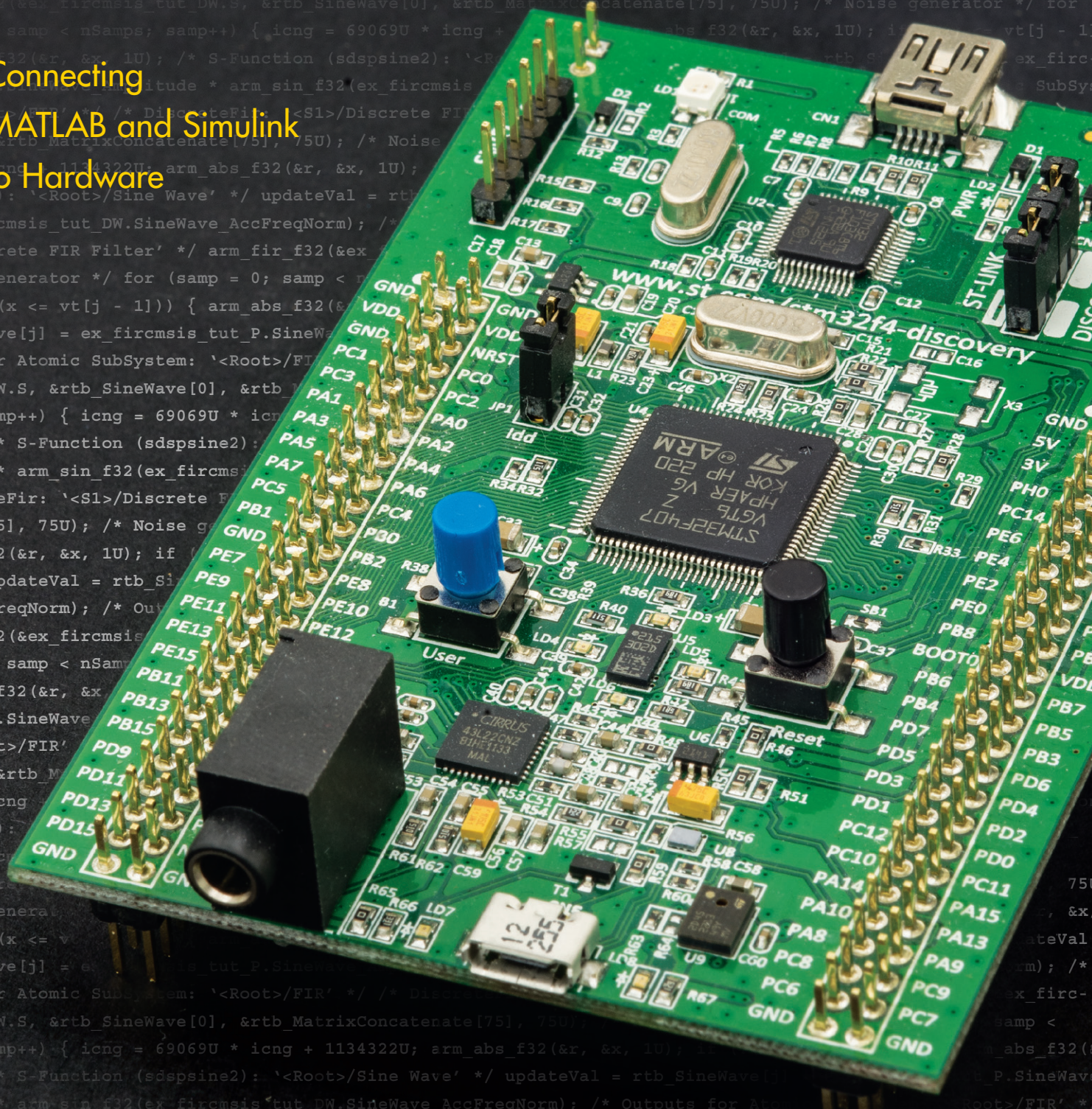
MathWorks News&Notes

The Magazine for the MATLAB® and Simulink® Community

Connecting MATLAB and Simulink to Hardware



ALSO IN THIS ISSUE

MathWorks®

Join the MATLAB and Simulink Conversation

File Exchange Contribute and download functions, examples, apps, and other files.

MATLAB Answers Ask and answer questions about programming.

Cody™ Play the game that expands your knowledge of MATLAB® with over 1000 programming problems.

Trendy™ Create and track web data trends using a hosted MATLAB service.

Blogs Get inspired by MATLAB and Simulink® thought leaders—including Cleve Moler.

mathworks.com/matlabcentral

MATLAB® CENTRAL

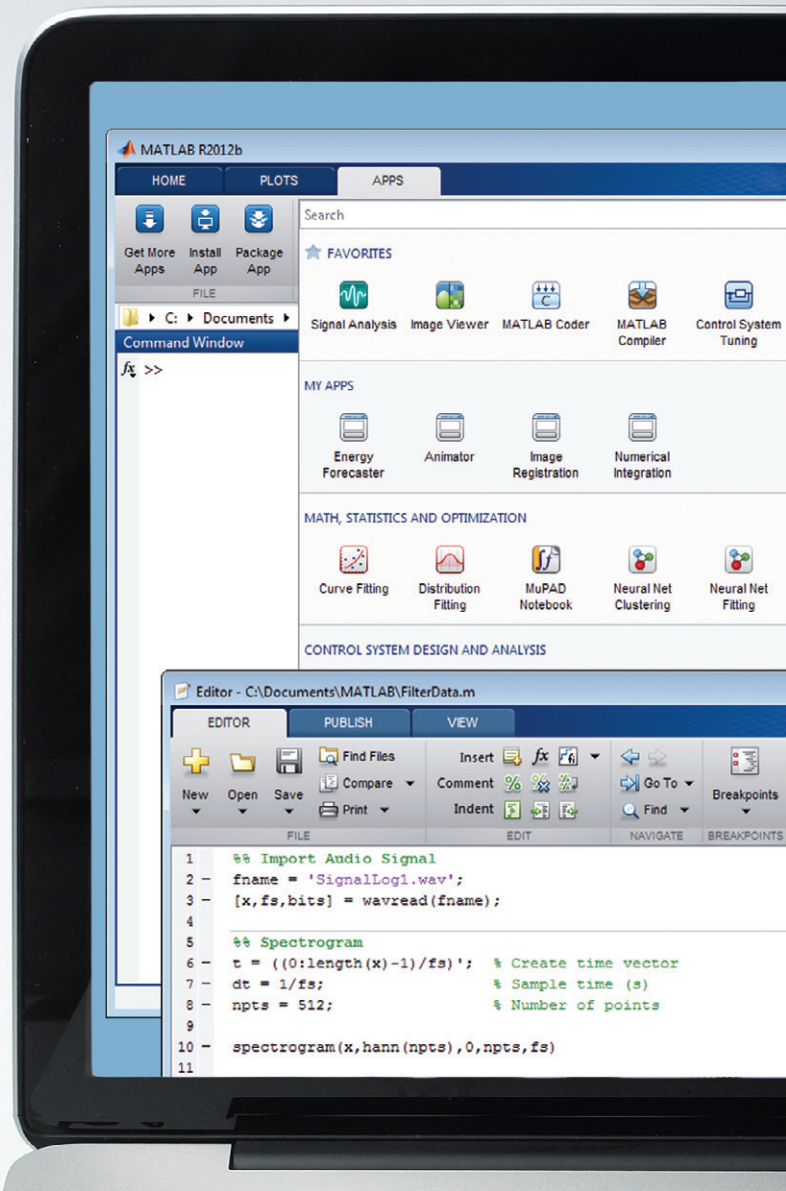
An open exchange for the MATLAB and Simulink user community

MathWorks®
Accelerating the pace of engineering and science

MATLAB Desktop

See what you've been missing.

WATCH VIDEO 5:20
mathworks.com/matlab-desktop



The MATLAB® Desktop makes it easy to find what you need.

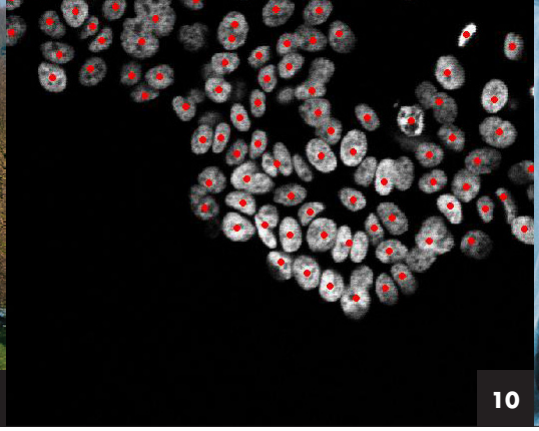
Toolstrip
Highlights commonly used functionality

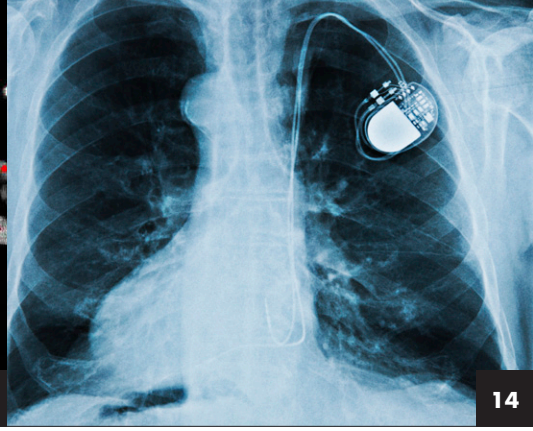
Apps Gallery
Displays in-product and user-written apps

Online Documentation and Redesigned Help
Improves searching, browsing, and filtering

MATLAB® & SIMULINK®

6


10


14

FEATURES

18


30

35

DEPARTMENTS

ABOUT THE COVER

The cover shows an STM32F4 high-performance discovery board with a 32-bit ARM® Cortex®-M4F core. Engineers use this board to implement motor control, audio, and other controls and signal processing applications. Embedded Coder® generates ARM Cortex-M optimized code from filters designed using DSP System Toolbox™. The article on page 4 shows how engineers and scientists connect MATLAB® and Simulink® to hardware in applications ranging from simple classroom projects to high-performance production systems.

Managing Editor
Linda Webb

Editor
Rosemary Oxenford

Art Director
Robert Davison

Graphic Designer
Katharine Motter

Technical Writer
Jack Wilber

Production Editor
Julie Cornell

Printer
DS Graphics

Production Staff
K. Calhoun, K. Camerlin, L. Goodman, L. Heske, K. Kevorkian, L. Macdonald, R. Marks, C. Richer

Editorial Board
T. Andraczek, S. Gage, C. Hayhurst, M. Hirsch, S. Lehman, D. Lluch, M. Maher, A. May, C. Moler, M. Mulligan, L. Shure, J. Tung

Contributors and Reviewers
B. Aldrich, A. Ananthan, G. Argast, P. Barnard, P. Bizzarri, G. Bourdon, M. Carone, B. Chou, K. Cohan, S. Craig, J. Doke, F. Engels, T. Erkkinen, N. Fernandes, D. Ferraro, P. Fricker, J. Friedman, D. Garrison, T. Gaudette, J. Ghidella, D. Gifford, J. Hicklin, D. Hoadley, Z. Jiang, L. Kempler, T. Kush, B. Levy, D. Lisin, K. Lorenc, R. Lurie, B. Madsen, R. Mangharam, J. Mather, B. McKay, L. Mehrez, M. Mestchian, P. Perkins, P. Pilotte, S. Popinchalk, C. Portal, S. Prakash, M. Ricci, R. Rovner, G. Sandmann, G. Sharma, S. Sharma, R. Shenoy, P. Smith, L. Tabolinsky, B. Tannenbaum, A. Thé, G. Thomas, P. Vallauri, N. Vasi, M. Vetsch, M. Violante, I. Wali, E. Wetjen, R. Willey, M. Yale, S. Zaranek

Subscribe
mathworks.com/subscribe

Comments
mathworks.com/contact

FSC MIX From responsible sources FSC® C084277

Printed on 30% post-consumer waste materials

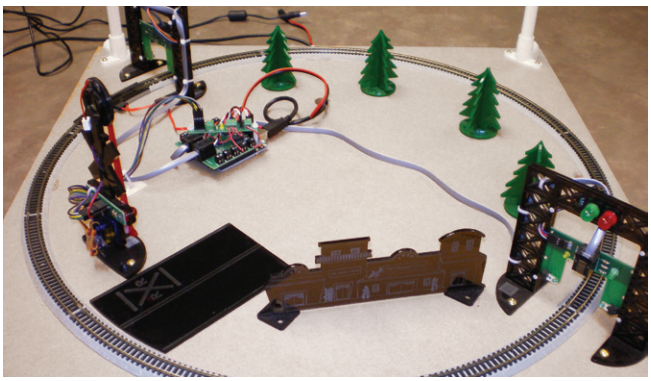
Made in the U.S.

MathWorks®

©2013 The MathWorks, Inc.
MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Connecting MATLAB and Simulink to Hardware

Engineers and scientists connect MATLAB® and Simulink® to FPGAs, microprocessors, cameras, instruments, and other hardware to design, test, and verify systems that combine hardware components and software algorithms. This approach supports project-based learning with low-cost hardware at schools and universities, as well as high-performance production system development at automotive, aerospace, communications, and other commercial organizations.



PROJECT-BASED LEARNING: OHIO STATE UNIVERSITY

Teaching programming with MATLAB and Arduino hardware

For their final project in *ENGR 1181: Fundamentals of Engineering I*, first-year OSU students develop a MATLAB program to control a model train as it travels around a circular track. They implement their programs on an Arduino Uno using the MATLAB Support Package for Arduino Hardware. To optimize limited lab time and hardware, instructors developed a virtual train simulator in MATLAB. All 1700 students can use the simulator to design, test, and debug their algorithm code. They can then use the same code on the actual train set in the lab.
mathworks.com/ohio-state

FPGA DESIGN: SIGLEAD

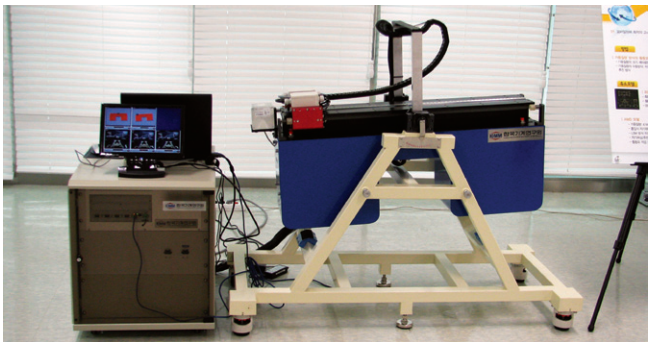
Accelerating development of signal processing components for mass storage devices

Storage devices, including solid state drives (SSDs) and hard disk drives (HDDs), require advanced signal processing subsystems for high-speed data encryption and error correction. Algorithms for these subsystems are often developed in C or C++. Translating the C algorithms into HDL is time-consuming and error-prone, as the sequential behavior of C must be mapped to the parallel behavior of hardware. Siglead engineers use MATLAB, Simulink, and HDL Coder™ to model, simulate, verify, and generate HDL code for the signal processing components in their SSD and HDD systems. With HDL code automatically generated from a verified, cycle-accurate Simulink model, modifications that used to take days are now completed in hours. ***mathworks.com/siglead***

EMBEDDED SYSTEMS: WEINMANN MEDICAL TECHNOLOGY

Implementing embedded software for certification of an emergency transport ventilator

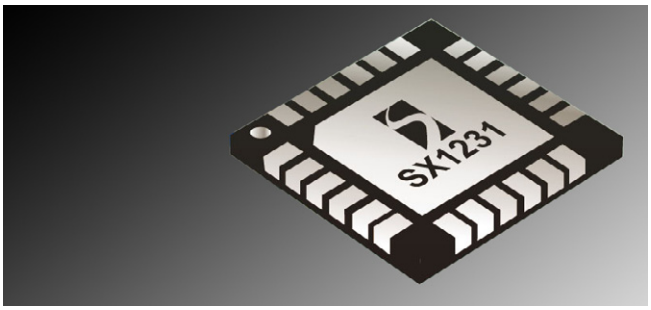
The MEDUMAT Transport ventilator is the most advanced—and the most complex—ventilator that Weinmann has ever developed. Model-Based Design with MATLAB and Simulink enabled Weinmann to handle the increased complexity and achieve compliance certification. Engineers developed a Simulink plant model, which included hardware components and a mechanical model of human lungs, and a Simulink and Stateflow model of the controller. After running closed-loop simulations of the controller and plant, they generated production code for deployment to a 16-bit Infineon® microcontroller using Embedded Coder®. This approach helped Weinmann certify MEDUMAT Transport to ISO/IEC 62304, ISO 10651-3, DIN EN ISO 13485, and DIN EN ISO 14971 standards.
mathworks.com/weinmann



