environments then implement the design on an embedded system with limited memory and execution speed resources. The primary goals after design are to generate optimized code and verify that the implementation matches the original model's functional behavior.

Model-Based Design simplifies fixed-point development by providing tools and workflows that support the complete design, implementation, and verification processes. System engineers performing on-target rapid prototyping for fixed-point ECUs benefit from automated scaling workflows that provide an initial fixed-point design. Production software engineers benefit from automated scaling as well, but they then require fine-grain control over fixed-point data specification within their modeling environment for items such as accumulator word size. Eventually a detailed software design is produced.

Automatic code generation is then invoked with options that maximize code efficiency for fixed-point processors. These options include portable ANSI/ISO C optimizations, plus target-specific optimizations. Automated checking tools and workflow advisors help ensure the appropriate optimization settings are enabled. Capabilities exist for fixed-point verification and validation, including bit-accurate fixed-point simulation and automated processor-in-the-loop testing. The latter is particularly useful when using target-optimized code, because the code cannot be simulated on the host and can only be tested on the actual embedded target.

This paper presents Model-Based Design capabilities and tools that support verification of optimized fixed-point ECU software used in mass production vehicles.

INTRODUCTION

With Model-Based Design, code is generated from models and then verified using software-in-the-loop (SIL), processor-in-the-loop (PIL), and hardware-in-the-loop (HIL) testing. These techniques support incremental verification and test reuse. The initial test cases are developed and tested using the algorithm and plant (or environment) model within the simulation environment.

For SIL, source code is generated from the algorithm and compiled using the host compiler. The tests developed earlier for the model are reused and executed with the host-compiled algorithm code. The results are compared to the original model behavior and analysis is performed to ensure an accurate match.

For PIL, source code is generated from the algorithm and cross-compiled on the host for deployment on an embedded microprocessor target. The tests developed earlier are again reused and executed with the target compiled algorithm code running on embedded target hardware or an instruction set simulator provided by the cross-compiler. The results are compared to the original model behavior and analysis is performed to ensure an accurate match.

For HIL, source code is generated from the plant or environment model and the code is compiled for deployment on a real-time simulator. The real-time

The Engineering Meetings Board has approved this paper for publication. It has successfully completed SAE's peer review process under the supervision of the session organizer. This process requires a minimum of three (3) reviews by industry experts.

ISSN 0148-7191

SAE Customer Service: Tel: 877-606-7323 (inside USA and Canada)
Tel: 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: CustomerService@sae.org
SAE Web Address: http://www.sae.org

Printed in USA

simulator then communicates directly with the embedded ECU containing the microprocessor used during a PIL test. The tests developed earlier are reused and executed, but this time using hard real-time execution. The results are compared to the original model behavior and analysis is performed to ensure an accurate match.

This verification approach for Model-Based Design is well established, as is the ability to generate optimized ANSI/ISO C code using model patterns and style guidelines [1]. PIL solutions based on specified cross-compiler or IDE tool chains are also well established [2].

What is more recent and described herein are technologies that facilitate generation of target-optimized code and enable PIL testing for any general-purpose embedded microprocessor target environment. This technology is available with the Real-Time Workshop Embedded Coder product from The MathWorks [3].

GENERATING TARGET-OPTIMIZED CODE USING TARGET FUNCTION LIBRARIES

One method to optimize source code for specific targets is to incorporate existing (or legacy) target-optimized code using the Legacy Code Tool. For this, engineers specify the legacy function's call signature including its interfaces, using a MATLAB API. When executed, a Simulink S-Function is created which allows the legacy code to be simulated and called appropriately by the generated code. Another way to generate optimized source code involves target function libraries.

INTRODUCTION TO TARGET FUNCTION LIBRARIES - Target function libraries (TFLs) are MATLAB APIs that allow engineers to create and register tables of code segments that replace the default code segments generated by Real-Time Workshop Embedded Coder.

TFLs currently support a variety of math functions such as `sin, cos, pow, sqrt`. and operators including:

- + (addition)
- − (subtraction)
- * (multiplication)
- / (division)

Advanced TFLs also exist for additional code replacements such as memory copy (`memcpy`).

