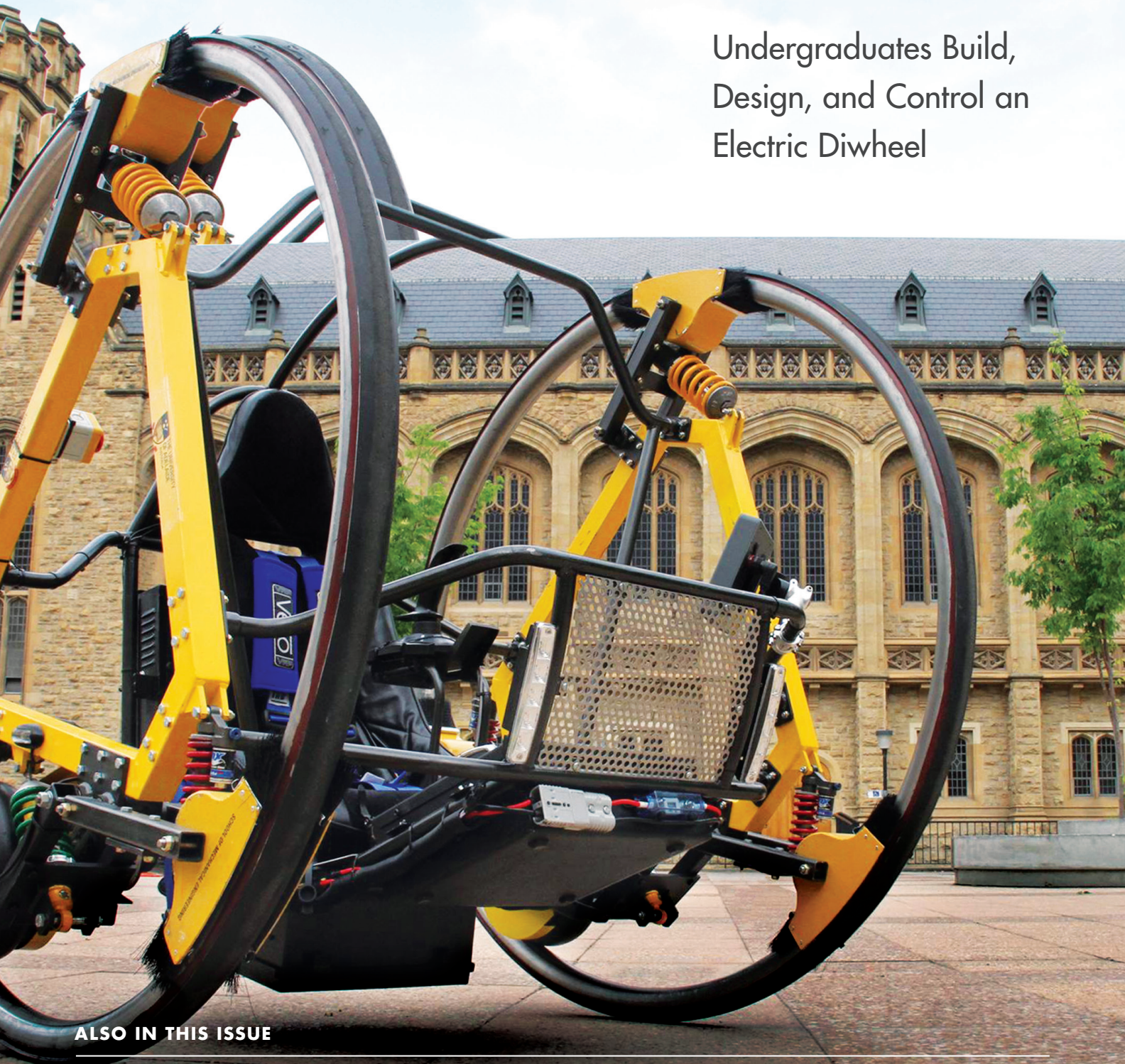
# MathWorks News&Notes

The Magazine for the MATLAB® and Simulink® Community

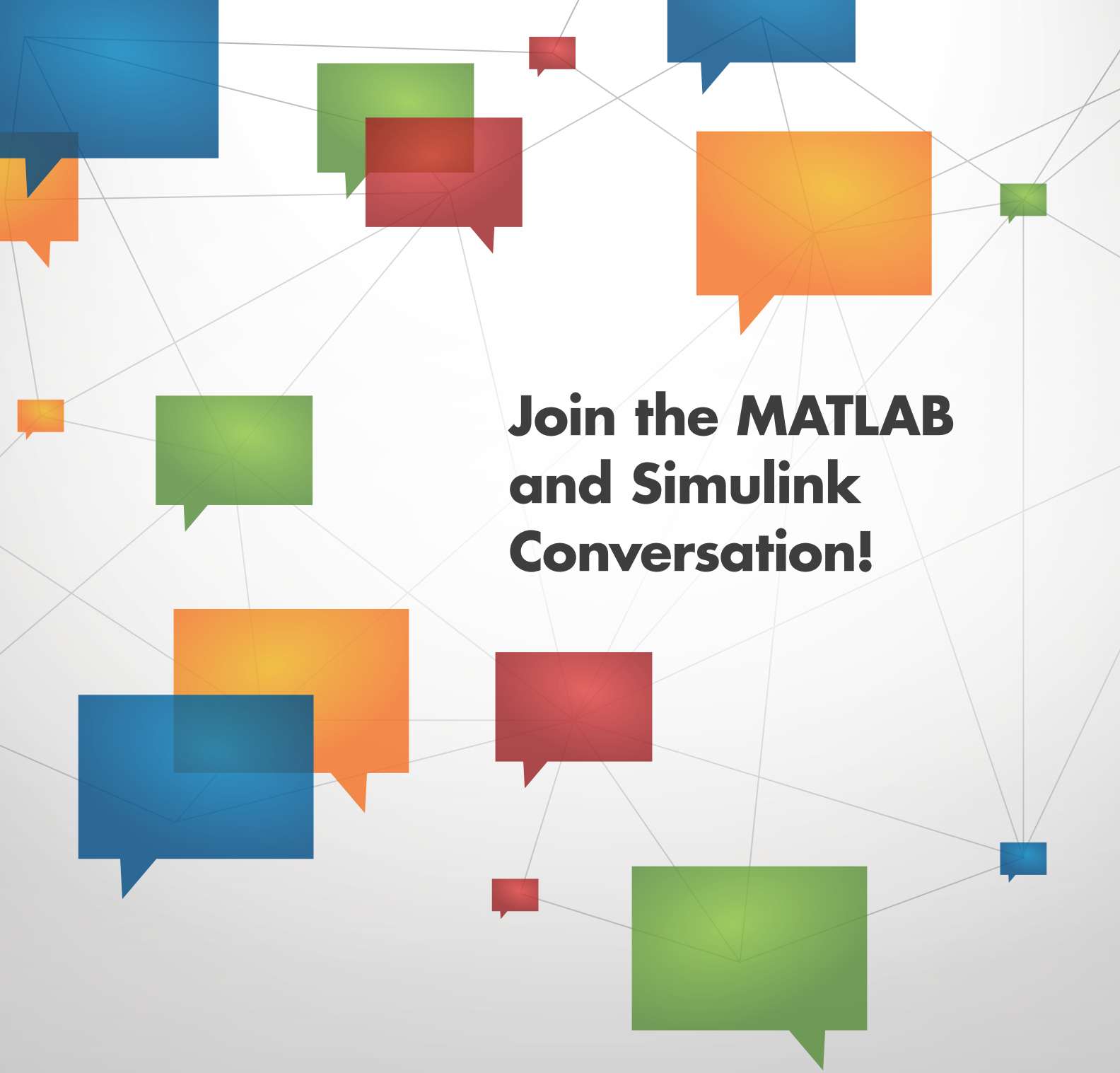
## Undergraduates Build, Design, and Control an Electric Diwheel



**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.

**Trendy** Track and plot trends using a MATLAB based web service.

**Link Exchange** Share links to videos, tutorials, examples, and other Internet resources.

**Newsgroups** Connect with community members.

**Blogs** Read commentary from engineers who design, build, and support MATLAB and Simulink.

**Contests** Compete in semiannual programming challenges.

mathworks.com/matlabcentral

MATLAB® CENTRAL

*An open exchange for the MATLAB and Simulink user community*

©2012 MathWorks, Inc.