REAL-TIME TESTING: KOREA INSTITUTE OF MACHINERY AND MATERIALS

Creating a real-time prototype of an antirolling control system to stabilize mobile harbors

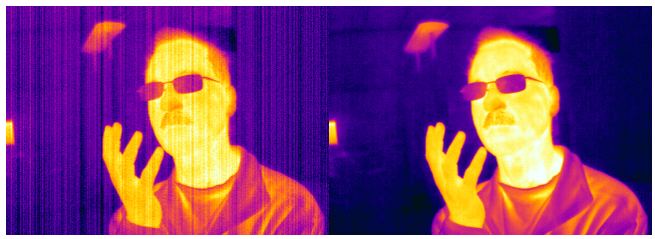
KIMM researchers needed to evaluate an antirolling system for a catamaran-based mobile harbor platform. An aggressive schedule allowed time for only one prototype. The researchers converted a SolidWorks® assembly of the mobile harbor platform into a SimMechanics™ model, and developed Simulink models of the controllers. They performed closed-loop simulations using the catamaran model to verify the control algorithms. C code was automatically generated from the controller models and executed in real time using xPC Target running on a PC/104 computer with an I/O board interface to the catamaran hardware. ***mathworks.com/kimm***



COMMUNICATION SYSTEMS: SEMTECH

Generating production VHDL code for FPGA and ASIC implementation of wireless RF receivers

Semtech engineers needed to accelerate development of optimized digital receiver chains for wireless RF devices. They developed a Simulink model of the complete receiver chain. After converting the model from floating point to fixed point, they conducted bit-true simulations. The team then generated VHDL® from the model, verifying the code by cosimulating their Simulink design with the Mentor Graphics® Questa® simulator. By automatically generating VHDL from the Simulink model, they eliminated the tedious hand-coding of each block, reducing prototype development time from months to weeks. ***mathworks.com/semtech***



COMPUTER VISION: FLIR SYSTEMS

Implementing advanced thermal imaging filters and algorithms

FPGAs in thermal imaging infrared cameras filter and process signals generated by sensors. Turning a new signal processing concept into an algorithm that runs in real time on a production camera can be time-consuming, because hardware engineers must translate algorithms developed by algorithm engineers into HDL. At FLIR Systems, engineers develop and simulate advanced algorithms in MATLAB. Synthesizable HDL code automatically generated by HDL Coder from these MATLAB algorithms is implemented and tested on the FPGA. As a result, prototyping time is reduced by up to 60%, enabling the FLIR team to explore new designs and enhance existing ones.
mathworks.com/flir

Learn More

Hardware Support
mathworks.com/hardware

User Stories
mathworks.com/user-stories

Developing In-Vehicle Traffic Jam Alleviation Technology for Android Using Simulink

By Frank Engels, TNO, TMC

One Sunday last May, a driver on the A270 highway from Helmond to Eindhoven in the Netherlands braked abruptly in heavy traffic. It was the type of sudden slowing that forces drivers following closely behind to brake, causing a ripple effect, or shockwave. Even if the lead vehicle accelerates again immediately, shockwaves can bring traffic to a standstill as each driver in the chain slows to avoid a collision with the driver ahead.

On this particular day, however, there was no traffic jam. This was because 10% to 30% of the cars on the highway were demonstrating a technology, using information from roadside units (RSUs), that damps shockwaves and helps ensure smooth traffic flow after sudden braking incidents[1].

A core component of the technology is the on-board unit (OBU), a hardware device currently under development by TomTom. The OBU, which runs on the Android™ operating system, provides drivers with navigation information. The shockwave damping software installed on the OBU uses a technique known as *cooperative driving*, in which OBUs work together with RSUs to improve traffic flow. TNO engineers modeled the software for the OBU in Simulink®. These engineers then developed a process for deploying the model to the Android-based OBU using Simulink Coder™.

Preparing for Cooperative Driving on the A270 Highway

The Android devices installed on the test vehicles are complemented by monitoring and control hardware installed along a 5-kilometer stretch of the A270. This hardware includes 48 video cameras mounted on poles spaced 100 meters apart and 11 communications gateways spaced 500 meters apart.

A single RSU computer processes images from the cameras, computes an optimal speed profile for vehicles traversing each section of the highway, and transmits this speed profile to the vehicles via Wi-Fi®-p communications gateways.

As soon as the RSU detects the start of a shockwave, it calculates adjusted speed profiles to gradually slow down approaching vehicles. Vehicles farthest from the incident may need to reduce their speed very little, if at all. Vehicles closer to the incident will slow down more, but not so much that they propagate the shockwave.

The behavior of vehicles in communication with the RSU influences the behavior of other vehicles in the traffic flow. As a result, we do not need to equip all vehicles with OBUs. In fact, of the 68 vehicles in the experiments conducted on the A270, only 20 had OBUs. Of these, 8 included an adaptive cruise control system (ACC), controlled directly by the OBU, that automatically sets the vehicle's speed. In the remaining 12 vehicles, the OBU displayed the suggested optimal speed (Figure 2), letting the driver accelerate or decelerate manually[2].

Modeling the OBU Software

The OBU control system determines the current speed of the vehicle using coordinates

Test setup on the A270 highway. Video cameras and wireless communications gateways placed along the road monitor and control the test vehicles to improve traffic flow.

from the OBU's integrated GPS system and its accelerometer. It also processes speed profile updates from the RSU, updates the OBU display, and generates CAN messages to set the ACC speed when the system is running in automatic mode.

We modeled the complete control system in Simulink and Stateflow® and ran simulations using input data recorded from earlier road tests. In the weeks immediately before the road tests, we used Simulink external mode to verify the real-time performance of the system's interfaces with the CAN bus, the RSU, and the on-board display. These simulations enabled us to find and eliminate problems in our algorithms before conducting road tests. This capability was vital to the success of the project, as each road test involved closing the highway to other vehicles and coordinating 68 volunteer drivers.

Generating Code for the Android Platform

Modeling and simulating the OBU control system was relatively straightforward, as the TNO engineering team has significant experience with Model-Based Design. What we lacked was experience in developing software for Android.

Based on the Linux® kernel, the Android operating system is designed to run Java™ applications, or apps. An Android device's input and output—including the accelerometer and GPS data—are available to any Java app running on the device. We could not generate Java code from our Simulink model, nor could we generate C code for a standalone executable and deploy it directly to Android.

To address this issue we developed a novel approach for running a Simulink model on Android. This approach uses the Android Native Development Kit (NDK), a toolset that lets engineers implement portions of their apps using native-code languages such as C and C++. The Java Native Interface (JNI) provides the framework for exchanging data between the Java portions of the app and a shared library created using C or C++.

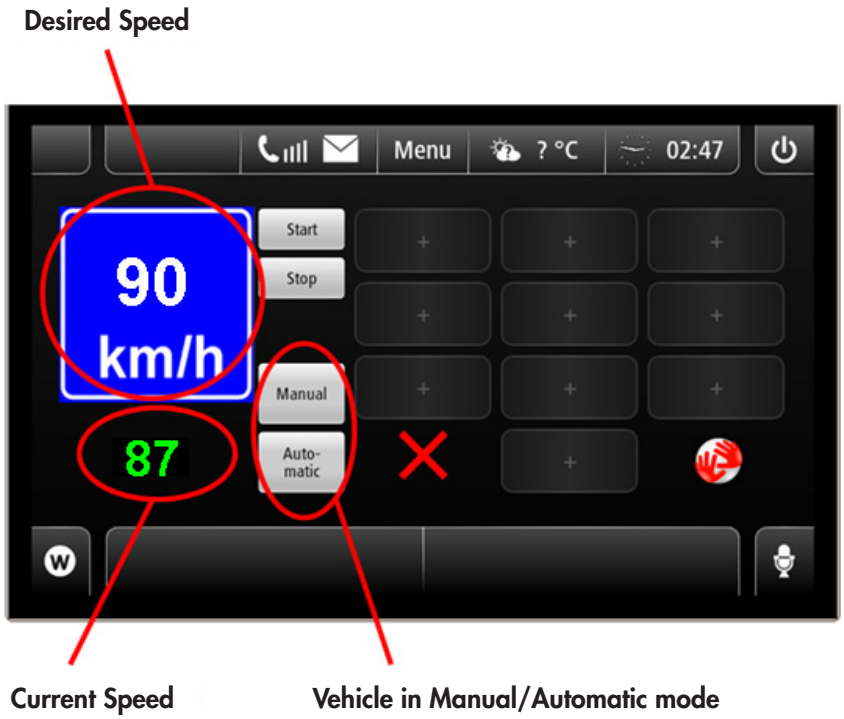


FIGURE 2. *The OBU display panel.*

We created a target for Simulink Coder that generates code for a shared library rather than a standalone executable. The C code in the shared library includes functions for initializing the model, executing a single model step, and closing the model. The code also includes functions for accessing input and output data from Java via JNI and an external mode interface to access parameters and data during execution. Because Java arrays are structured differently from C arrays, the code also includes functions that transform arrays from Java to C and from C to Java. Lastly, we updated the standard template makefile to support compilation for an Android target using the NDK.

Running Road Tests and Processing the Results

Although we had eliminated most problems with the control system through simulation, when we began road tests we discovered some issues with the display and with the processing of the speed profile information from the RSU. To resolve these issues, we had to update our Simulink model in the field and then redeploy the system. The ability to generate code and create an updated Android app with a single click proved invaluable—it enabled us to rapidly make the needed changes and resume road testing in the limited time that we had access to the highway.

Following road tests, we used MATLAB® to postprocess data, including measurements of vehicle speed and the distance between vehicles, that we had collected during the tests. We generated plots of vehicle position as a function of time (Figure 3). The plots confirmed what the tests had demonstrated: The system effectively damped the shockwave and enabled a smooth flow of traffic.

The left-hand plot in Figure 3 shows the

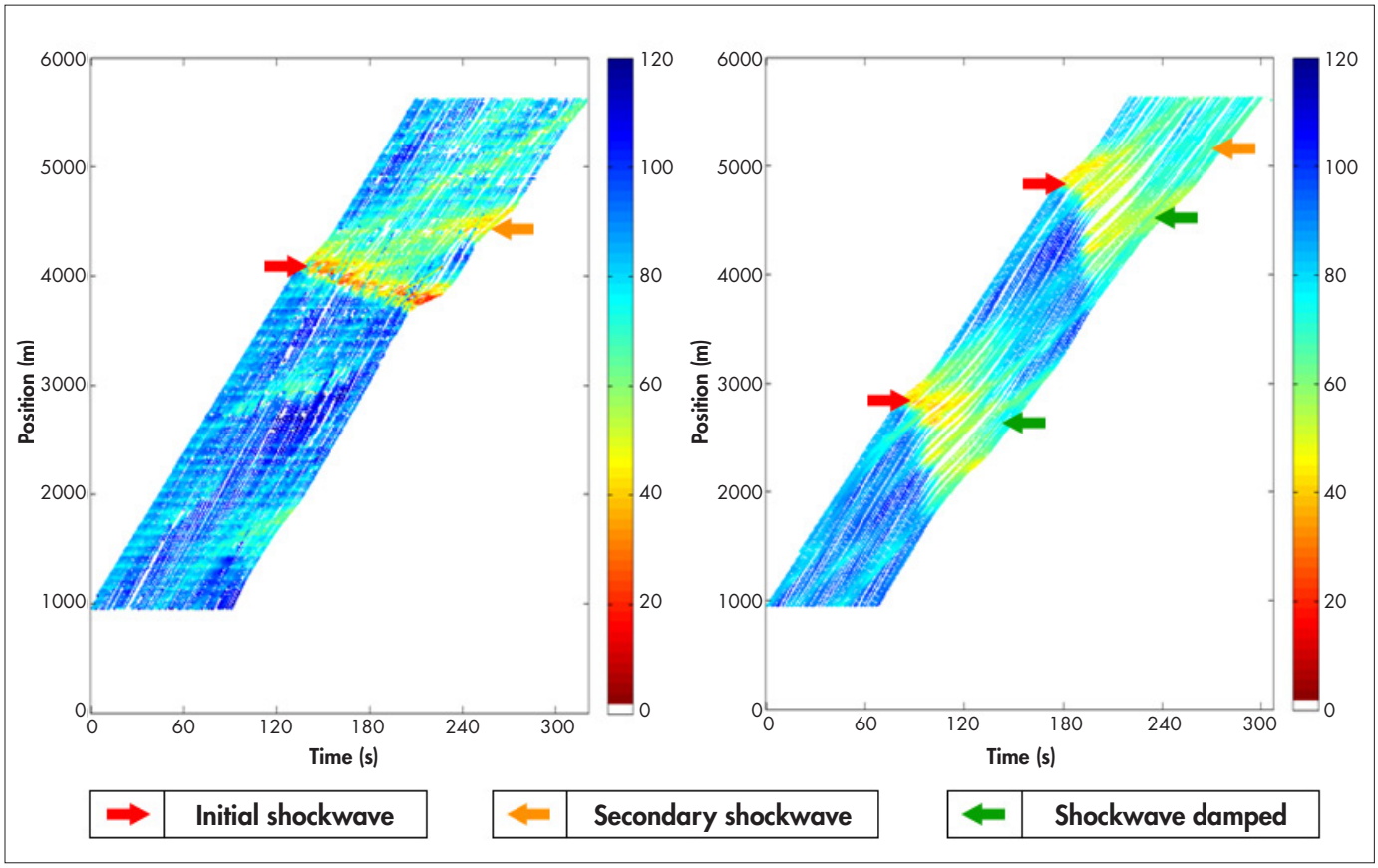


FIGURE 3. *Plots showing vehicle position as a function of time for normal conditions (left) and with the shockwave damping system enabled (right). The color of each plotted position indicates the speed in kph of the vehicle at that position.*

results of a road test with the shockwave damping system turned off. At 3900 meters the lead vehicle brakes, slowing from 85 kph to 45 kph. This creates a shockwave that affects all vehicles, ultimately causing some vehicle speeds to fall below 20 kph. A second shockwave, caused by vehicles' accelerating after leaving the first shockwave, is also apparent[1].

The right-hand plot shows the results of a test conducted under the same conditions but with shockwave damping enabled; the braking action of the first vehicle is exactly the same as in the previous experiment. In this test, the RSU generated a speed profile when it detected the first shockwave. Vehicles equipped with OBUs 1500 meters beyond the shockwave reduced their speed to 65 kph. As a result, gaps emerged in the traffic flow (visible as white zones in the plot). The first shockwave was successfully damped after only 30 seconds. The right-hand plot also shows a second braking action of the lead vehicles. Again, the shockwave was successfully damped. The difference between both braking actions is that during the first, the OBU-equipped vehicles were driving in advisory mode, while during the second, they were driving in automated mode.

We are currently using Model-Based Design and the new Android target for Simulink Coder to develop an enhanced version of the system. This version factors in input from traffic lights to improve traffic flow within and between cities. ▪

[1] B.D. Netten, T.H.A. van den Broek, I. Passchier, and P. Lieverse, "Low Penetration Shockwave Damping with Cooperative Driving Systems," 8th ITS European Congress, Lyon, France, June 6-9, 2011.

[2] T. van den Broek, B. Netten, and P. Lieverse, "Results of Cooperative Driving Applications," 8th International Automotive Congress, Eindhoven, The Netherlands, May 16-17, 2011.

Learn More

Analyzing Test Data from a Worldwide Fleet of Fuel Cell Vehicles at Daimler AG
mathworks.com/fuel-cell-fleet

Developing a Motion-Stereo Parking Assistant at BMW
mathworks.com/parking-assistant

Using Image Processing and Statistical Analysis to Quantify Cell Scattering for Cancer Drug Research

By Gretchen Argast, OSI Pharmaceuticals, LLC and Paul Fricker, MathWorks

Epithelial to mesenchymal transition (EMT), a process vital to embryonic development, has been linked to the spread of cancer in adults. As a result, there is increased interest in developing cancer drugs that target EMT in addition to drugs that target cell proliferation and survival.