Once a TFL table is created, it must be registered so that Simulink can incorporate the custom TFL table with default tables that are provided within the code generation options panel. The creation of a TFL and its registration use the TFL API and created MATLAB files.

Once created and registered, TFL tables can be selected for a model based on its embedded target configuration. If the target configuration changes, the model developer simply needs to select a different TFL. This is a major benefit over S-Function based approaches to target optimization, since S-Functions are blocks that are placed inside of models, so switching targets requires switching blocks, which can be a tedious process. It is much easier to leave the blocks as is within the model, and instead substitute a different target configuration using a *build* script or simple user interface option toggle prior to the code generation process.

When a TFL is selected for a model, the individual entries that comprise the TFL tables can be displayed using a viewer or with a library tool tip. Within each table, if multiple matches are found for a TFL entry object, the priority level determines the match that is returned. A higher-priority (lower-numbered) entry is used over a similar entry with a lower priority (higher number).

USING TARGET FUNCTION LIBRARIES - The general steps for creating and using a target function library are as follows:

1. Create a TFL replacement table using the TFL API.
2. Register the TFL table using a registration file.
3. Confirm the TFL implementation using a viewer.
4. Select the TFL using the code generation interface panel.
5. Generate the code and observe the replacements.

An example TFL that illustrates the general steps follows. It replaces the default division operator with a more robust version that prevents division by zero. Instead of the division operator '/', a function will be called that examines the denominator and returns a nominal value or default value based on the floating-point data type size. Thus, two robust division functions are required to be generated based on the data types used in the model: one function for doubles and another for singles, `_rdbl_div()` and _rsgl_div(), respectively. The example is written in ANSI-C and can execute on a host or target environment.

1. Create a TFL Replacement Table

A TFL table entry definition requires the implementation function or operator name (e.g., `_rdbl_div`) as well as the source and header files in which the function is defined and declared. The TFL also requires additional information, such as the priority in case of conflicts from an entry from 0 to 100, where 0 has highest priority. The number of and types of arguments are also specified. Figure 1 shows the entire TFL definition for the double data type division replacement operators. A similar TFL is defined for the single-precision operator replacements, but is not shown.

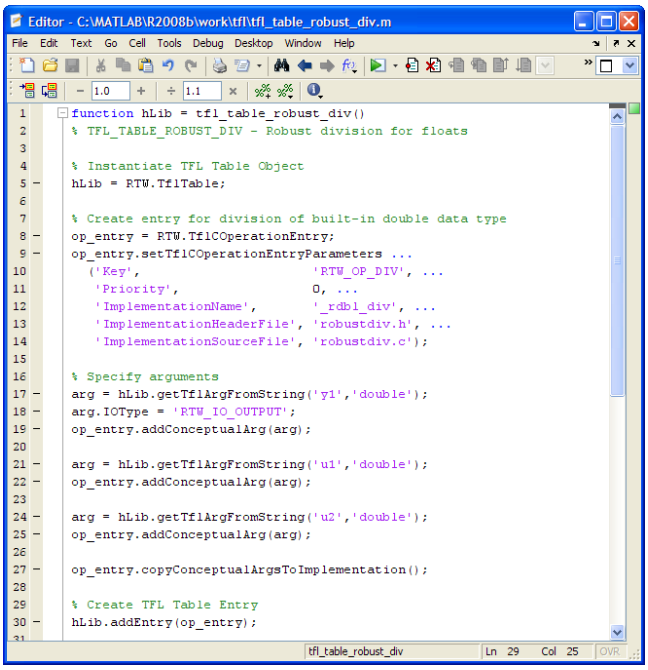

Figure 1. TFL definition file (*tfl_table_robust_div.m*).

2. Register the TFL Table

To use a TFL, it must be registered within the modeling environment. Registration information includes the name of the TFL displayed to the model developer and the base TFL that the replacement TFL is derived from. The registration information is written using a MATLAB based API and is stored in a file named *sl_customization.m*. The registration file(s) need to be on the MATLAB path or local working directory. See Figure 2 for the example registration.