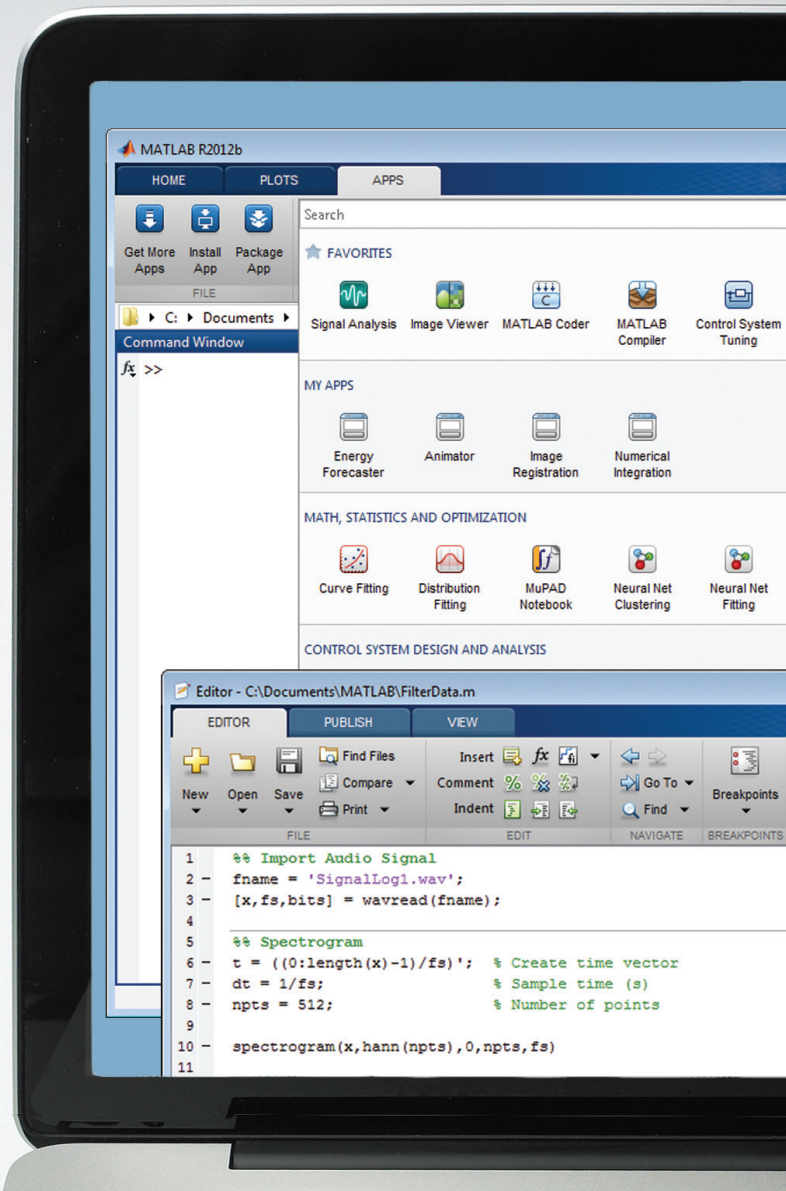
MathWorks®
*Accelerating the pace of engineering and science*

*The New*

# MATLAB Desktop

## See what you've been missing.

TRY IT TODAY
visit mathworks.com/matlab-new-features



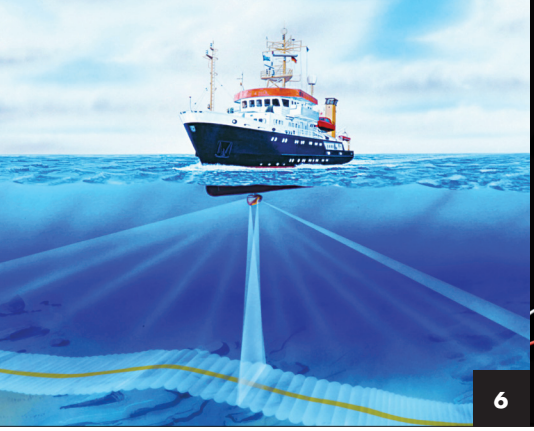

R2012b introduces a fresh new MATLAB® Desktop, making it easier to find what you need.

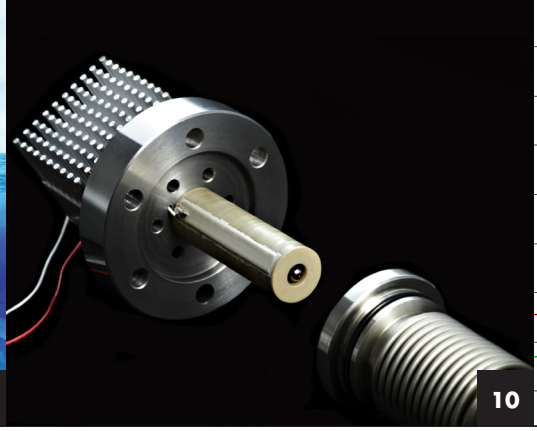
**Toolstrip**
Highlights commonly used functionality

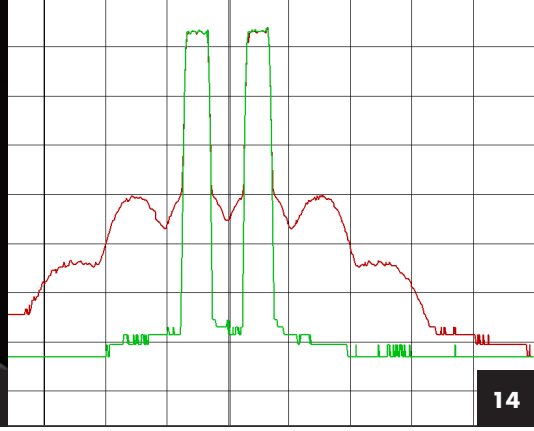
**Apps Gallery**
Displays in-product and user-written apps