Until recently, measuring how a drug affected one aspect of EMT, cell scattering, was a manual process that involved subjectively assessing the relative closeness of cells in a culture. Researchers at OSI Pharmaceuticals worked with MathWorks consultants to develop an automated system for quantifying the scattering of cells in a sample. Based on MATLAB®, Image Processing Toolbox™, and Statistics Toolbox™, the system measures nucleus-to-nucleus distances of nearest-neighbor cells. The ability to measure scattering is essential to evaluating the efficacy of drugs that may inhibit or reverse EMT because it gives researchers a reliable way to compare the effects of one drug against another.

Analyzing Cell Sample Images

OSI researchers have developed pancreatic and lung tumor models and identified a set of ligands, or binding molecules, that drive EMT in these models. Two of these ligands, hepatocyte growth factor (HGF) and oncostatin M (OSM), induced EMT in the models, enabling us to produce samples that demonstrate the cell scattering associated with EMT. The samples are stained so that the nucleus of each cell shows blue in the images captured by our microscopes (Figure 1).

To quantify the scattering of the cells, we developed a numerical procedure that uses image processing and statistical analyses. Measuring the spatial density of the cells would be relatively straightforward if the images were completely covered by the cells: We would simply count the number of nuclei in each image and then divide by the total image area. The images that we generate are almost always partially covered, however, making it difficult to estimate the cell density correctly. We decided to develop an alternative approach to quantify the scattering, based on measurements of the distances between the cell nuclei.

To analyze the cell images, we used an algorithm consisting of four main steps:

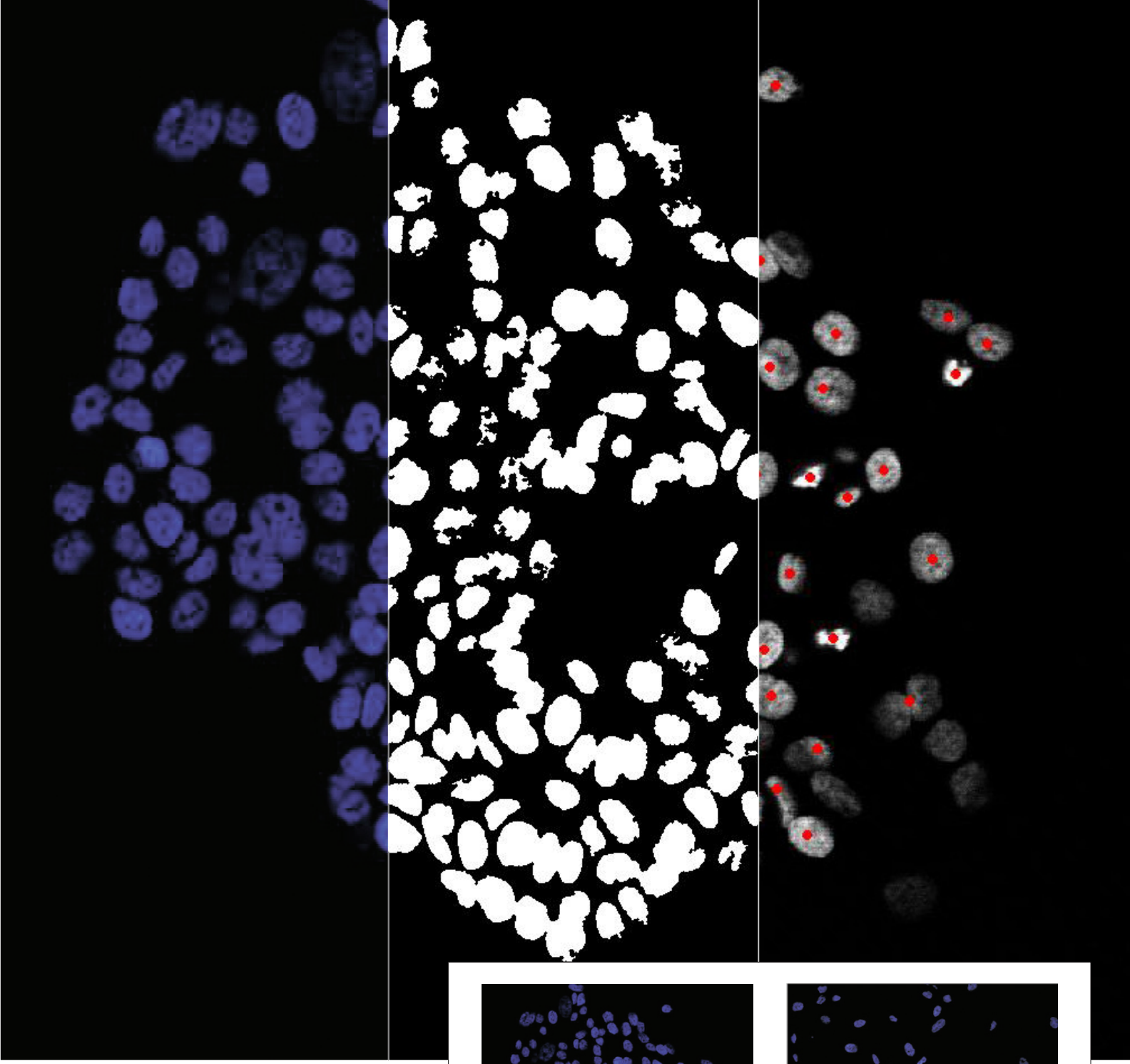
1. Threshold the entire image to segment the cell nuclei, or clusters of nuclei.

2. Analyze the resulting blobs to determine their sizes (areas).

3. Zoom in on larger blobs to perform a localized analysis, to identify individual cells within the blob.

4. Identify the (x,y)-location for each cell nucleus in the image.

Because the intensity scaling is consistent across all the captured images, we can capture most of the individual blobs using a single hard-coded threshold value. This thresholding procedure produces a binary



Processing a cell sample to identify individual cells: original, untreated cell sample (left); cell sample after applying a thresholding procedure (center); and final, processed image (right).

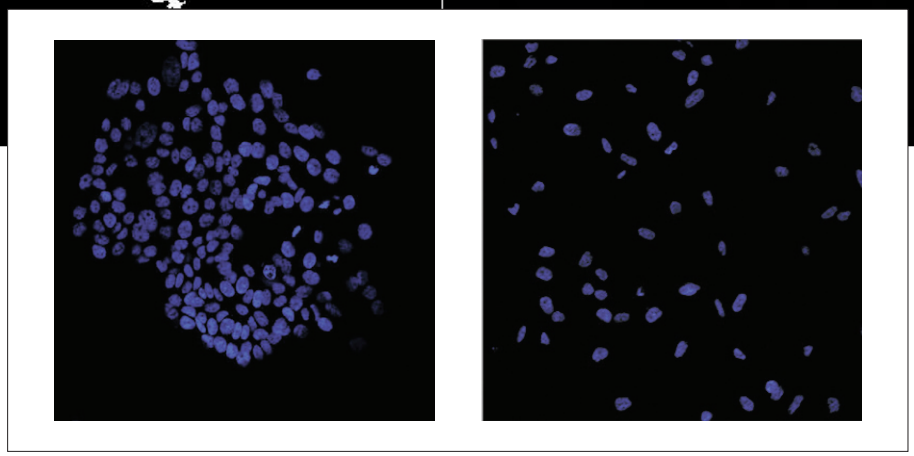


FIGURE 1. *Left: An untreated cell sample showing cell nuclei in blue. Right: A similar sample treated with HGF and OSM (H+O).*

image in which the cell nucleus is indicated by 1, or white, and its absence is indicated by 0, or black (Figure 2). Using Image Processing Toolbox, we analyzed these black and white images to find the locations and sizes (areas) of all the blobs.

In some cases, a few cells are so close together that their nuclei appear to be touching one another, and they cannot be distinguished as separate nuclei. To enhance the processing of the images, we sorted the blobs into three categories based on their size. Those with areas below a certain size were deemed to be noise or partially occluded cells, and were discarded from the subsequent analysis. Blobs of intermediate size were classified as individual nuclei that had already been successfully segmented. The largest blobs were presumed to be clusters of overlapping cells requiring further analysis.

To distinguish the individual nuclei within the larger blobs, the algorithm crops the subregions of the image containing the largest blobs and performs local, adaptive thresholding to more accurately distinguish the individual cells (Figure 3).

At the end of the image analysis procedure, the algorithm has identified the location of most of the cell nuclei in the image, and stored this data in an array. The success of the algorithm can be verified visually by overlaying the input images with markers at each measured nucleus location (Figure 4).

Measuring and Analyzing Distances Between Cells

Once we have processed the images and obtained an array of cell nucleus coordinates, we use basic MATLAB matrix operations to compute the distances between an individual nucleus and all the other nuclei in the cell cluster. To assess the scattering of the cells, we compute the distance between each cell and its nearest neighbor. Each image generates a set of nearest-neighbor distances, with one value for each cell. The distance values computed from the image data are initially measured in pixels, and are converted to microns using a known length scale.

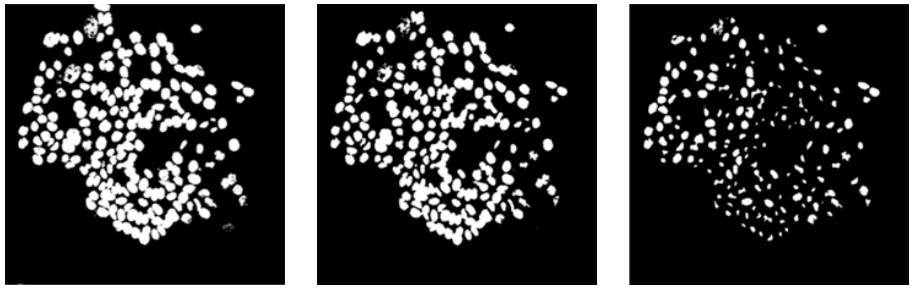


FIGURE 2. *A cell sample image after applying a thresholding and erosion procedure.*

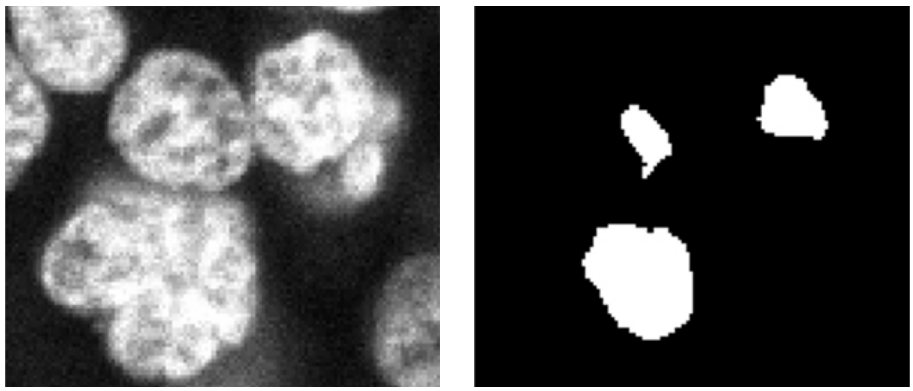


FIGURE 3. *Left: Unprocessed image of a cluster of nuclei initially identified as a single large blob. Right: The same cluster re-analyzed to identify three individual cell nuclei.*

MATLAB histograms of these nearest-neighbor distances show clearly that the data fits into meaningful distribution patterns. These patterns reveal distinct differences between each of the four types of cells that we were studying: untreated, HGF-treated, OSM-treated, and HFG+OSM-treated lung cancer cells (Figure 5).

These histogram results suggested that the data could be characterized using a statistical distribution. Using Statistics Toolbox we fitted the measured distance values to a series of probability distributions. Narrowing our search to asymmetric, continuous distributions, after an iterative process we found that the loglogistic distribution provided the best fit for the nearest-neighbor distance results.

In addition to characterizing the scattering of the cells, one of the main objectives of this project was to develop a method for differentiating the degree of scattering produced by the treatment of cell samples with different ligands. To accomplish this, we used MATLAB to compute the mean (μ) and variance (σ) parameters for the loglogistic distribution for each of the four samples (Figure 6).

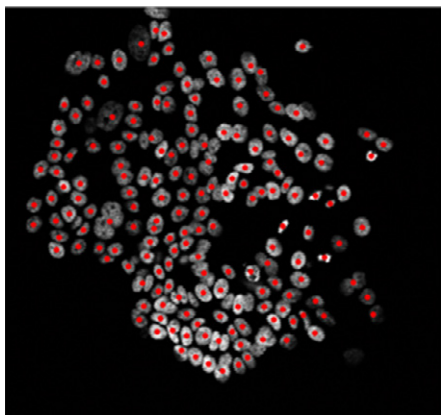


FIGURE 4. *The processed image with each identified nucleus located and marked in red.*

The statistical fitting plots show that the computed values of μ and σ capture distinct

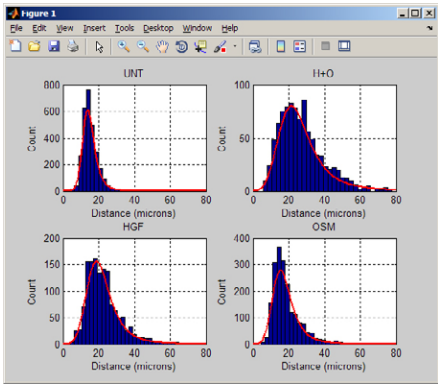


FIGURE 5. *Histograms and curve fitting of nearest neighbor distances for untreated cells (top left), HGF-treated cells (bottom left), OSM-treated cells (bottom right), and HGF+OSM-treated cells (top right).*

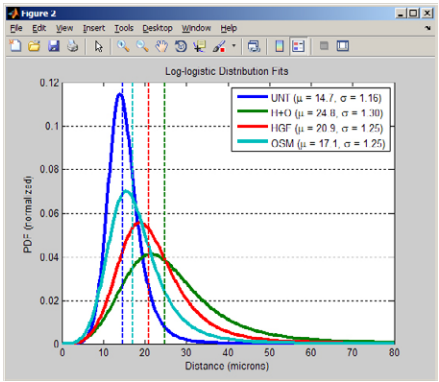


FIGURE 6. *Statistical fitting results for the four nearest-neighbor data sets.*

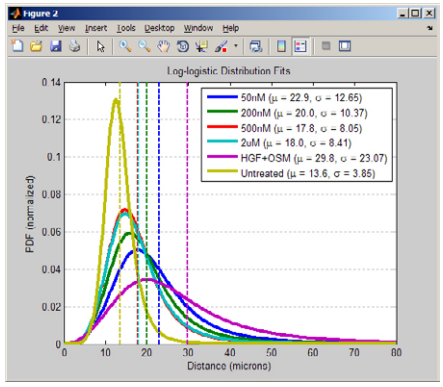


FIGURE 7. *Probability density function (PDF) fitting results for the nearest-neighbor data for a range of doses.*

What is EMT?

In humans and other vertebrates, there are two basic cell types: epithelial and mesenchymal. Epithelial cells depend on cell-to-cell contact for survival. Mesenchymal cells are characterized by their independence from nearby cells and by their mobility, two requirements for cell scattering.

In EMT, cells lose their epithelial traits and acquire mesenchymal traits. In adults EMT is associated with pathologies such as cancer and fibrosis. Scientists believe that mesenchymal tumor cells facilitate metastasis, or the spread of tumor cells. EMT also diminishes the effectiveness of chemotherapy treatments that target epithelial cells.

differences in the magnitude of cell scattering in the four data sets. Conversely, when these parameters are computed for a given data set, they can be used to identify which ligand (HGF, OSM, or HGF+OSM) was used to treat the original cell sample. The distributions show that either ligand alone induced scattering in the cells, and that the combined ligand treatment resulted in a further increase in scattering. These distributions reflect what we observe qualitatively in the cells after treatment with ligands. From these results we concluded that the mean and variance parameters of the loglogistic distribution fitting of computed nearest-neighbor distances could be used to reliably quantify the scattering of cell nuclei in a given sample.