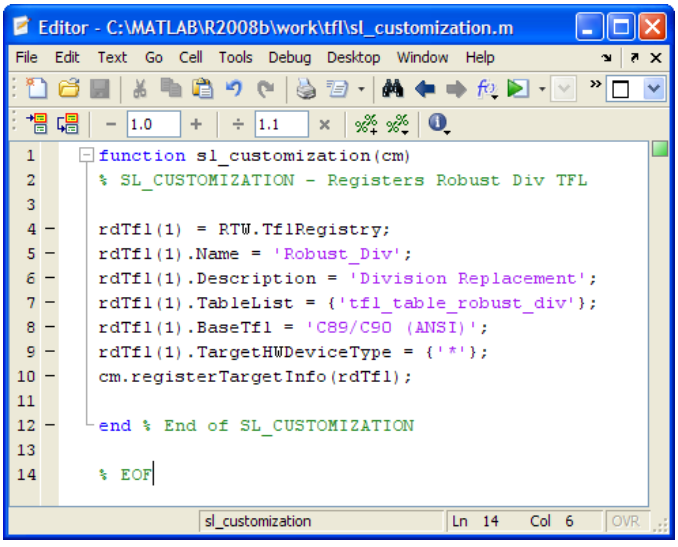

Figure 2. TFL registration file (*sl_customization.m*).

3. Confirm the TFL Implementation

Engineers can view their custom TFL implementations along with TFLs provided by Real-Time Workshop Embedded Coder using a viewer. To invoke the viewer, type the view command at the MATLAB prompt, `RTW.ViewTfl`. A window similar to that shown in Figure 3 is displayed.

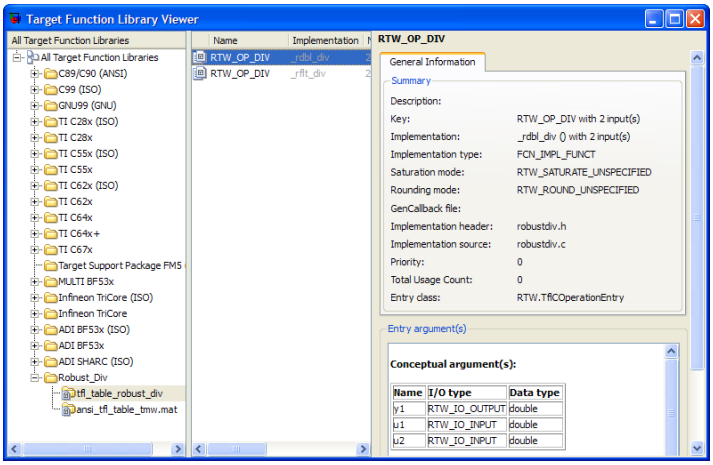

Figure 3. TFL Viewer.

4. Select the TFL

After a TFL has been developed and registered, it can be used by any model via a pull-down menu within the code generation configuration panel or via a build script. The developer simply needs to select the appropriate TFL and generate code.

For the example, the simple model shown in Figure 4 contains division operations in the form of Simulink blocks, Stateflow charts, and Embedded MATLAB functions, all of which are supported by TFLs. Note that you can also use the TFL for code generated directly from MATLAB using the `emlc` command. The subsystem and function use single-precision floats, while the chart uses double-precision.

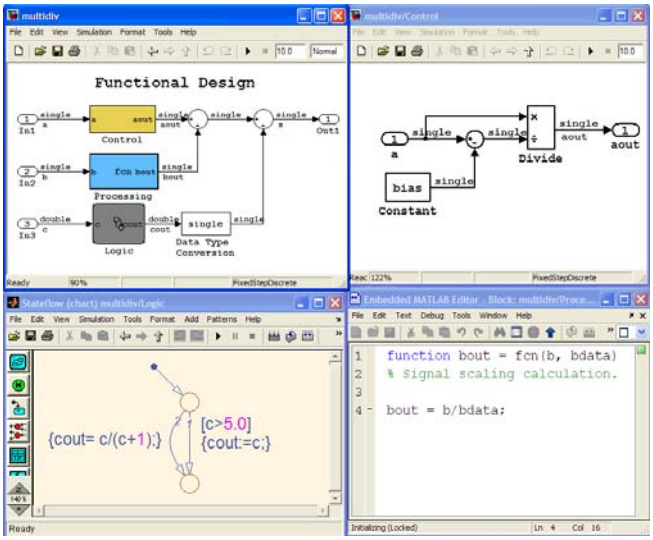

Figure 4. Example division model (clockwise from top left): top-level model, Simulink subsystem, Embedded MATLAB function, and Stateflow chart.