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

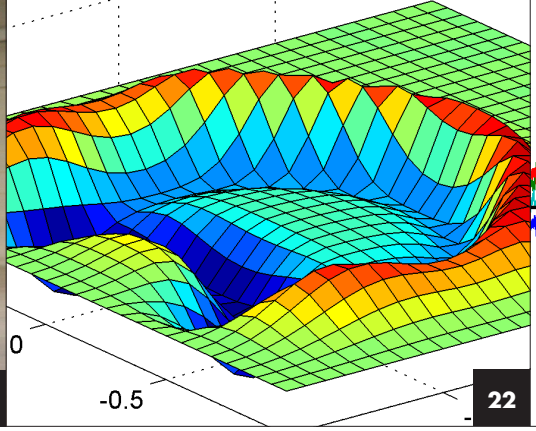
MATLAB® & SIMULINK®

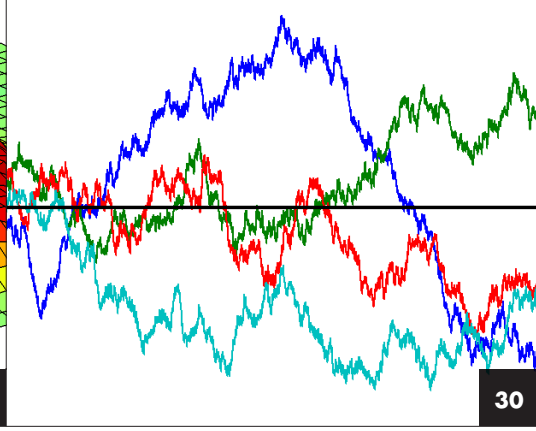



## FEATURES


## DEPARTMENTS



## ABOUT THE COVER

The cover shows an electric diwheel, designed and built in Simulink by mechanical engineering students at the University of Adelaide. The diwheel software incorporates slosh control, which keeps the frame stable in aggressive braking or acceleration maneuvers, and a unique swing-up and inversion controller that lets the rider drive the vehicle when upside down.


**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. Carlson, L. Goodman, L. Heske, L. Macdonald, N. Perez



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

**Contributors and Reviewers**
R. Aberg, C. Aden, M. Agostini, H. Atzrodt, J. Augustin, P. Barnard, D. Basilone, G. Bourdon, M. Carone, B. Cazzolato, A. Chakravarti, H. Chen, K. Cohan, D. Cook, S. Craig, L. Dean, G. Dudgeon, K. Dunstan, E. Ellis, N. Fernandes, A. Frederick, D. Garrison, C. Garvin, T. Gaudette, J. Ghidella, S. Grad-Freilich, E. Johnson, C. Kalluri, M. Katz, L. Kempler, S. Kozola, T. Kush, T. Lennon, K. Lorenc, S. Mahapatra, J. Martin, S. Miller, A. Mulpur, S. Oliver, S. Popinchalk, J. Reese, M. Ricci, G. Rouleau, R. Rovner, G. Sandmann, G. Schreiner, V. Shapiro, G. Sharma, R. Shenoy, N. Stefansson, A. Stevens-Jones, L. Tabolinsky, B. Tannenbaum, G. Vella-Coleiro, J. Wendlandt, S. Wilcockson, R. Willey, M. Woodward, M. Yeddanapudi, S. Zaranek



**Subscribe**
mathworks.com/subscribe

**Comments**
mathworks.com/contact-us


MathWorks®


©2012 The MathWorks, Inc.
MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.t

# Financial Services

Financial professionals from more than 2300 organizations worldwide use MATLAB® to develop and implement financial models, analyze substantial volumes of data, and operate under tightening regulation. With MATLAB they measure and manage risk, construct portfolios, trade at low and high frequencies, value complex instruments, and construct asset-liability models.


### A2A SPA

**Building commodity risk management platforms: analyzing multiple risk factors**

A2A is one of the largest utility companies in Italy. The company's risk management unit facilitates and monitors daily trading activities and supports longer-term strategy-setting. A2A's risk management platform, built using MATLAB and related toolboxes, enables analysts to gather historical and current market data, apply sophisticated nonlinear models, rapidly calculate more than 500 risk factors, perform Monte Carlo simulations, and quantify value at risk (VaR). With MATLAB Compiler™ and MATLAB Builder™ NE, A2A deploys dashboard-driven analytics that help analysts and traders visualize results, manage risk, and record contract information.

***mathworks.com/a2a***

### LIQUIDNET

**Increasing model scale: monitoring transaction performance on an exchange**

To reduce the market impact of large trades, institutional equity traders turn to alternative trading systems, such as Liquidnet, in which trades are executed anonymously. To measure execution performance, Liquidnet must compare the execution price of the traded equity with price trends preceding and following the transaction. Using MATLAB, Liquidnet built an automated system capable of analyzing all orders executed every day on the Liquidnet platform. The system compares transaction data with market trends and measures the performance of trades within a short-term time scale.

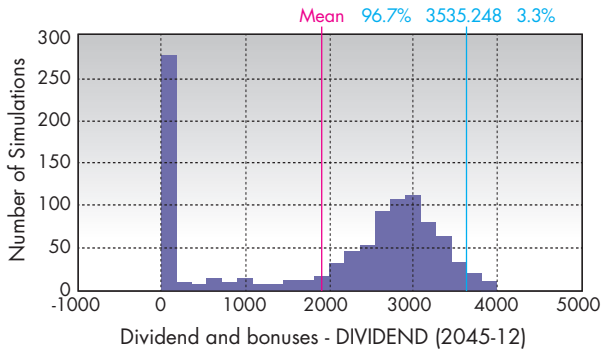
***mathworks.com/liquidnet***

### NYKREDIT ASSET MANAGEMENT

**Deploying reliable performance analytics: calculating and visualizing risk statistics**

With a strong emphasis on balanced risk management, the Analytic Support Unit within Nykredit Asset Management provides quantitative support for the division's fund managers and analysts. The Analytic Support Unit provides reports and tools that drive asset allocation, corporate bond profiling, performance metrics, and risk analyses. The unit uses MATLAB and related toolboxes to rapidly prototype algorithms, access information in databases and legacy C++ applications, visualize results, and implement operational dashboards that portfolio managers can use without formal training or knowledge of the underlying algorithms.