In addition to characterizing the responses of the cells to different ligands, we also looked at the effect of drug treatment on the degree of cell scattering. We computed the loglogistic distributions for samples treated with HGF+OSM that were also treated with increasing concentrations of a drug that blocks the effects of HGF (50 nM to 2 μM) (Figure 7). At concentrations of 500 nM and above, the drug inhibited the effects of HGF and reduced the degree of scattering to one that approximated the effects of OSM by itself. This type of analysis is essential for determining the optimal dose for a new drug.

At the beginning of the EMT quantification project, our goal was to use image analysis techniques with our microscope data to quantify the scattering or density of cells in our samples. After successfully analyzing the basic attributes of the cell nuclei using MATLAB and Image Processing Toolbox, we realized that the resulting data could best be characterized in terms of a statistical distribution. It was easy to transition to a statistical analysis of the data using Statistics Toolbox. MATLAB enabled us to work within a single development environment, from the initial image thresholding and nearest-neighbor distance calculations, through selecting and validating an appropriate statistical distribution, to the final comparison of different ligand dose responses.

With a system in place for quantifying the scattering of cells in a sample, OSI researchers now have an objective computational method for measuring the ability of drugs in development to reduce or reverse EMT, and potentially, for increasing the drug's ability to inhibit cancer metastasis. ▪

Learn More

New Features for High-Performance Image Processing in MATLAB
mathworks.com/hp-ip-matlab

MathWorks Consulting Services
mathworks.com/consulting

University of Pennsylvania Develops First Electrophysiological Heart Model for Real-Time Closed-Loop Testing of Pacemakers

By Zhihao Jiang and Rahul Mangharam, University of Pennsylvania

From 1990 to 2000, manufacturers recalled more than 600,000 cardiac medical devices. About 200,000 of those recalls were due to software problems.

With a typical pacemaker containing 80,000 to 100,000 lines of code, it is difficult for manufacturers to identify errors using current industry practices. These practices rely on open-loop testing in which testers feed prerecorded heart signals to the pacemaker and evaluate the corresponding output. Open-loop testing can reveal how the pacemaker responds to various heart conditions, but not how the heart responds to the pacemaker in a closed-loop system.

At the University of Pennsylvania Department of Electrical and Systems Engineering, our team has developed a first-of-its-kind electrophysiological model of the heart that enables real-time closed-loop testing of pacemakers. Developed with MATLAB® and Simulink®, this heart-on-a-chip system can be configured to match a patient's specific electrophysiological characteristics, and it can simulate a variety of heart conditions to enable early verification of pacemaker software.

Building the Heart Model

To better understand how electrical signals work in the heart, we sat in on heart surgery operations at the Hospital of the University of Pennsylvania (Penn). During these operations, physicians in the electrophysiology department direct an electrical current into the heart via a catheter. They then monitor the electrical behavior of the heart and measure conductivity along pathways.

We created an initial electrophysiological model of the heart in MATLAB. We chose MATLAB for several reasons. First, we needed to conduct extensive design exploration in the early stages of development, and MATLAB is an excellent environment for trying out new ideas. Second, most of our graduate students are already experienced MATLAB users, so they can contribute to the project immediately, without having to learn new software. Third, MATLAB and Simulink support code generation for FPGAs and embedded systems, which we needed for real-time testing.

After simulating and refining the MATLAB heart model, we created a Simulink and Stateflow® version. This model gave us a visual representation of the nodes and pathways in the heart (Figure 1), and enabled much faster simulations than the MATLAB version. The Simulink and Stateflow model comprises approximately 30 nodes and 30 pathway submodels, which we developed as Stateflow charts. The node automaton submodel comprises three states corresponding to a rest state and two active states, the

A chest x-ray image showing a heart with a pacemaker installed. The heart and pacemaker form a closed-loop system in which changes in the heart's behavior lead to changes in pacemaker activity, and vice versa.

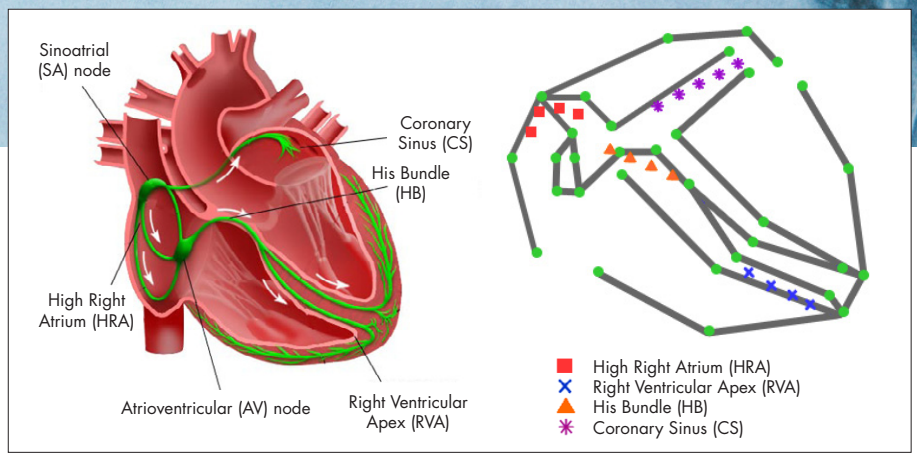


FIGURE 1. *The electrical conduction system of the heart (left), represented as a network of nodes and pathways (right).*

effective refractory period (ERP) and the relative refractory period (RRP) (Figure 2).

The path automaton is more complex, comprising five states that describe the conduction of the path: ante (forward conduction), idle (no conduction), retro (backward conduction), double (conduction in both directions), and conflict.

Building the Pacemaker Model

Before modeling the pacemaker in Simulink and Stateflow, we built a model specifically for formal verification using the UPPAAL modeling environment, which was developed jointly by Aalborg University in Denmark and Uppsala University in Sweden. Based on a specification provided by a pacemaker manufacturer, this model enabled us to explore the design space and prove that specific properties of the design met the requirements. A more conventional approach would have been to start with the Simulink and Stateflow model and then perform the formal verification using Simulink Design Verifier™. We plan to use Simulink Design Verifier for formal verification of the timing of events and for the run-time and safety properties of the design.

We used UPP2SF, a model translation tool developed at Penn, to automatically convert our timed-automata-based models in UPPAAL into a Simulink and Stateflow model for simulation and testing (Figure 3).

Closed-Loop Simulation and Real-Time Implementation

With Simulink and Stateflow versions of the heart model and pacemaker model completed, we conducted closed-loop simulations to verify pacemaker behavior. For example, we simulated several arrhythmias, including atrial fibrillation and bradycardia; scenarios that our formal verification analysis highlighted as potentially problematic; and failure conditions, such as a displaced pacemaker lead. We also used simulation to compare and evaluate different versions of the pacemaker algorithms.

Pacemakers and the Cardiac Conduction System

From one engineering perspective, the heart is a mechanical pump that circulates blood throughout the body. From the perspective of a pacemaker, it is a real-time system requiring precise timing of electrical signals.

The cardiac conduction system is a network of nodes and pathways carrying electrical signals that cause the chambers in the heart to contract and relax (Figure 1). In a healthy heart, contractions happen in rhythm. Disruptions in the cardiac conduction system can cause fast heartbeats—including atrial flutter and fibrillation—as well as slow heartbeats, known as bradycardias.

Pacemakers are fitted with electrodes that sense the heart's electrical activity. When the pacemaker software detects an abnormal heart rhythm, it sends precisely timed electrical pulses to one or more of the heart's chambers. The heart and pacemaker form a closed-loop system in which changes in the heart's behavior lead to changes in pacemaker activity, and vice versa.

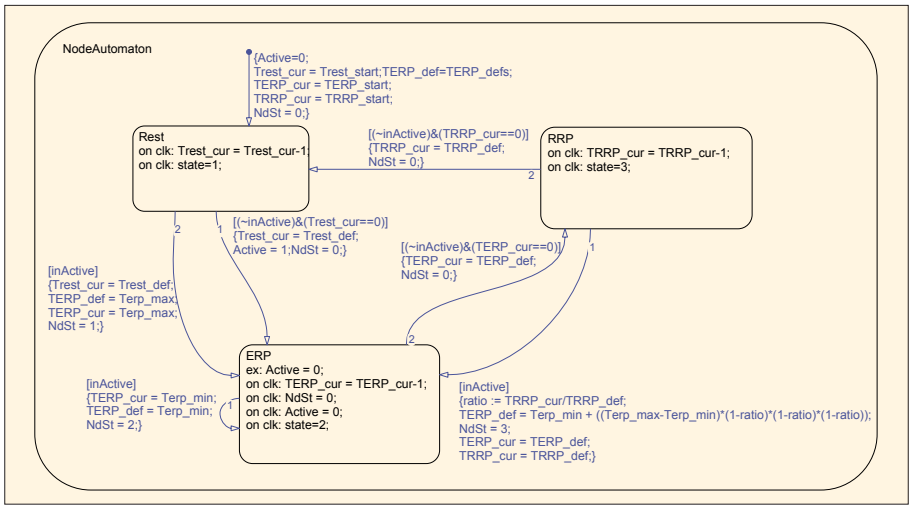


FIGURE 2. *Stateflow chart of a node automaton.*

Testing actual pacemakers—not just models—is one of our principal objectives. To achieve this goal, we needed to implement our heart model on real-time hardware. We used HDL Coder™ to generate VHDL® code from the Simulink and Stateflow model, enabling us to deploy the heart-on-a-chip on an Altera® FPGA (Figure 4). The generated code was so efficient that we were able to implement multiple versions of the heart model on a single low-end FPGA.

For our first real-time tests, we built a simplified pacemaker by generating C code from our Simulink and Stateflow pacemaker model with Embedded Coder® and deploying it to an Atmel® embedded microcontroller. Following the success of those tests, we conducted closed-loop tests using the FPGA heart-on-a-chip and production pacemakers.

Creating an Interface for Configuring the Model

Our heart model is highly configurable, and can account for variations in electrophysiological conductivity across patients. At first, we made configuration changes directly in the Simulink model. To simplify this step for our team and other researchers, we developed the Penn Virtual Heart

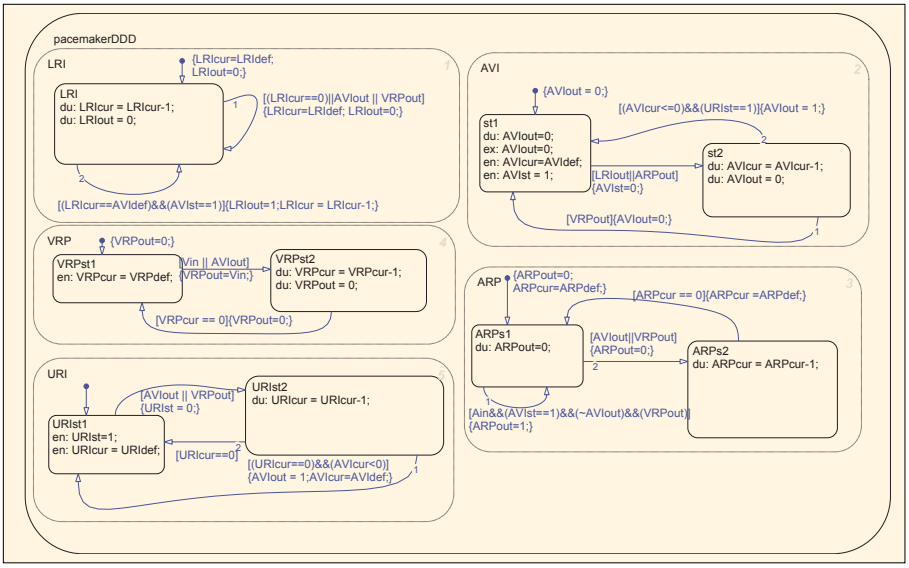


FIGURE 3. *Stateflow chart of the pacemaker design.*

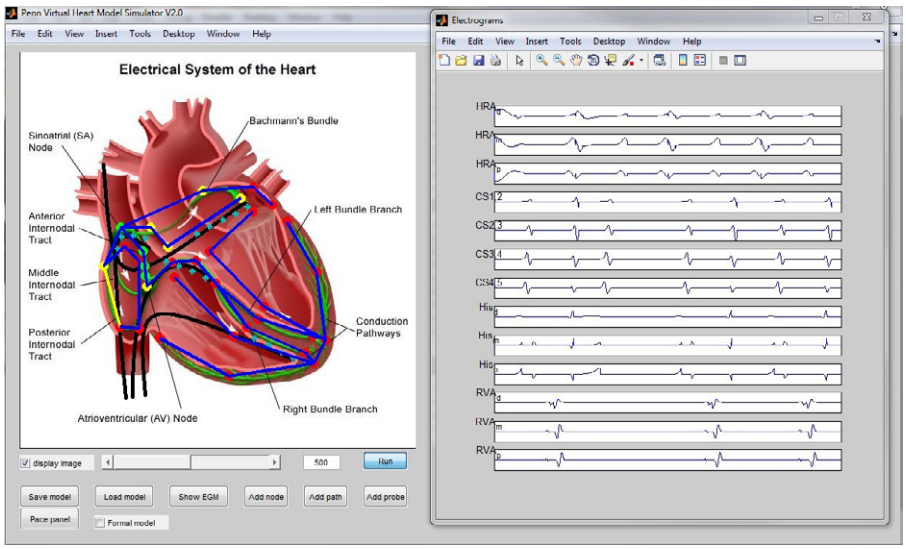


FIGURE 5. *Penn Virtual Heart Model Simulator interface.*

Model Simulator, an interface built with MATLAB that lets users specify a topology for a new heart model and then automatically configure the model based on the requested topology and configuration parameters (Figure 5).

Behind the scenes, the Penn Virtual Heart Model Simulator uses a MATLAB script to generate a Simulink and Stateflow model based on the requested topology and configuration parameters.

Next Steps

We are working with the U.S. Food and Drug Administration and pacemaker manufacturers to establish a framework and guidelines for using our heart model for early verification of pacemaker software. We plan to develop a system that automatically translates a patient's electrophysiology test results into a customized model with parameters optimized for that patient's heart.

Our work on the heart model complements

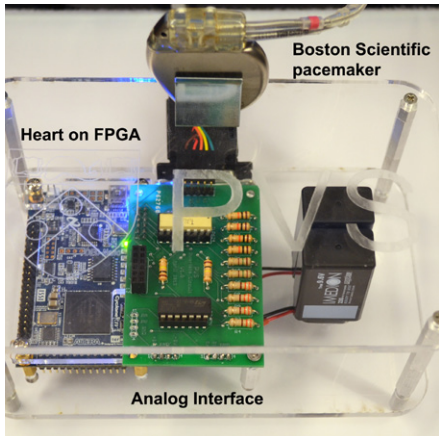


FIGURE 4. *FPGA implementation of the heart model.*

a related project in which Penn researchers developed generic infusion pump models and reference specifications that are used to verify safety properties in medical infusion pumps. Both projects aim to provide realistic models of biological systems to enable improved testing of medical equipment. We plan to continue advances in this area by developing Simulink models of human systems that interact with a range of medical devices, including analgesic pumps and blood glucose and insulin monitoring systems. By enabling early verification of software as well as real-time, closed-loop testing, these models will make it easier for medical device manufacturers to improve the quality of their software and reduce the number of device recalls. ▪

Learn More

Medrad Ensures Safety of MRI Vascular Injection Pump
mathworks.com/medrad

Weinmann Develops Life-Saving Transport Ventilator Using Model-Based Design
mathworks.com/weinmann

Teaching Model-Based Design at Politecnico di Torino

By Massimo Violante, Politecnico di Torino

In the region around Turin, Italy, there is strong demand for engineers with the skills and knowledge required to develop complex, high-integrity embedded software. The need is particularly acute among automotive companies, but it affects avionics and other industries, as well. Companies seek engineers who can deliver sophisticated software in compliance with safety standards such as ISO 26262 and DO-178B while adhering to stringent quality requirements and tight production deadlines.

To meet this demand, Politecnico di Torino has introduced *Model-Based Software Design*, a course for fifth-year students that combines lectures and practical exercises with seminars conducted by local industry experts. Using MATLAB®, Simulink®, Stateflow®, and Embedded Coder®, students learn how to manage the growing complexity of today's embedded systems by designing and simulating an executable model; rigorously validating, testing, and debugging it; and generating code for an embedded target. By fostering collaboration and knowledge-sharing between academia and industry, the course is helping prepare the students for the engineering challenges they will face in their future jobs.