Real-Time Workshop Embedded Coder generates production code with default ANSI-C TFL. Code is then regenerated using the custom TFL for robust divisions. See Figures 5 and 6 for a comparison of the resulting source code.

```
void multidiv_step(void)
{
  {
    real_T rtb_cout;

    /* Embedded MATLAB: '<Root>/Processing' */
    /* Embedded MATLAB Function 'Processing': '<S3>:1' */
    /*  Signal scaling calculation.  */
    /* '<S3>:1:4' */

    /* Stateflow: '<Root>/Logic' incorporates:
     *  Inport: '<Root>/In3'
     */
    /* Gateway: Logic */
    /* During: Logic */
    /* Transition: '<S2>:1' */
    if (multidiv_U.c > 5.0) {
      /* Transition: '<S2>:2' */
      rtb_cout = multidiv_U.c;
    } else {
      /* Transition: '<S2>:3' */
      rtb_cout = multidiv_U.c / (multidiv_U.c + 1.0);
    }

    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<S1>/Constant'
     *  DataTypeConversion: '<Root>/Data Type  Conversion'
     *  Inport: '<Root>/In1'
     *  Inport: '<Root>/In2'
     *  Product: '<S1>/Divide'
     *  Sum: '<Root>/Sum'
     *  Sum: '<Root>/Sum2'
     *  Sum: '<S1>/Sum'
     */
    multidiv_Y.Out1 = (multidiv_U.a / (multidiv_U.a - 4.0F) + multidiv_U.b /
                       0.5F) + (real32_T)rtb_cout;
  }
}
```

Figure 5. Code generated from example model using ANSI-C (default).

```
void multidiv_step(void)
{
  {
    real_T rtb_cout;

    /* Embedded MATLAB: '<Root>/Processing' */
    /* Embedded MATLAB Function 'Processing': '<S3>:1' */
    /*  Signal scaling calculation.  */
    /* '<S3>:1:4' */

    /* Stateflow: '<Root>/Logic' incorporates:
     *  Inport: '<Root>/In3'
     */
    /* Gateway: Logic */
    /* During: Logic */
    /* Transition: '<S2>:1' */
    if (multidiv_U.c > 5.0) {
      /* Transition: '<S2>:2' */
      rtb_cout = multidiv_U.c;
    } else {
      /* Transition: '<S2>:3' */
      rtb_cout = _rdbl_div(multidiv_U.c, multidiv_U.c + 1.0);
    }

    /* Outport: '<Root>/Out1' incorporates:
     *  Constant: '<S1>/Constant'
     *  DataTypeConversion: '<Root>/Data Type  Conversion'
     *  Inport: '<Root>/In1'
     *  Inport: '<Root>/In2'
     *  Product: '<S1>/Divide'
     *  Sum: '<Root>/Sum'
     *  Sum: '<Root>/Sum2'
     *  Sum: '<S1>/Sum'
     */
    multidiv_Y.Out1 = (_rflt_div(multidiv_U.a, multidiv_U.a - 4.0F) + _rflt_div
                       (multidiv_U.b, 0.5F)) + (real32_T)rtb_cout;
  }
}
```

Figure 6. Code generated from example model using custom TFL (robust division).

There are many uses for TFLs beyond implementing robustness code. Production applications often use special hardware instructions, or pragmas, to provide optimized fixed-point math with overflow and underflow protection, as well as fast signal processing routines such as FFTs. Some organizations also use their own custom functions for fixed-point operations in the generated code.

If the replacements are based on ANSI-C code and are host compliable, it is easy to verify the functional behavior using SIL. If the code uses target hardware or cross-compiled features, then it can only be tested on the target using techniques including PIL and HIL. An approach for developing a custom PIL environment using recently published APIs follows.

VERIFYING TARGET-OPTIMIZED CODE USING PIL APIs

During PIL simulation, Simulink simulates the non-PIL part of the model, such as the plant model, for one sample interval and sends output signals to the target platform. When the target receives the signals, it executes the PIL algorithm for one sample step and returns its output signals back to Simulink. At this point, one sample cycle of the simulation is complete and the model proceeds to the next sample interval. The process repeats and simulation progresses. At each sample period, the model test harness and object code exchange all I/O data. PIL simulations do not run in real time.