***mathworks.com/nykredit***



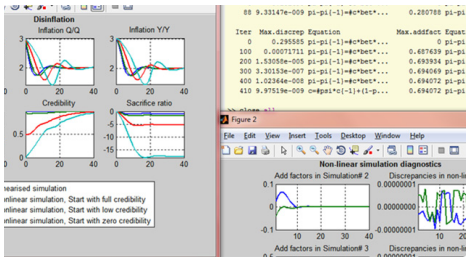

### MODEL IT

**Complying with insurance regulations: determining solvency, assessing insurance risk, and identifying embedded value**

Europe's Solvency II and Own Risk Solvency and Assessment (ORSA) frameworks require insurers to model and price insurance

contracts to the most granular level of detail. This requirement is particularly challenging in life insurance, where policies include many possible capital market relations and optionalities. Model IT used the MATLAB based mSII Toolbox to develop a contract-level simulation methodology to simplify the problem and minimize model risk. This methodology enabled a Finnish life insurer to develop a Pillar 1 Market Consistent Embedded Value (MCEV) and Solvency Capital Requirement (SCR) calculation environment and run authority reporting tests according to Eiopa's Quantitative Impact Study QIS5 guidelines.

***mathworks.com/model-it***

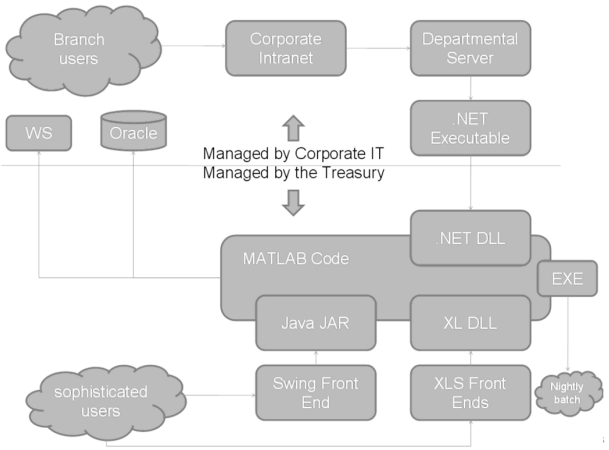



## IMF

### Designing modern forecasting and policy analysis systems: collaborating with economist peers

Many governments and their central banks base monetary policy on inflation targeting. Effective inflation targeting requires forecasting and policy analysis systems that model inflation, output, and other macroeconomic variables. Jaromir Benes at the International Monetary Fund (IMF) employs MATLAB and the MATLAB based, open-source IRIS Toolbox to perform dynamic stochastic general equilibrium (DSGE) modeling, multivariate time-series and macroeconomic time-series management, and PDF and LaTeX-based reporting. Economists can use these same tools to investigate components of macroeconomic analysis; for example, they can assess the properties of stochastic economic models, build policy scenarios, and condition model simulations upon judgmental adjustment.

***mathworks.com/imf***




## BANC SABADELL

### Building robust corporate development tools: providing analytics to thousands of users

Banc Sabadell's Quantitative Tools team develops and maintains a library of interest rate, commodities, foreign exchange, inflation hedging, and investment analytics. The team manages more than 80,000 lines of code in the library through Subversion source control. With MATLAB Compiler and MATLAB Builder JA, the team rapidly deploys analytics, tools, and market information, such as trading ideas and market parameters to more than 2000 users, through web interfaces, databases, and Microsoft® Excel®. In one project, the analyst defines a payoff structure through a web form, which is then priced using Monte Carlo methods. Execution speed for performance-critical applications is optimized using parallel and grid computing.

***mathworks.com/banc-sabadell***

## Learn More

**MATLAB Computational Finance Virtual Conference**
*mathworks.com/vc-comp-finance*

**User Stories**
*mathworks.com/user-stories*

# Developing and Deploying Sonar and Echosounder Data Analysis Software

By Jean-Marie Augustin, Ifremer

While satellite photographs and radar provide high-resolution images of virtually every square meter of the earth's surface, views of the ocean floor are spottier and less detailed. Yet accurate seabed maps are vital to scientific research and to many industrial applications. Beyond its interest for oceanography, the geosciences, and biology, precise knowledge of the contours and composition of the seabed helps power companies place wind farms and drilling platforms, communications companies plan where to lay fiber-optic cable, and environmental specialists evaluate the effects of climate change on oceans and seas.