Course Goals

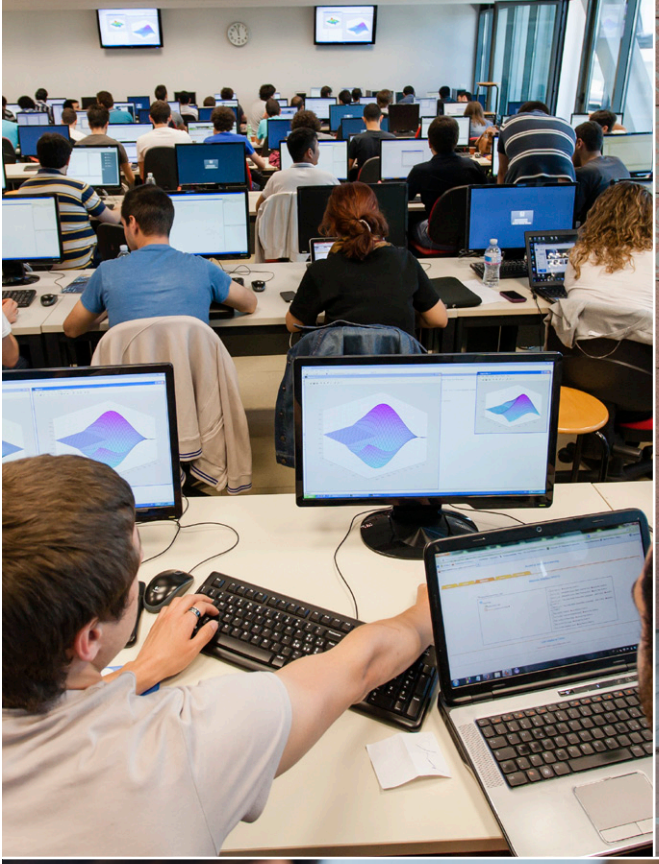
The course was introduced a year ago to cover topics not fully addressed by the computer, electronic, and mechatronic engineering curricula, and to better prepare students for engineering careers in local industries, primarily the automotive industry.

In my research and collaboration with companies in and around Turin, I heard repeatedly that there is a shortage of skilled engineers who can apply Model-Based Design to embedded systems development. Companies see the value in using this approach, but have limited time and resources to train engineers in-house. In developing the course, my goal was thus to prepare students to step into these in-demand roles and be productive immediately. Because the course would clearly benefit local area industries, many companies agreed to support the course by sending engineers to conduct seminars and presentations on their work.

The first time it was offered, *Model-Based Software Design* became the most popular of the 14 elective courses open to computer, electrical, and mechatronics students in their final year at Politecnico di Torino. Of approximately 200 engineering students 73 enrolled, reflecting a strong motivation among the students to develop expertise in this discipline. To date, 107 students have already enrolled in the second session of the course. The 46% increase in enrollment year-over-year is a further indicator of the strong interest the course has generated.

Teaching Modeling, Simulation, and Code Generation

I begin the course with a series of lectures on using modeling and simulation as the core of a



We designed the Model-Based Software Design *course curriculum to help students find jobs with local companies. At the same time, we wanted to provide a service to local industries by training students in the skills those industries need. The course's success can be measured by the high enrollment rate and by the fact that eight students have already started master's theses in cooperation with automotive companies in the Turin area.*

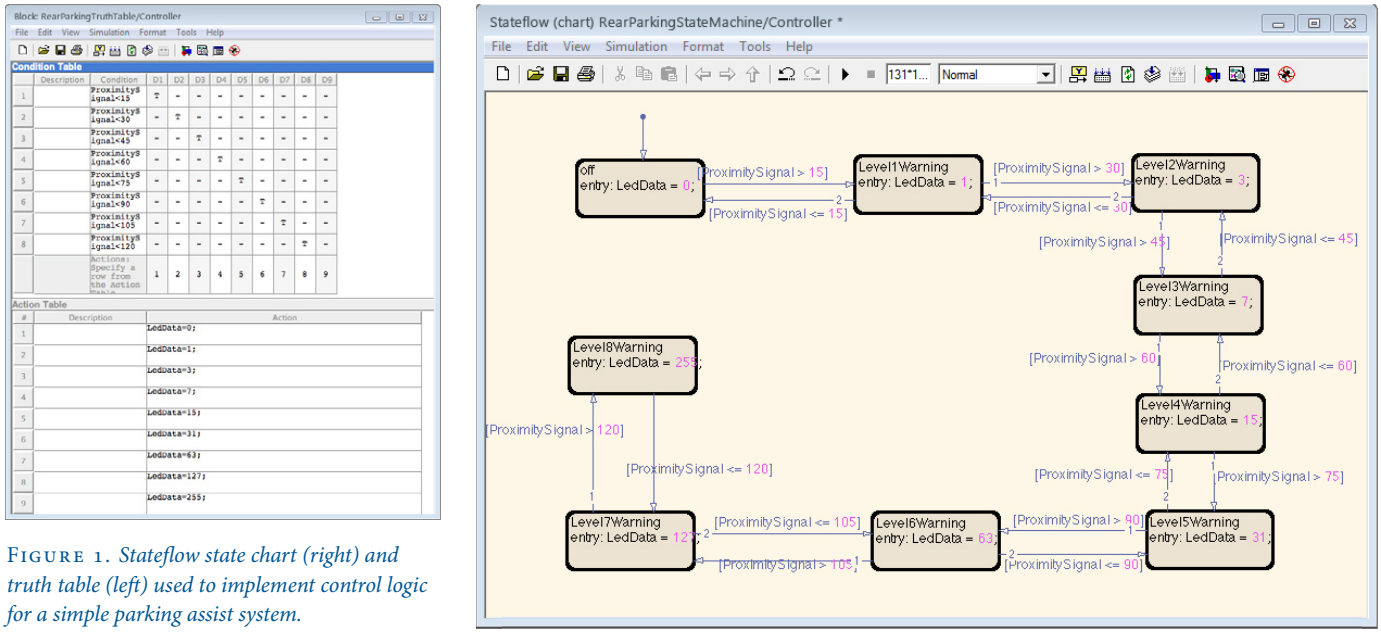


FIGURE 1. *Stateflow state chart (right) and truth table (left) used to implement control logic for a simple parking assist system.*

development process for embedded software. I explain the factors motivating companies to take this approach, including rapid growth in the complexity of the systems under development, time-to-market pressures, and the high cost of defects found late in development.

Next, I describe how Model-Based Design with MATLAB and Simulink enables engineers to model complete systems, validate the system specification and functionality via simulation, and automatically generate test cases and embedded code.

Most students have used MATLAB and Simulink in earlier controls classes or other coursework, so they come up to speed quickly on the practical exercises.

In one exercise, the students use Model-Based Design to build a parking assist system. The system includes a proximity sensor, control software running on a Cypress programmable system on a chip (PSoC® 5, donated to Politecnico di Torino by the Cypress University Alliance), and an array of LEDs that light up when the proximity sensor approaches an obstacle.

The control software is developed entirely in Simulink and Stateflow using state charts and a truth table (Figure 1).

After simulating and verifying their designs, the students use Embedded Coder to generate code for the Cypress PSoC target. We then cover verification and validation, which provides a natural lead-in to a series of lectures on ISO 26262.

Working with ISO 26262, AUTOSAR, and Other Standards

Many automotive companies must adhere to ISO 26262, the functional safety standard for road vehicle software systems. Despite its prevalence in the industry, ISO 26262 is a new concept for the students, who have never had to follow rules governing how they design and build software. Over the course of several lectures, and with the support of my colleague Professor Maurizio Morisio, I provide an introduction to real-world safety standards, including ISO 26262, IEC 61508, and DO-178. Students learn that developing to these standards requires additional work, but that automation and available resources, such as qualification kits, can help reduce the effort required. For example, I introduce them to DO Qualification Kit (for DO-178) and IEC Certification Kit (for ISO 26262 and IEC 61508), and show how these products can streamline the certification process within Model-Based Design.

Like ISO 26262, the AUTOSAR architecture is a concept that is both new to and immediately useful for many students upon graduation. Students learn that with a layered structure, software functionality can be decoupled from the hardware used to implement it. They readily grasp that models enable them to capture and simulate the behavior of an AUTOSAR software component and to generate code for AUTOSAR environments.

Seminars by Industry Experts

Students sometimes view a topic covered in lecture as an academic exercise. When that topic is presented by a professional engineer, however, they can immediately see its practical relevance.

Throughout the course, speakers from various industries provide additional motivation and insights via in-class seminars on a variety of subjects. This year, speakers from two local automotive companies presented their experiences of applying Model-Based Design.

Working in the lab with some of my students on the parking assist system. This project gives students an opportunity to apply what they have learned about code generation, verification, and validation.

These sessions were followed by a seminar on DO-178 presented by an avionics engineer. Our speaker on ISO 26262 discussed gap analysis—understanding what is needed to make an existing development process compliant with the functional safety standard—using the largest carmaker in Italy as a case study. Lastly, an embedded system designer who develops powertrain ECU software gave a presentation on MISRA® coding guidelines.

In addition to these industry experts, I received extensive support from MathWorks engineers in Italy, who demonstrated how MATLAB and Simulink enable Model-Based Design.

Advancing to Graduate Study

Several of the students who completed the course are now working on master's theses on a range of topics involving Model-Based Design. Because Model-Based Design is the de facto industry standard in automotive, avionic, and industrial applications, the students often have the opportunity to work in cooperation with industry partners. Two students are working with a local automotive company on best practices for modeling control algorithms and for verification and validation. Two more are developing a control system for an industrial vehicle using Model-Based Design, and three others are working on an electric vehicle for a competition. Finally, two students are working on a master's thesis to develop an open-source AUTOSAR stack that students who take the course next year will be able to use in the lab.

Next year's course will include additional hands-on lab exercises, enabling students to see more of the systems they design and develop in Simulink running on embedded systems. ▪

Learn More

University of Adelaide Undergraduates Design, Build, and Control an Electric Diwheel Using Model-Based Design
mathworks.com/diwheel

Embedded Control Systems Education at the University of Michigan
mathworks.com/embedded-michigan

Automating Image Registration with MATLAB

By Garima Sharma and Andy Thé, MathWorks

Image registration is the process of aligning images from two or more data sets. It involves integrating the images to create a composite view, improving the signal-to-noise ratio, and extracting information that would be impossible to obtain from a single image. Image registration is used in remote sensing, medical imaging, cartography, and other applications that rely on obtaining precise information from images—for example, discovering from satellite images how an area became flooded, or detecting tumors from MRI scans.

Determining an effective image registration approach is situation-dependent, and can be a complex and time-consuming process. It requires careful selection of a point transformation model to provide reference points between the images, and a method for comparing information to identify the parameters required to properly align the images.

There are two well-known approaches to the process of automatic image registration: feature-based and intensity-based registration algorithms. In this article we will use a fever-detection example to illustrate an intensity-based automatic image registration workflow based on `imregister()` and related capabilities in Image Processing Toolbox™. This workflow is a fast and effective way to integrate images from different cameras.

Fever-Detection Example: Goals and Challenges

During the outbreak of severe acute respiratory syndrome (SARS) in 2003, Taiwan's Taoyuan International Airport began screening passengers for fever symptoms to contain the spread of the deadly virus. Since it was impossible to examine each passenger individually, clinicians used infrared thermography, a noninvasive technique for detecting fever by analyzing infrared images of thermal data.

While this approach is effective, it can be challenging to implement. Infrared cameras are extremely sensitive to environmental conditions, and they must be correctly calibrated to account for all the elements that can affect the temperature reading—including ambient room temperature, relative humidity, reflective surfaces, and the distance of the subject from the camera. Effective thermal screening also depends on consistent identification of a body part that can produce reliable thermal information—in our example, the area around the eyes.

We will build a screening thermography prototype using MATLAB®, a FLIR infrared (IR) camera, and a webcam. The IR camera can measure facial temperatures in increments of 100 millikelvins, while the webcam provides more detailed information on the facial features. By registering the images from both sources, we will be able to detect the location around the eyes from the webcam image (Figure 1)

and measure the temperature around the eyes from the IR camera image.

Acquiring the Images and Calibrating the IR Camera

Using Image Acquisition Toolbox™ we capture the images from the webcam and the IR camera and import them into the MATLAB workspace. The IR camera uses the GigE Vision® interface, while the webcam uses the standard DirectShow® interface.

To calibrate the IR camera we adjust for subject distance, humidity, emissivity (the relative power of a surface to emit heat by radiation), and other characteristics. During the image acquisition process, the atmospheric temperature was 295.15K and the emissivity of the wall and the subjects was 0.98 (code excerpt 1, page 24).

Visualizing the Images

We view the IR image using the `imshow()` function in Image Processing Toolbox. Because we have captured 16-bit data, in which the actual temperature readings were measured in 100 mK increments, we perform a contrast adjustment to scale the data before displaying the image on the monitor (Figure 2).

We display the two images in the same figure window using `imshowpair()` (code excerpt 2).

This function provides several visualization options, including `'falsecolor'`, for creating a composite RGB image using different color bands, and `'blend'`, for visualizing alpha blended images (Figure 3).

FIGURE 1. *Webcam image.*

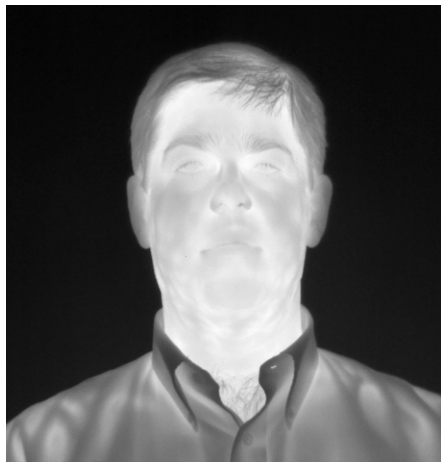


FIGURE 2. *Thermal IR image with contrast adjustment.*

Registering the Images

We begin the registration process by designating the IR image as the fixed image and webcam image as the moving image. The fixed image is the static reference. Our goal is to align the moving image with the fixed image. Since intensity-based image registration algorithms require grayscale, we convert the color webcam image to grayscale using `rgb2gray()`.

To align the images, we use the Image Processing Toolbox `imregister()` function. In addition to a pair of images, intensity-based automatic image registration requires a metric, an optimizer, and a transformation type. We obtain the `'metric'` and `'optimizer'` values using `imregconfig()` with the `'multimodal'` option. We then plug the returned values into `imregister()` as a starting point for the image registration (code excerpt 3).

To begin the registration process we use

FIGURE 3. *Left to right:* `'falsecolor'`, `'diff'`, *and* `'blend'` *visualizations of both images.*

Image Registration Code Excerpts

Code Excerpt 1

```
%% IR Camera object creation and configuration:
irCam = imaq.VideoDevice('gige',1,'Mono16');
irCam.ReturnedDataType = 'native';
irCam.DeviceProperties.IRFormat =
  'TemperatureLinear100mK';
irCam.DeviceProperties.ObjectDistance = 0.91;
irCam.DeviceProperties.AtmosphericTemperature =
  295.15;
irCam.DeviceProperties.ObjectEmissivity = 0.98;
%% Webcam object creation:
webCam = imaq.VideoDevice('winvideo');
%% Acquire snapshots
imageIR = step(irCam);
imageRGB = step(webCam);
```

Code Excerpt 2

```
%% Read in the image pair
Fixed = imread('IR _ image.png');
Moving = imread('Webcam _ image.png');
Moving = rgb2gray(Moving);
%% View the 2 images
imshow(Moving);
% Use imtool to view the IR image
imtool(Fixed);
%% View the images side by side in a montage
imshowpair(Fixed,Moving,'montage');
```

Code Excerpt 3

```
%% Configure parameters in imregconfig
[optimizer,metric] = imregconfig('Multimodal');
```

Code Excerpt 4

```
%% Default registration
registered = imregister(Moving,Fixed,
  'translation',optimizer,metric);
figure;
imshowpair(registered,Fixed);
title('Default registration');
```

Code Excerpt 5

```
%% Final registration
registered = imregister(Moving,Fixed,
  'Similarity',optimizer,metric);
figure;
imshowpair(registered,Fixed);title('Final
  Registration');
```

Code Excerpt 6

```
%% Detect the eyes in the RGB image
eyesDet = vision.CascadeObjectDetector('EyePair
  Small');
bbox = step(eyesDet, Moving);
drawBox = vision.ShapeInserter('BorderColor',
  'Black');
image = step(drawBox, registered, int32(bbox));
hold on; rectangle('Position',bbox,'EdgeColor',
  [1 1 0]);
subsIR = int32(bbox(:,1:2)+bbox(:,3:4)/2);
%% Compute temperature near the eyes
value = mean2(imcrop(registered,bbox));
foreheadTemperature = value/10 - 272;
% In Celsius
foreheadTemperature =
  (foreheadTemperature*9/5) + 32;
% Convert to Fahrenheit
%% Embed temperature on IR image and display
ti = vision.TextInserter('Color',[255 0 0]);
ti.Location = int32(bbox(:,1:2)+bbox(:,3:4)/2);
ti.Text = sprintf('%3d F',
  int8(foreheadTemperature));
contAdj = vision.ContrastAdjuster('CustomProduct
  InputDataType',numerictype([],32,8));
imageContrastAdjusted = step(contAdj, Fixed);
textAdded = step(ti, imageContrastAdjusted);
text(320, 180,'98 \circ F ','Color',[1 1 0]);
```

FIGURE 4. *Default registration.*

FIGURE 5. *Final registration.*

FIGURE 6. *Registered image with temperature reading.*

Image Registration Glossary

Reference (fixed) image: Image in the target orientation, specified as a 2D or 3D grayscale image.