Two techniques are available for PIL: using PIL blocks and using PIL simulation mode. The use of PIL blocks is similar to other legacy code or communication approaches. An S-Function block is created during the code generation process that provides an interface to the target compiled algorithm code. Developers then place the PIL block in the model, either as a replacement for the existing algorithm subsystem, or, if the model is not too large, in parallel with it.

As with TFLs, some developers prefer to not add blocks or constructs to their model to do PIL simulation and want a more seamless approach. This is now possible with the PIL simulation mode and accompanying APIs released with R2008b by The MathWorks in October 2008.

INTRODUCTION TO PIL MODE AND PIL APIS - A Model block can be configured to run in normal (interpreted) simulation, accelerated simulation using the generated and host-compiled code running, or PIL mode. In PIL mode, the referenced model is a conduit that gives the model access to the generated and cross-compiled code running in the target environment. When a Model block is in PIL mode, the label (PIL) appears on the block.

USING PIL SIMULATION MODE AND PIL APIs - With Model block PIL mode, a Processor-in-the-Loop (PIL) Connectivity API is used to establish communication

with the target processor. With so many different target environments, there are numerous, ever- changing tools and approaches for building, downloading, and communicating with an executable. So having an API for custom integration, as opposed to point solutions or products sold and maintained by vendors, is often preferred by production organizations.

Once the PIL communication is established and PIL simulation is started, an automated procedure begins:

1. Build the PIL application (i.e., target executable).
2. Download the executable to the target.
3. Run the executable.
4. Start communication between the model and executable.

The above functionalities are enabled using the PIL Target Connectivity API components shown in Figure 7.

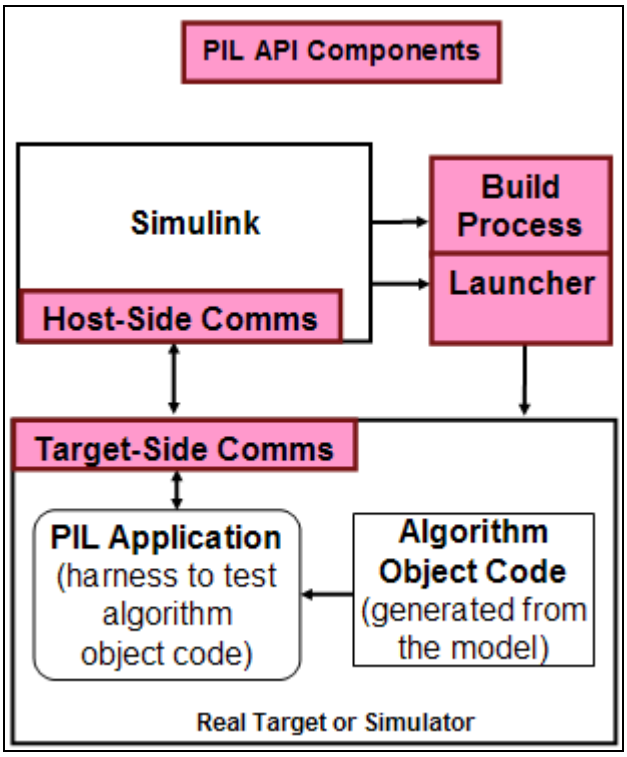

Figure 7. PIL target connectivity API components.

The communications part of the target connectivity API builds upon the `rtiostream` API, which implements a communication channel to exchange data between different processes such as a host-target communications channel. The communications channel comprises separate driver code running on the host and target. The `rtiostream` API defines the signature of both target-side and host-side functions that must be implemented by this driver code.

The API is independent of the physical layer used to send the data. Possible physical layers include RS232, Ethernet, or Controller Area Network (CAN). Developing (or acquiring) host and target drivers are the primary tasks to be done before beginning a PIL implementation using the PIL APIs.