The relatively obscure view of the seafloor is not due to a lack of data; ships equipped with multibeam echosounders (MBESs) and side-scan sonar systems survey wide areas of the sea, gathering terabytes of raw oceanographic data. Before scientists and engineers can apply this information in their work, however, they must convert it into meaningful data.

Although I am not an expert programmer, MATLAB® enabled me to apply my expertise in signal processing and sonar to develop and deploy SonarScope®, a high-performance software product for processing, analyzing, and visualizing raw MBES data. SonarScope is highly customizable and provides an original approach to MBES data processing, making it an invaluable tool for researchers and engineers who need to test and calibrate MBES systems.

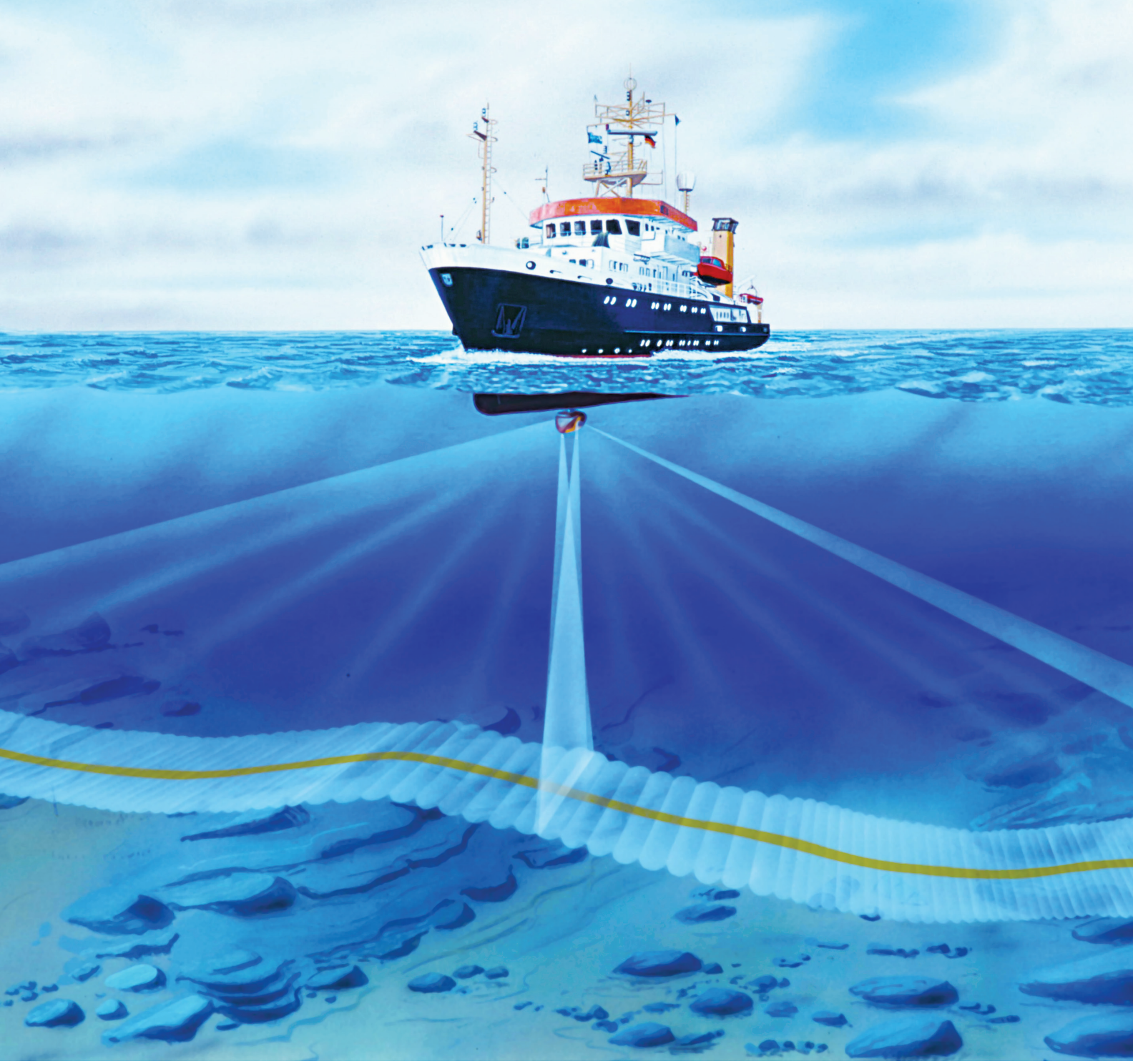
MATLAB proved to be an ideal environment for developing SonarScope because it enabled me to develop algorithms, visualize results, and then refine the algorithms in an iterative cycle. Design iterations take much longer with a language like C++, which requires additional compiling and linking steps, as well as a significant amount of additional programming to visualize results.

## From Basic Algorithms to Standalone Software

In MATLAB I develop algorithms interactively and can instantly plot the results to see the effect of any changes. Functions from Signal Processing Toolbox™, Image Processing Toolbox™, Optimization Toolbox™, and Statistics Toolbox™ further speed development because I don't have to write and debug them myself.