Target (moving) image: Image to be transformed into alignment with the reference image, specified as a 2D or 3D grayscale image.

Intensity-based registration: The alignment of images based on their relative intensity patterns.

Feature-based registration: The alignment of images using feature detection, extraction, and matching.

the `imregister()` default transform type `'translation'` and view the result with a call to `imshowpair()`. The outlines of the subject in the two images are somewhat misaligned (Figure 4). The gaps between the images around the head and shoulders indicate issues with both scale and rotation (code excerpt 4).

Note that getting good results from optimization-based image registration frequently involves multiple modifications of optimizer and metric values. Note, too, that while `imshowpair()` in its default mode works well for the images in our example, it might not work for all image pairs. It is best to explore all the visualization styles in `imshowpair()`, such as `'falsecolor'`, `'diff'`, `'blend'`, and `'montage'`, to identify the best one for a specific image pair.

To account for the scale and rotation distortion, we switch the transformation type in `imregister()` from `'translation'` to `'similarity'` (code excerpt 5).

We now have a reasonably accurate registered image in which the eyes are closely aligned (Figure 5).

The green and magenta areas are present because the images come from different sources. They do not indicate misregistration.

Detecting the Eyes and Reading the Temperature

To detect the eyes, we use the cascade object detector in Computer Vision System Toolbox™. This object detector uses the Viola-Jones algorithm, which detects the eyes using Haar-like features and a multistage Gentle Adaboost classifier. We then draw a bounding box near the eyes to highlight the region of interest on the registered image (code excerpt 6).

Because we have registered the images, we can use the bounding box enclosing the eyes in the webcam image to sample temperature values near the eyes in the infrared image. Using this reading we convert the temperature measurement from millikelvin to Fahrenheit before displaying it near the eyes on the registered image. We see that the subject does not have a fever (Figure 6). ▪

Learn More

Image Processing with MATLAB
mathworks.com/wbnr43666

Computer Vision with MATLAB
mathworks.com/wbnr61958

Writing Apps in MATLAB

By David Garrison, MathWorks

An app is a self-contained MATLAB® program, with a GUI, that automates a task or calculation. All the operations required to complete the task—getting data into the app, performing calculations on that data, and getting results—are performed within the app.

Starting in R2012b, you can package your own apps to share with other MATLAB users. You can also download apps written by others from the MATLAB Central File Exchange or other sources and install them in the apps gallery (Figure 1).

Writing Apps

Because an app has a GUI that a user interacts with, writing an app differs in certain respects from writing other MATLAB programs. When you write an app, you are creating an *event-driven* program. Once your app is on the screen, it typically remains idle until a user causes an event by interacting with the app—for instance, by entering text or clicking a button. In response to that action, a *callback function* is executed. That callback function, provided by the app's author, executes some code in response to the event that triggered it. For example, clicking a **Run** button might execute a callback function that performs some engineering calculations and updates a plot shown in the GUI.

In event-driven programming, each event callback is a short function that must obtain the data that it needs to do its job, update the app as necessary, and store its results where other callbacks can access them. The underlying app is essentially a collection of small functions working together to accomplish the larger task. When writing an event-driven program, you face the issues of writing the callbacks for the controls in your app and managing the information to be shared among these callbacks.

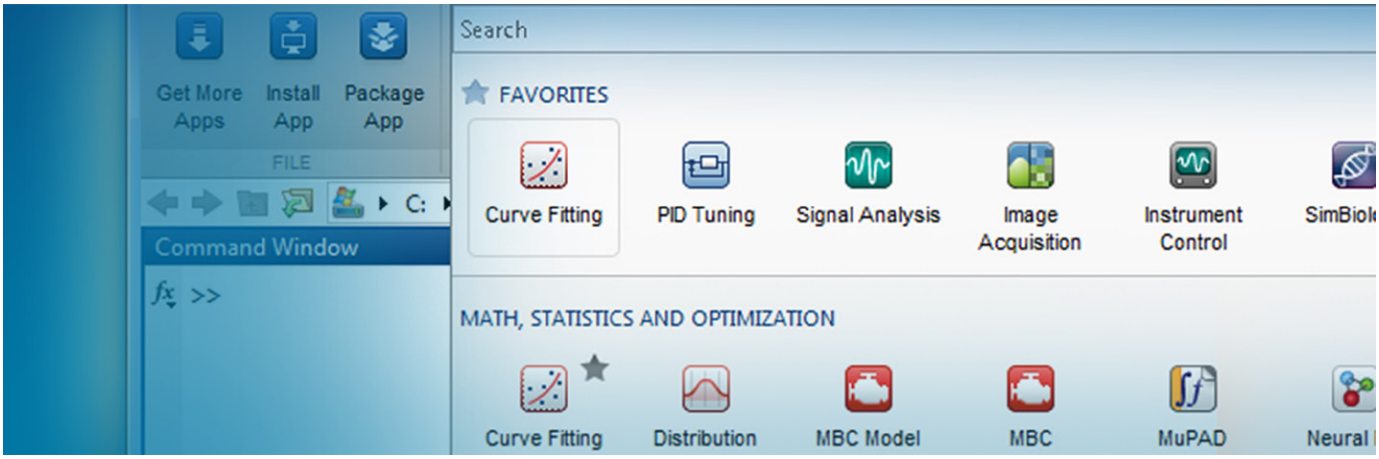
MATLAB supports two approaches to writing apps. You can:

- Write the code from scratch
- Use GUIDE, the MATLAB Graphical User Interface Design Environment

Most users find it easier to use GUIDE to graphically lay out the GUI and generate an event-driven framework for the app. Some, however, prefer the extra control they get from authoring apps from scratch. This article describes a method for writing apps from scratch using object-oriented programming. We've found this method to be an efficient way to create robust user interfaces.

Working with Objects: A Stock Ticker App Example

An object manages a collection of related functions and the data they share. Objects



The MATLAB apps gallery, accessed via the Apps tab on the MATLAB toolstrip. The Apps tab shows all the apps that you currently have installed.

are particularly useful for writing event-driven programs. Unfortunately, however, many programmers avoid using objects, either because they think they are too complicated or because they find the task of learning object-oriented programming daunting.

Don't worry. You do not need to become an expert in object-oriented programming to use objects to build an app. You just need to understand a few basic concepts.

When you create an object, you need to define two things: its list of *Properties*—the data stored within the object—and its *methods*—the functions that operate on the data stored in the properties of the object.

Let's look at a simple stock ticker app that updates a graph of stock prices over time for a given ticker symbol (Figure 2).

The `simpleStockTicker` MATLAB program creates the object that implements the app.

The first line of the program tells MATLAB that you are defining a new class (code excerpt 1, page 28).

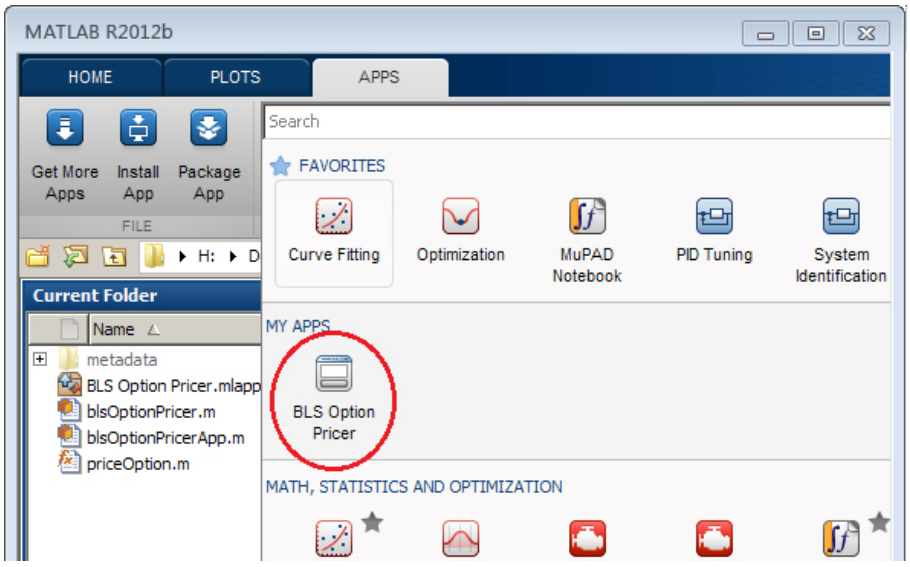


FIGURE 1. *The MATLAB apps gallery with a custom app installed in "My Apps."*

The `classdef` keyword defines a new type of object (a "class"). The name of the class, `simpleStockTicker`, must match the name of the MATLAB file. The last part of the line, `< handle`, instructs MATLAB not to make copies of this object. All your apps will start like this; only the class name (`simpleStockTicker` in our example) will change.

The properties section comes next. The properties section is defined by the `properties...end` syntax. It is where you define the data that will be used by the object (code excerpt 2).

This class uses two groups of properties. The first five properties store the handles to the visual components of the user interface—the figure, the axis, the line plotted for the prices, the 'Symbol' label, and the edit box where you can type the ticker symbol name. The last four properties store data that is used to obtain and plot the stock prices. These properties can be used by any method of the class.

Using properties helps address a common problem in authoring apps: where to store data that needs to be shared by different parts of the app. Traditionally, the most common approaches have been to use `guidata` or global variables to store shared data, but these approaches have limitations. It can be

Code Excerpts from the simpleStockTicker MATLAB Program

Code Excerpt 1

```
classdef simpleStockTicker < handle
```

Code Excerpt 2

```
properties
  % Graphics handles
  Figure
  Axis
  Line
  TickerText
  TickerEdit
  % Timer object to get updated prices
  Timer
  % In seconds
  TimerUpdateRate = 1
  % Number of values shown in the plot
  NumValues = 30
  % Current ticker symbol (initial value = 'GOOG')
  TickerSymbol = 'GOOG'
end
```

Code Excerpt 3

```
function app = simpleStockTicker
% This is the "constructor" for the class, it
  runs when an object of this class is created
  % Main figure
  app.Figure = figure('MenuBar','none',
    'NumberTitle','off','Name','Simple Stock
    Ticker','CloseRequestFcn',@app.closeApp);
  % Axis for prices
  app.Axis = axes('Parent',app.Figure,
    'Position',[.13 .15 .78 .75]);
  % 'Symbol' label
  app.TickerText = uicontrol(app.Figure,
    'Style','text','Position',[20 20 50 20],
    'String','Symbol:');
  % Symbol edit box
  app.TickerEdit = uicontrol(app.Figure,'Style',
    'edit','Position',[75 20 50 20 ],
    'String',app.TickerSymbol,'Callback',
    @app.symbolUpdateCallback);
  % Initialize prices
  prices = NaN*ones(1,app.NumValues);
  app.Line = plot(app.Axis,prices,'Marker','.',
    'LineStyle','-');
  ylabel(app.Axis,'Stock value ($)');
  set(app.Axis,'XTickLabel','');
  title(app.Axis,['Stock Price: '
    app.TickerSymbol])
  % Create timer
  app.Timer = timer;
  app.Timer.ExecutionMode = 'fixedRate';
  app.Timer.Period = app.TimerUpdateRate;
  app.Timer.TimerFcn = @app.valueUpdateCallback;
  start(app.Timer) ;
end
```

Code Excerpt 4

```
function closeApp(app,hObject,eventdata)
% This function runs when the app is closed
  try
    stop(app.Timer)
    delete(app.Timer)
  end
delete(app.Figure)
end
```

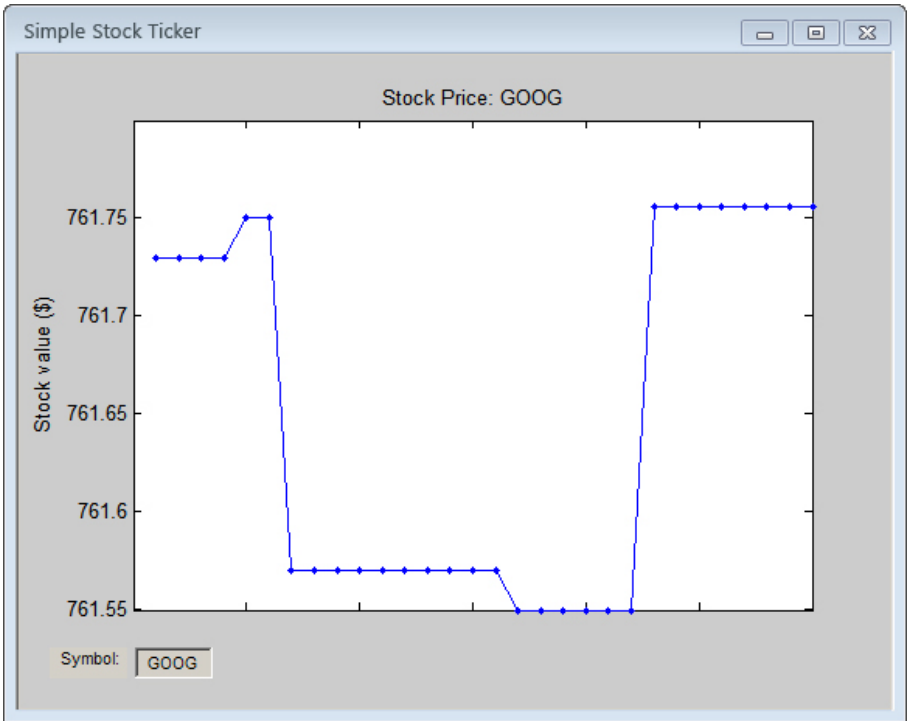


FIGURE 2. *Simple stock ticker app.*

difficult to keep `guidata` up-to-date and to access that data when the app needs it. Shared data stored as properties is easy to define and easy to access from anywhere in the app.

After defining the object's properties, you define its methods using the `methods...end` syntax. The first method, the *constructor*, is used to create the object. The name of the constructor is always the same as the name of the class (code excerpt 3).

Note that the constructor must have one output variable. The output variable is used to refer to the object created by the constructor function. You can give it any name you like. The class in our example uses the name `app`. The output variable is special in that it is used inside the class definition file to refer to the object's properties and methods. For example, you would refer to the object's `NumValues` property by using the syntax `app.NumValues`. All methods of the class are defined with this special variable as their first argument.

In our example, the constructor function performs three tasks: It creates all the visual objects in the user interface, initializes the prices to be plotted, and creates a Timer object that will update periodically to get the latest stock price. The update rate is controlled by the `app.TimerUpdateRate` property of the class.

The next three methods in this class are the callbacks. Below is the `CloseRequestFcn` callback for the figure window. It is called when the figure is closed. It looks like other callback functions you may have written, with one exception: The variable `app` must be inserted at the beginning of the argument list for the method (code excerpt 4).

Note that a class definition file can contain other methods that are not callbacks—for example, the `getQuote` method. This method is called by other methods of the class.