Once the basic communication interfaces are written, the general steps to implement a PIL environment are:

1. Create the basic PIL connectivity framework using `rtw.connectivity.Config`
2. Create a mechanism to configure the build process using `rtw.connectivity.MakefileBuilder`
3. Create a mechanism to download and execute the application using `rtw.connectivity.Launcher`
4. Establish a communications implementation using `rtiostream` for:
   - Target-side driver code integration using `rtw.pil.RtIOStreamApplicationFramework`
   - Host-side driver code integration using `rtw.connectivity.RtIOStreamHostCommunicator`
5. Register the PIL configuration for use with Simulink using the `rtw.connectivity.ConfigRegistry` and an `sl_customization.m` file

A processor-in-the-loop example is provided below. The example is based on a TCP/IP implementation and runs the target as a separate process on the host computer. Figure 8 shows a segment of the PIL implementation for the example.

```
% Create an instance of MakefileBuilder; this works in
% conjunction with your template makefile to build the PIL
% executable
builder = rtw.connectivity.MakefileBuilder(componentArgs, ...
    targetApplicationFramework, ...
    exeExtension);

% Launcher
launcher = mypil.Launcher(componentArgs, builder);

% File extension for shared libraries (e.g. .dll on Windows)
sharedLibExt=system_dependent('GetSharedLibExt');

rtiostreamLib = [prefix 'rtiostreamtcpip' sharedLibExt];

communicator = rtw.connectivity.RtIOStreamHostCommunicator(componentArgs, ...
    launcher, ...
    rtiostreamLib);
```

Figure 8. PIL API example.

With the PIL connectivity components established, it is straightforward to run a PIL test. The example in Figure 9 has two model blocks that reference the same model, a simple counter. The top block is set to PIL simulation mode, which uses the TCP/IP communication described above. The bottom model block is set to normal mode and simulates the counter using Simulink. The plots on the right show the counter output results for PIL mode (top), Normal mode (bottom), and the difference or error between the two results (middle).

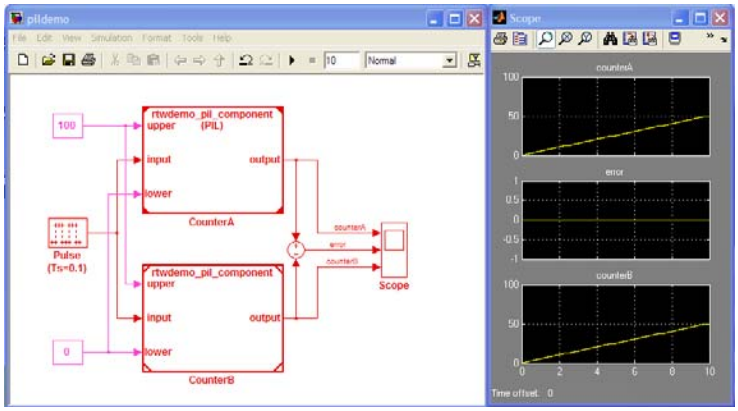

Figure 9. PIL model and execution results.

In this simple example, there was no error. In more sophisticated models, differences might arise from a variety of sources, including floating-point numerical approximation differences between host and target compilers and hardware.

CONCLUSION

This paper introduces recently developed technologies that further enable organizations to adopt Model-Based Design for embedded system deployment and verification. The first technology involves generation of target-optimized code using target function libraries. These APIs make it easy for production organizations to substitute their processor-specific code for default ANSI/ISO C generated by Real-Time Workshop Embedded Coder.

The second technology involves Processor-in-the-Loop (PIL) APIs that let engineers develop an automated test environment for running a model in cosimulation with the actual generated and cross-compiled production code on the target microprocessor or instruction set simulator.

Using both technologies together allows engineers to generate and verify highly optimized target code in a way that is generic and customizable. This flexibility is important, since every organization has unique hardware and software needs that point solutions and add-on products from vendors cannot always address.

REFERENCES

1. B. Chou, S. Mahapatra, “Techniques for Generating and Measuring Production Code Constructs from Controller Models,” SAE Paper Offer 09AE-0021.
2. T. Erkkinen, S. Breiner, “Automatic Code Generation – Technology Adoption Lessons Learned from Commercial Vehicle Case Studies,” SAE Paper 2007-01-4249.
3. MathWorks Release 2008b, The MathWorks, Inc. www.mathworks.com