Soon after developing the algorithms that would ultimately form the basis of SonarScope I realized that my work could have a much broader reach, and that more of my colleagues could make use of my expertise, if I created a graphical interface to drive the analysis of MBES data. That motivated me to build the SonarScope interface using customized labels and buttons with terminology that my colleagues would readily understand (Figure 1). The interface was well received by my fellow researchers. It also raised awareness of my work among researchers at other institutions.

As interest in SonarScope expanded beyond Ifremer, I used MATLAB Compiler™ to build a standalone software package that can be used by researchers even if they do not have access to MATLAB. Today, SonarScope can be used by any researcher or



*Gathering MBES data. A single ping is transmitted into the water, and the echoes from the seafloor are received inside multiple beams. Image courtesy ATLAS HYDROGRAPHIC.*

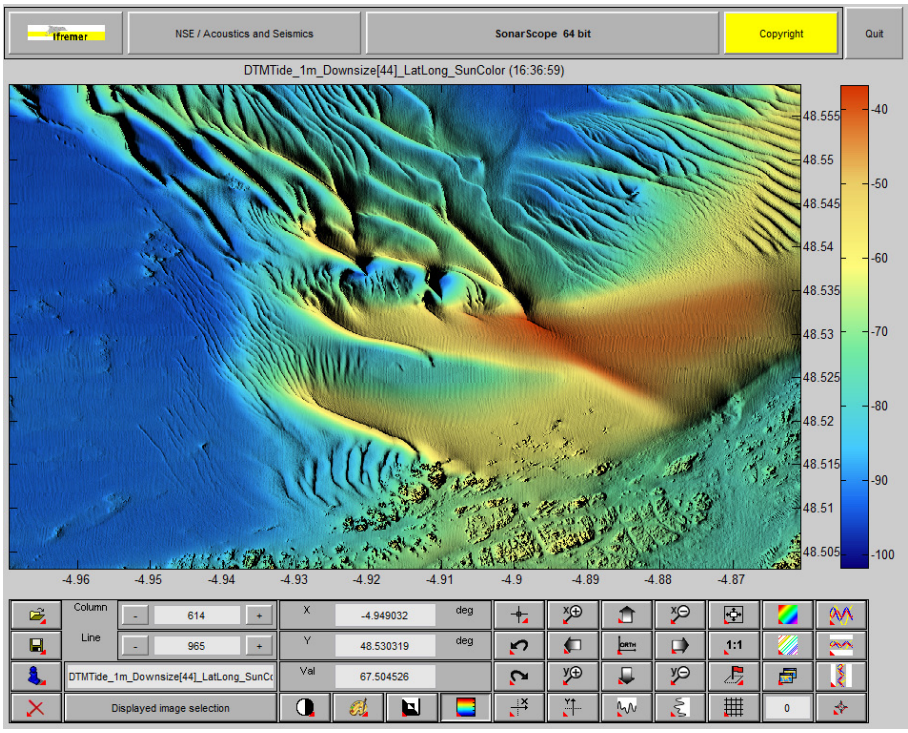



FIGURE 1. *The SonarScope interface, built using MATLAB. The central image displays the active data layer. The buttons in the lower part enable interaction with the data. The menus in the upper part provide quick access to the processing tools.*

engineer wishing to process seafloor mapping sonar data.

## Gathering the Data

To survey the seafloor, a ship equipped with an MBES covers the area to be mapped, following a series of parallel lines. The swaths covered on both sides of the survey lines can be up to 20 kilometers wide. They overlap to ensure that no area is left unmapped. As the ship travels along each swath, the MBES transmits a series of high-intensity, short-duration pings. Each ping is received and processed inside several hundred beams, which are steered in a span from directly below the ship to the edges of the swath.

For each beam, the MBES records the time lapse between transmitting the signal and receiving the echo. This metric is used to compute the oblique range from the ship to the ocean floor spot "seen" by the beam and, ultimately, to build a digital terrain model depicting the seafloor. The intensity of the echo is recorded because it correlates with the reflectivity of the seabed and hence with its physical characteristics. This reflectivity can be used to distinguish rock, sand, vegetation, and other features of the seabed.

To produce a map of the seafloor that is accurate to within centimeters, algorithms must take into account the echo delay, the transmit angle of the beam, and the motion (roll, pitch, and heave) of the ship. All this raw data is recorded, together with the ship's latitude and longitude for each ping transmitted by the MBES. Gigabytes—sometimes terabytes—of data are collected on a single survey, which usually lasts from a few days to several weeks.

## Performing Complex Transformations

Converting the raw data acquired from the MBES equipment into seafloor maps requires complex transformations involving multiple

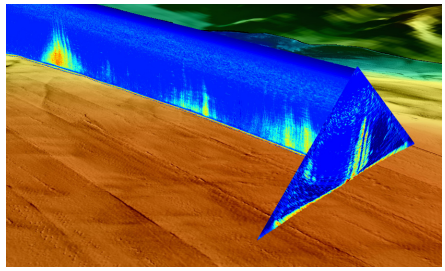


FIGURE 2. *A 3D compilation of bathymetry data processed by SonarScope.*

geometries. Using MATLAB and Mapping Toolbox™, I developed algorithms that process the data for a single ping and calculate the depth of the seafloor covered by the ping (Figure 2). In addition to identifying traits of the seafloor, the algorithms also detect features of the water. They can, for example, detect bubble plumes caused by gas released from the seafloor.