Advantages of Using Classes

Programmers often promote the advantages of object-oriented programming over traditional functional programming. They cite *encapsulation*, *abstraction*, and *polymorphism* as reasons for using an object-oriented approach. While these are all useful concepts, you do not need to understand them to write your app as a MATLAB class. The most important reason for using a class to create your app is that the class provides a useful way to manage data shared by different parts of your app. The properties of the object hold all the data that needs to be shared among the methods (callbacks) of your app. You no longer need to worry about using `guidata` or global variables because now all the data is stored in the properties of the class.

For more examples of apps built using a class, see the Learn More section. ▪

Learn More

Simple Stock Ticker
mathworks.com/simple-stock-ticker

IMAGEVIEWER
mathworks.com/imageviewer

Apps in MATLAB Central
mathworks.com/matlab-central-apps

The Gatlinburg and Householder Symposia

By Cleve Moler, MathWorks

Start MATLAB and issue these commands:

```
load gatlin
image(X)
colormap(map)
```

You will see a photo of the organizing committee for the 1964 Gatlinburg Symposium on Numerical Algebra (Figure 1). If you have been following my blog, you will know that I have written a great deal about the men on this committee. Here, I want to describe the conferences themselves. They have played an important role in the history of MATLAB® and in my own professional life.

In 1961, Gatlinburg was a quiet resort town near the Great Smoky Mountains National Park in eastern Tennessee. Alston Householder, who was in charge of computing at Oak Ridge National Laboratory (ORNL), chose the Mountain View Hotel in Gatlinburg as the site of a week-long research workshop on matrix computations (Figure 2). The town was easy to reach but far from the distractions of a big city. The hotel had excellent facilities for the kind of workshop he had in mind.

In a paper published in *SIAM Review*, Householder recalled that the suggestion for holding such a meeting had been made a year earlier at the Old German restaurant and pub in Ann Arbor during a University of Michigan summer school session on numerical analysis. Householder formed an organizing committee, which invited about 75 participants. Funding was obtained from the National Science Foundation and the Atomic Energy Commission to support attendees who could not pay their own expenses.

The format for the meeting, repeated in later years, consisted of two or three invited talks in the morning, two or three in the afternoon, and several impromptu, small sessions that ran simultaneously in the evening. Everyone ate in the hotel, and efforts were made to vary the groups of people seated together at each meal.

In 1963, a second meeting was held in Gatlinburg. This one, which focused on approximation theory, was chaired by Frank Olver. The organizers did not know at the time that this meeting was the second in a series, and it wasn't until some years later that it became known as Gatlinburg II.

My First Meeting

In 1964, a meeting later known as Gatlinburg III was held at the Mountain View Hotel. All six members of the committee—Jim Wilkinson, Wallace Givens, George Forsythe, Alston Householder, Peter Henrici, and Fritz Bauer—had been at the Old German that evening in 1960. I saved the 1964 photo of the group that is now distributed with MATLAB. At the time, I was a Stanford graduate student working for Forsythe. Students had not been invited to previous Gatlinburg meetings. In what had apparently been a contentious move within the committee, Forsythe obtained an invitation for me.

FIGURE 1. *The organizing committee for Gatlinburg III, 1964.*

I was the only student at Gatlinburg III. It was enormously important for me to meet the leaders in the field where I would spend most of my professional life.

There have been 16 Gatlinburg meetings since 1964, and I have been to all but one. Originally, the meetings were criticized because attendance was by invitation only from a self-appointed organizing committee. Today, the meetings are still by invitation, but there is now an open application process, and graduate students and other young people are encouraged to apply.

Gatlinburg IV, held in 1969, was the last of the series to take place in Gatlinburg. Householder retired from ORNL and from the conference organizing committee. (Coincidentally, in 1966, I became an assistant professor at the University of Michigan and organizer of the summer school. So it fell to me to invite Householder, Wilkinson, and the others to Ann Arbor, and thus, back to the Old German.)

The Gatlinburg Symposia were on numerical algebra, a topic general enough to allow attendees latitude to talk about their own research. The titles and speakers for some of the presentations at

FIGURE 2. *Mountain View Hotel, Gatlinburg, Tennessee, site of the first four conferences, 1961–1969.*

FIGURE 3. *Dodge Chapel, built in 1915 on the Asilomar Conference Grounds in Pacific Grove, California. Image courtesy ARAMARK Parks and Destinations.*

Gatlinburg IV provide a sense of the range of topics covered:

- "Some Basic Results in Automatic Solution of Polynomial Equations," A. M. Ostrowski
- "Alternating Direction Methods for Galerkin Approximations for Parabolic Problems," J. Douglas
- "Another Algorithm for Minimizing a Sum of Squares of Nonlinear Functions," M. J. D. Powell
- "Natural Norms in Algebraic Processes," V. N. Faddeeva
- "The Ax = λBx and Related Problems," J. H. Wilkinson
- "Bounds for the Abscissa of Stability of a Stable Polynomial," P. Henrici
- "The Method of Odd/Even Reduction with Application to Poisson's Equation," G. H. Golub

Vera Faddeeva was from the Steklov Institute in what was then called Leningrad. She made the trip at a time when it was not easy for Russians to visit the United States, and was one of several attendees from Eastern Europe. The symposia have always had a rich mix of international participants.

I gave short evening talks on the L-shaped membrane at both Gatlinburg III and IV. Gatlinburg III took place in 1964, when I was still working on my thesis, so the talk was about finite difference methods for the membrane eigenvalues. My thesis, and that talk, used the L-shaped membrane as the archetypical example of a region where the singularity at the 270^0 corner causes slow convergence of the finite difference eigenvalues.

Gatlinburg IV was in 1969. By then, finite difference methods had been replaced by vastly superior methods that employed expansions in fractional order Bessel functions. These had been described in a 1967 paper by Fox, Henrici, and Moler, but in my Gatlinburg IV talk I was able to describe improvements to the original computational techniques. We use these improved techniques to compute the MathWorks logo today.

A Change of Name and a Change of Venue

The fifth meeting was held in 1972 at Los Alamos National Laboratory in New Mexico, with Richard Varga as the chairman. The present-day pattern for the series began to emerge. The conference is held roughly every three years in venues alternating between Europe and North America. Members of the organizing committee serve limited terms, and the committee itself appoints new members. While the conferences are not formally SIAM activities, SIAM provides logistic and organizational support. At some point, the name of the series changed from the Gatlinburg to the Householder Symposia.

A particularly proud moment in my professional life occurred at Householder V in Los Alamos. For some time before the meeting, Pete Stewart and I had been working on the development of the QZ algorithm for the generalized matrix eigenvalue problem $Ax = \lambda Bx$. The theoretical foundation of the algorithm was Pete's idea, but we had worked together on the algorithmic aspects and the computer code. Pete was invited to give a talk about our work. Afterwards, Jim Wilkinson stood up and said, "I wish I had thought of that." It was a wonderful moment.

In 1974, Householder VI was organized by Bauer at the Kurhotel Enzensberg in Hopfen am See, at the foot of the Bavarian Alps. This was the first European meeting in the series.

In 1977, Householder VII, organized by Gene Golub, was held at Asilomar, a spectacular California state park and conference center on scenic 17-Mile Drive on the Monterey Peninsula. The lectures took place in the historic chapel (Figure 3).

Introducing MATLAB

In 1981, Householder VIII, organized by Leslie Fox and Jim Wilkinson, was held in Oxford (Figure 4). (The city of Oxford certainly violates the rule of being isolated from other distractions.) This meeting marks the beginning of the close association between the Householder

FIGURE 4. *The participants in Householder VIII, Oxford, England, 1981.*

Symposia and MATLAB that continues to this day.

It was before MathWorks existed. I had installed the code for classic Fortran MATLAB on the mainframe in the campus computer center, and provided show-and-tell from a remote terminal during the coffee breaks and at one of the evening sessions. Some attendees were interested in using the code at their own universities, and so I sent them copies. We can trace many of the early academic adoptions of MATLAB to people who were at this meeting.

In 1984, Householder IX was organized by Alan George at the University of Waterloo. At the time of this meeting, the IBM PC had been available for over a year. In California, Jack Little was already at work on the MathWorks version of MATLAB, but we could not show that yet. I had ported my Fortran code to the PC. I borrowed a PC AT from Alan, and again, set up near the coffee area. This time, instead of a mag tape with source code, I could promise that a powerful new professional version of MATLAB was on its way.

Figure 5 shows a recreation of what appeared when you started up that classic Fortran MATLAB compiled for Householder IX. There were only 77 functions and keywords. That's all. There were no M-files, no toolboxes, and only very crude graphics. If you wanted to add new functions, you had to write Fortran subroutines and modify the source code. Basically, MATLAB was just a simple matrix calculator.

In 1987, Householder X was organized by Bob Ward and Pete Stewart and held in Fairfield Glade, Tennessee. Unfortunately, I couldn't attend. This conference was the only one in the series that I've ever missed. It would have been the first at which I could have shown the MathWorks version of MATLAB.

In 1990, Householder XI was organized by Åke Björck and held in Tylösand, on the west coast of Sweden, about 100 miles north of Copenhagen. In an evening talk, I gave a preview of the capabilities for sparse matrices in MATLAB that John Gilbert, Rob Schreiber, and I were then developing. Many of the symposia participants were involved in research on the numerical solution of partial differential equations, and were particularly interested in sparse matrices. We did not have many powerful tools to demonstrate—those would come later—but the idea of having sparse matrices in MATLAB was enthusiastically received.

```
                    < M A T L A B >
            Gatlinburg IX Version 6/03/1984

HELP is available

<>
help
 Type HELP followed by
 INTRO    (To get started)
 NEWS     (recent revisions)

 ABS   ANS   ATAN  BASE  CHAR  CHOL  CHOP  CLEA
 COND  CONJ  COS   DET   DIAG  DIAR  DISP  EDIT
 EIG   ELSE  END   EPS   EXEC  EXIT  EXP   EYE
 FILE  FLOP  FLPS  FOR   FUN   HESS  HILB  IF
 IMAG  INV   KRON  LINE  LOAD  LOG   LONG  LU
 MACR  MAGI  NORM  ONES  ORTH  PINV  PLOT  POLY
 PRIN  PROD  QR    RAND  RANK  RCON  RAT   REAL
 RETU  RREF  ROOT  ROUN  SAVE  SCHU  SHOR  SEMI
 SIN   SIZE  SQRT  STOP  SUM   SVD   TRIL  TRIU
 USER  WHAT  WHIL  WHO   WHY
 < > ( ) = . , ; \ / ` + - * :

<>
```

FIGURE 5. *The initial screen of classic Fortran MATLAB.*

The Gatlinburg and Householder Symposia, 1961–2014

I	1961	Gatlinburg, Tennessee	USA
II	1963	Gatlinburg, Tennessee	USA
III	1964	Gatlinburg, Tennessee	USA
IV	1969	Gatlinburg, Tennessee	USA
V	1972	Los Alamos, New Mexico	USA
VI	1974	Hopfen am See	Germany
VII	1977	Asilomar, California	USA
VIII	1981	Oxford	England
IX	1984	Waterloo	Canada
X	1987	Fairfield Glade, Tennessee	USA
XI	1990	Tylösand	Sweden
XII	1993	Lake Arrowhead, California	USA
XIII	1996	Pontresina	Switzerland
XIV	1999	Whistler, British Columbia	Canada
XV	2002	Peebles	Scotland
XVI	2005	Seven Springs, Pennsylvania	USA
XVII	2008	Zeuthen	Germany
XVIII	2011	Lake Tahoe, California	USA
XIX	2014	Spa	Belgium

In 1993, Householder XII was organized by Tony Chan and Gene Golub and held at the UCLA Lake Arrowhead Conference Center in the San Bernardino Mountains, east of Los Angeles. When I demonstrated the latest features of MATLAB at one of the evening sessions, I had some valuable help. Eleven-year-old Tor Bjørstad had come to Lake Arrowhead with his father, Petter, a professor at the University of Bergen in Norway. MATLAB was available on the Bjørstad family computer because Petter had been a fan since his grad student days at Stanford. Tor was a proficient MATLAB user, and so I asked him to operate the computer while I gave the talk. We were listed as co-presenters on the day's program.

In 1996, 1999, and 2002, there were Householder conferences in Pontresina, Switzerland; Whistler, British Columbia; and Peebles, Scotland, respectively. By this time it would have been presumptuous to give coffee-break demonstrations of MATLAB because almost all the attendees were using MATLAB themselves. Or, at least, their students were. Most of the talks included computational results presented in MATLAB plots. Today, I see MATLAB plots at every conference I attend, but they are more frequent at Householder meetings than at any other.

Announcing Parallel MATLAB

In 2005, Householder XVI was organized by Jesse Barlow and colleagues. It was held at Seven Springs, Pennsylvania, a resort and conference center in the foothills of the Appalachian Mountains, an hour southeast of Pittsburgh. This meeting was particularly significant for me because I made the announcement of parallel MATLAB. Ten years earlier, I had written a one-page Cleve's Corner article entitled "Why There Isn't a Parallel MATLAB." There were three primary reasons:

- Memory model: Matrices are inconsistent with distributed memory.
- Granularity: Efficiency requires large parallel tasks.
- Business situation: The number of users with parallel machines is too small.

This article became one of my most widely cited papers.

My talk at Householder XVI showed how the world of technical computing, and MATLAB, had changed in 10 years. The introduction of distributed arrays provided a much improved memory model. The introduction of the `parfor` construct provided coarser granularity. By 2005, many MATLAB users had access to parallel computing environments and were anxiously awaiting parallel MATLAB.

In 2008 and 2011, there were Householder conferences in Zeuthen, Germany, and Lake Tahoe, California. In 2014, Householder XIX is being organized by Ilse Ipsen and Paul Van Dooren. It will be held at Domain Sol Cress near the town of Spa in eastern Belgium. ▪

Further Reading

Alston S. Householder, "The Gatlinburgs," *SIAM Review* 16-3, 1974, pp. 340–343, epubs.siam.org/doi/abs/10.1137/1016054.

Josef Stoer, "The Gatlinburg Symposia and Their Influence on the Development of Numerical Linear Algebra," *IMAGE*, The Bulletin of the International Linear Algebra Society 46, 2011, pp. 13–25, ilasic.org/IMAGE/IMAGES/image46.pdf.

Learn More

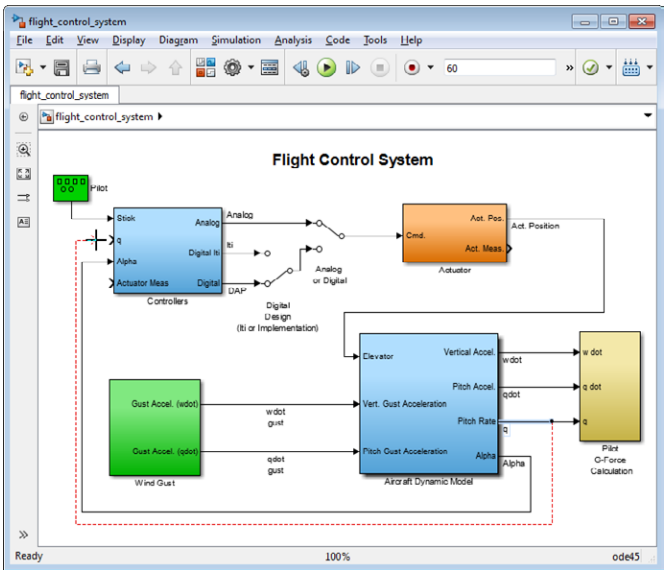
Householder Symposium XIX
sites.uclouvain.be/HHXIX

Cleve's Corner Blog
blogs.mathworks.com/cleve

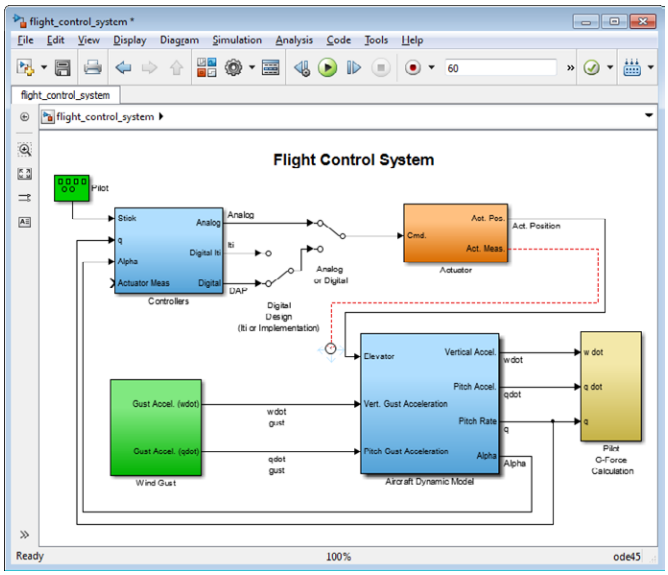
Cleve's Corner Collection
mathworks.com/cleves-corner

Smart Signal Routing in Simulink

With smart signal routing, Simulink® automatically finds the shortest path that has the fewest 90-degree turns, without overlapping other blocks or text. Not only that, but as you draw the signal line, Simulink shows you exactly what the path will look like before you release the mouse button. You get real-time feedback, and there are no surprises.



Simulink model with smart signal routing enabled. Smart signal routing automatically finds a "smart" path from block to block.



Custom signal path created using smart signal routing. Custom signal paths give you precise control over the shape of the signal line.

Custom Signal Paths

Smart signal routing has significantly improved the automatic signal routing algorithms used in Simulink; however, there are still times when you want to draw a signal path yourself so as to precisely determine the shape of the line. Smart signal routing can help here, as well. This is how it works: Draw a signal line. To create a bend in the line, release the mouse button. Three blue arrows appear, one pointing straight ahead, one left, and one right. Choose the direction that you want the signal line to follow, and repeat the process until you've completed the path.

Minimum Disturbance

Part of the "smartness" of the smart signal routing algorithm is that it operates on the principle of minimum disturbance—that is, it changes the shape of existing signal lines as little as possible as you make changes to the model. This is especially important when you have taken the time to create a custom signal path; you do not want that entire path to get redrawn when another block is moved.

In some cases, tiny kinks can appear in the signal line, especially if the model was created in Simulink R2012b or earlier. In those instances, simply move the connected blocks. This automatically removes the kinks, giving you a cleaner-looking model. ▪

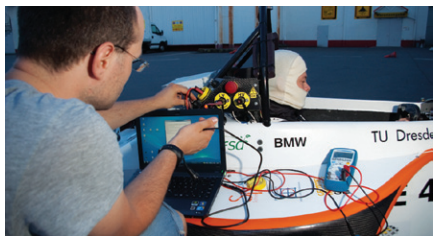
Learn More

Video: Smart Signal Routing 1:36
mathworks.com/smart-signal-routing

Guy and Seth on Simulink
blogs.mathworks.com/seth

Student Competitions: Project-Based Learning in Action

Competitions support project-based learning by inviting students to think like engineers. Each competition presents a unique engineering challenge that may require months, or even years, of intense focus and hard work. Using industry-standard tools such as MATLAB® and Simulink®, and low-cost hardware such as Arduino®, Beagleboard, LEGO® MINDSTORMS®, and Raspberry Pi™, students tackle real engineering problems. They also acquire the collaboration, project-management, and leadership skills they will need for careers in industry.



Left to right: TU Dresden Robotik AG (TURAG) team winning second place in the Eurobot German Nationals, TU Dresden Elbflorace team working on a Formula Student racecar, Ohio State University team with the Buckeye Bullet 2.

In the Students' Own Words

"Our primary aim is the yearly participation in the Eurobot competition with our own individually designed and manufactured robot. Building an autonomously acting robot requires a huge amount of complex calculation, for which we use MATLAB and Simulink. Using these tools we developed an A-algorithm-based routing process for the robot and analyzed data measured by laser distance sensors."* —Richard Rudat, TU Dresden Robotik AG (TURAG) team

"At Mississippi State University, I used MATLAB and Simulink extensively to develop hybrid vehicle system control algorithms and model the various components that make up a hybrid vehicle. These tools allowed me to rapidly develop and validate highly sophisticated vehicle control algorithms. By putting these control algorithms to use in the EcoCAR competition, I was able to find success in a highly diverse team environment, which in turn landed me my job at Argonne National Laboratory." —Brian Benoy, EcoCAR2

Competitions Sponsored by MathWorks

MathWorks equips student teams with software, training, and mentoring in competitions worldwide.

Automotive: American Solar Challenge, EcoCAR2, Formula SAE Competition of Japan, Smart Car Contest, EducEco, Formula Student Germany

Robotics: BEST Robotics, FIRST Robotics, ET ROBOCON, ROBO-ONE, RoboCup

Aerospace: AUVSI RoboBoat and RoboSub, The International Micro Air Vehicle Flight Competition, UAV Challenge – Outback Rescue

Programming and Math: Contemporary Undergraduate Mathematical Contest in Modeling, Cornell Cup

Biotech: iGEM

Developing Advanced Controls for the Buckeye Bullet

A battery-powered landspeed streamliner built by students at Ohio State University's Center for Automotive Research set a new international electric vehicle record. Competing in the FIA Landspeed Record for Electric and Battery Powered vehicles, held each August at Bonneville Salt Flats in Utah, the Buckeye Bullet 2 reached a speed of 307 miles per hour, making it the fastest vehicle on the planet.

The OSU team's current vehicle, Buckeye Bullet 3, is expected to achieve 400 mph. When a vehicle covers 1 mile every 9 seconds and takes only 6 miles to reach its top speed, a control failure would be catastrophic. The team uses Simulink system models and runs multiple simulations to predict vehicle performance and test new control systems.

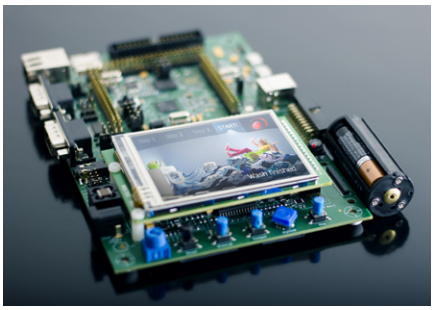
Learn More

Student Competitions
mathworks.com/student-competitions

Hardware for Project-Based Learning
mathworks.com/project-hardware

Embedded HMI Development with Model-Based Design

Third-party products help developers prototype and deploy human machine interface (HMI) applications with integrated graphics and multitouch and voice input. By using these products with Simulink® and Stateflow®, developers can design and verify the underlying logic and test HMI concepts with models of system components or real-world conditions. Developers can deploy designs onto embedded platforms, integrating code generated from Simulink models with graphics code to create HMIs for vehicle dashboards, avionics simulators, test equipment, and other applications.



Altia Design and DeepScreen

With Altia® Design, engineers can design and simulate HMI prototypes, including graphics objects, animations, and stimulus inputs, without writing code. The Altia Connector enables a Simulink block to include control logic from Stateflow, resulting in functionally complete, accurate models. With Altia DeepScreen®, HMI prototypes can be converted into deployable graphics code targeting embedded real-time operating systems (RTOS) and graphics libraries. Generated HMI code can be combined with code for control logic generated from Stateflow functional models.
Altia *www.altia.com*

GL Studio

GL Studio® helps developers create high-fidelity graphics and interactive software controls, including 2D and 3D interface elements for automotive, aviation, and industrial displays. GL Studio generates C++ code for deploying HMIs to embedded environments, including Windows®, Linux®, and iOS. DiSTI also offers libraries of graphical elements, cockpits, dashboard instruments, and a package for certifiable safety-critical displays. The Simulink interface includes a block that can be added to models to enable drag-and-drop mapping of HMI class properties to Simulink elements without coding.
DiSTI Corporation *www.disti.com*

EB GUIDE

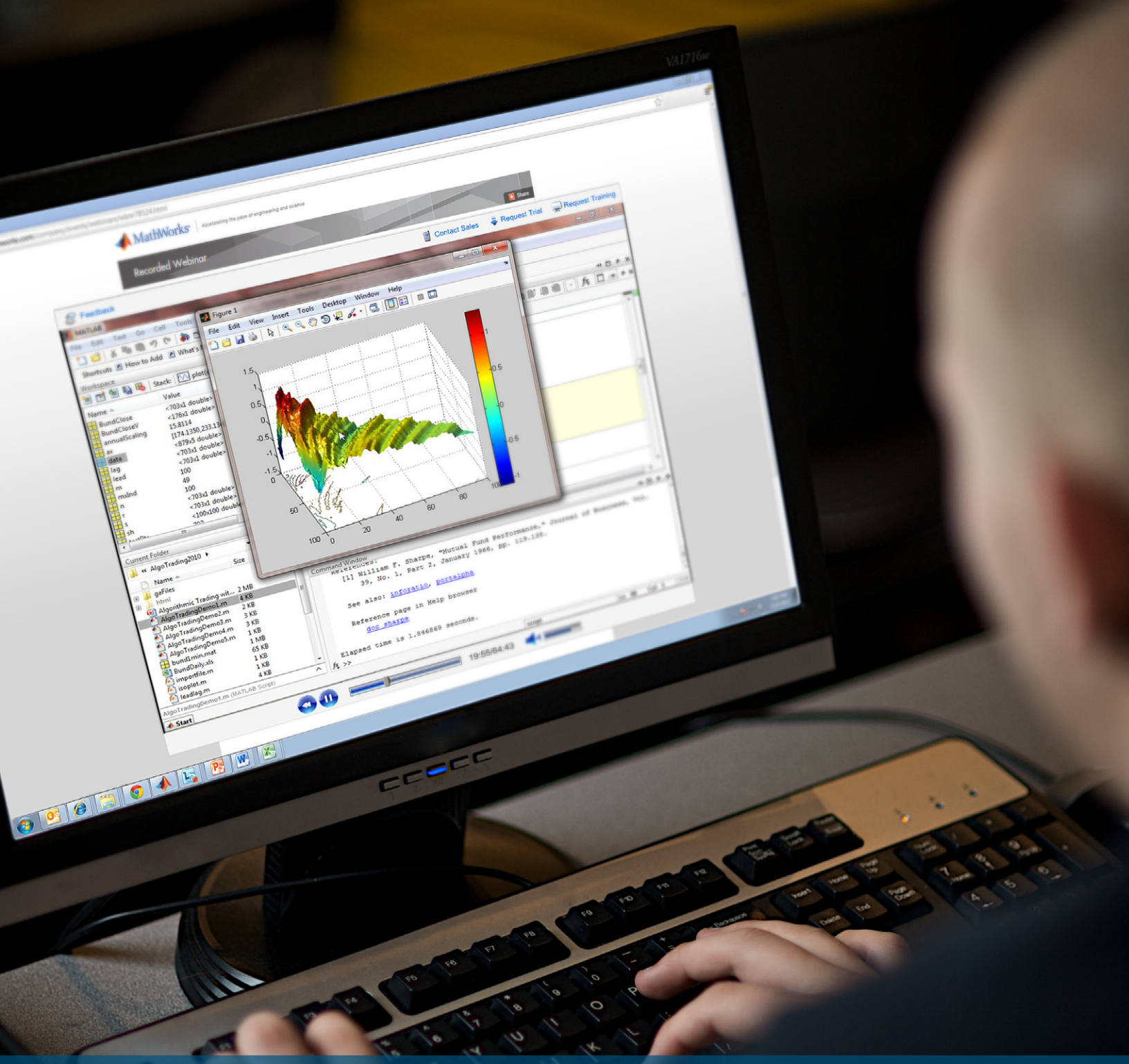
Engineers use EB GUIDE to develop user interfaces with 3D elements, animations, and multimodal inputs. The HMI designer graphically defines menus and underlying logic and adds customizable skins, multitouch support, and multilanguage support, as well as voice recognition and audio output. The EB GUIDE interface to Simulink enables designers to define shared data items and cosimulate HMIs with models of related subsystems and real-world conditions. EB GUIDE offers a run-time environment that supports RTOS and embedded hardware platforms, enabling OEMs to update automotive applications without recompilation.
Elektrobit Automotive *automotive.elektrobit.com*

VAPS XT

VAPS XT™ lets developers design interactive graphical interfaces for safety-critical avionics and embedded displays. Developers define the look of visual elements, assign behavior, and generate a standalone executable of the application. Optional packages support ARINC 661 compliant widgets and DO-178B certification. The developer can combine code generated from Simulink models with VAPS HMIs into a single, embedded executable. VAPS includes code generation support for COTS hardware platforms, embedded operating systems, and OpenGL variants. **Presagis** *www.presagis.com*

Learn More

Third-Party Products and Services
mathworks.com/connections



Explore MATLAB and Simulink Webinars

Join a live session or browse the library of commercial and academic recordings.

mathworks.com/webinars

Technical experts present practical tips and examples. Topics include:

- Plant modeling and control design
- Embedded code generation
- Rapid prototyping and HIL simulation
- Signal processing and communications
- Physical modeling
- Verification, validation, and test
- Image and video processing
- Test and measurement
- Math, statistics, and optimization
- Algorithm development
- GPU and parallel computing
- Desktop and web deployment

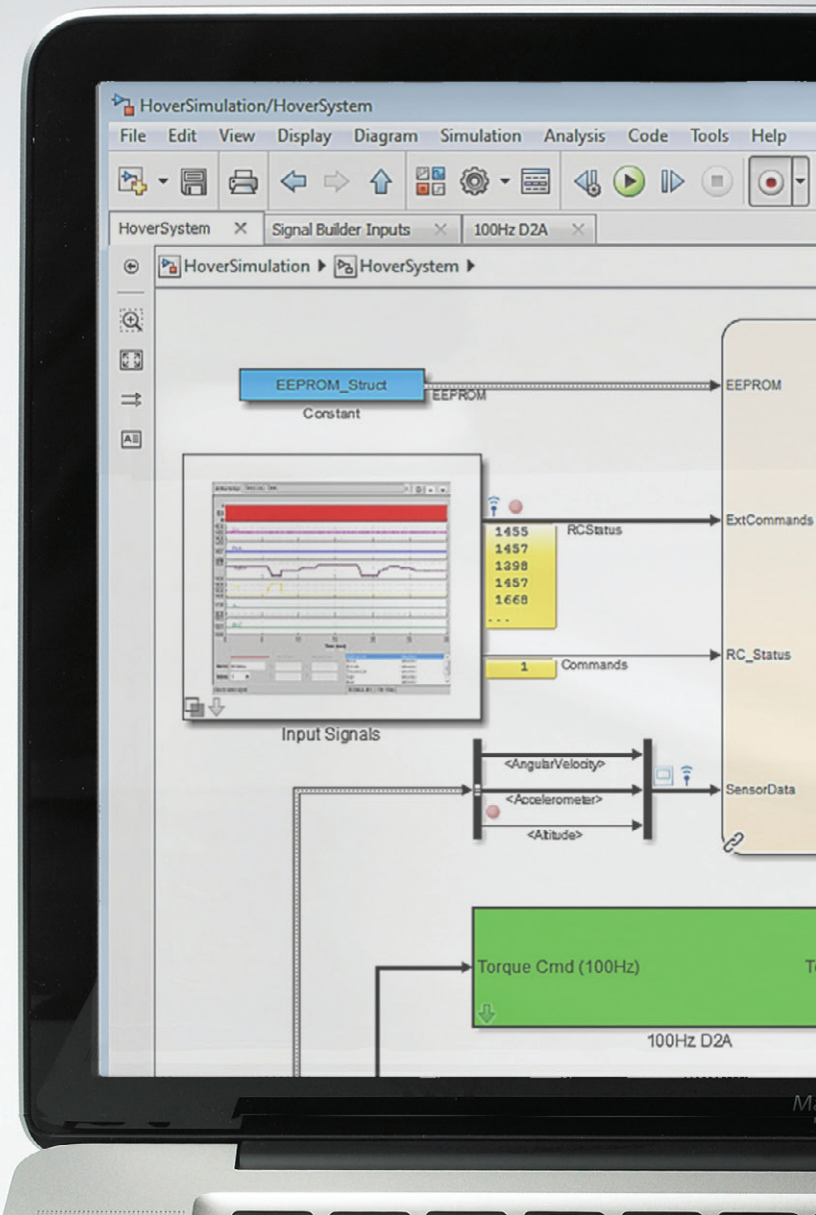
MathWorks®
Accelerating the pace of engineering and science

MODELING AND SIMULATION MADE EASY

with

Simulink

WATCH VIDEO 1:47
mathworks.com/simulink-editor



It's even easier to build, manage, and navigate your Simulink® and Stateflow® models using the new Simulink Editor.

- Smart line routing
- Tabbed model windows
- Simulation rewind
- Signal breakpoints
- Explorer bar
- Subsystem and signal badges
- Project management

MATLAB® & SIMULINK®

92126v00 10/13