Beyond bathymetry, SonarScope performs many different kinds of data processing, all supported by MATLAB and related toolboxes. I used MATLAB, Statistics Toolbox, Image Processing Toolbox, and Signal Processing Toolbox to implement noise reduction, speckle filtering, segmentation, and bottom detection techniques. I used Optimization Toolbox for curve fitting throughout SonarScope. I plan to replace my own cartographic projections with the ones provided in Mapping Toolbox.

The segmentation algorithm is based on texture analysis using co-occurrence matrices and Gabor filters (Figure 3). The frontier lines were drawn automatically by the MATLAB based segmentation algorithm.

## Applying Object-Oriented Programming Techniques to Handle Extremely Large Data Sets

SonarScope comprises about 270,000 lines of MATLAB code. Similar software packages written in C commonly contain a million or more lines of code. Nevertheless, developing and maintaining any project with hundreds of thousands of lines of code can be a challenge

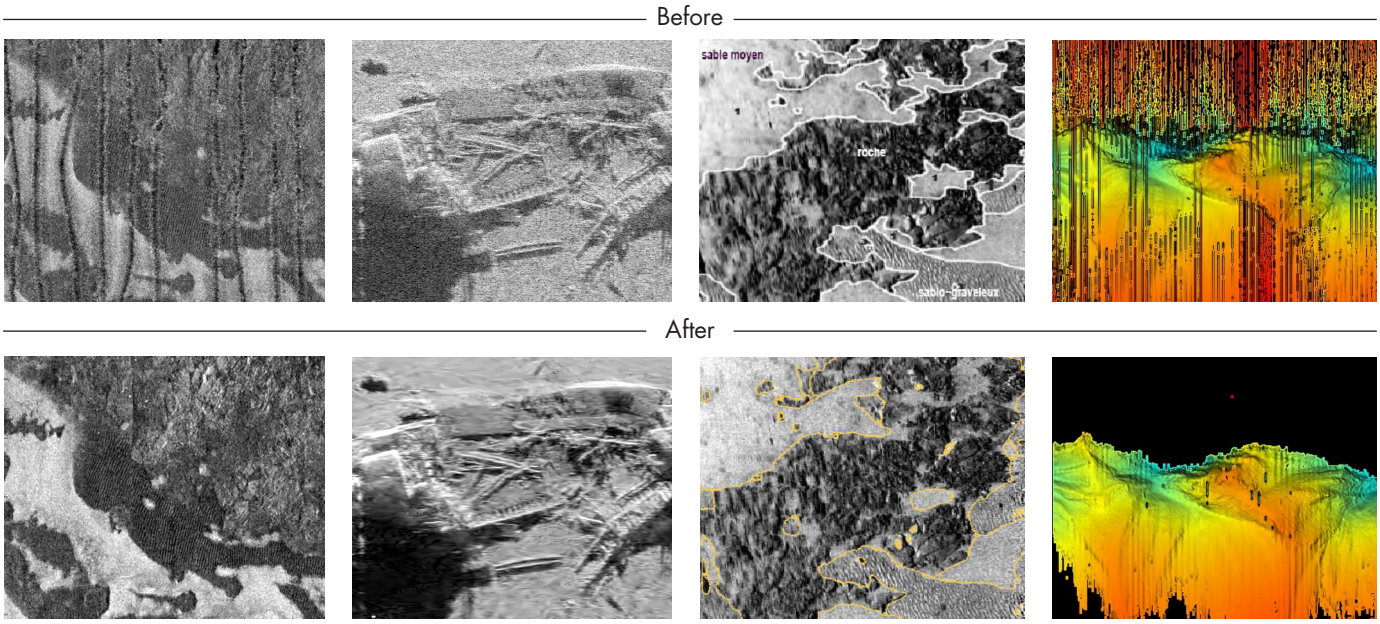



FIGURE 3. *Results of four image processing techniques. Left to right: mosaicing, speckle filtering, segmentation, and bottom detection. The speckle filter was designed by Pr. Alexandru Isar, Politechnica University, Timisoara, Roumania. The segmentation algorithm was developed by Imen Karoui, ENST Brest.*

without a way to organize and reuse the code efficiently. I used the object-oriented (OO) programming capabilities of the MATLAB language to create classes for components that are reused frequently throughout the application. For example, I defined a class for images, which makes it easy to manage and manipulate instances of images by changing parameter values.

Handling the extremely large data sets that SonarScope processes was also made easier by applying OO principles. When performing operations on large matrices holding gigabytes of data, it's easy to run out of computer memory. To solve this problem, I created a class that uses a MATLAB `memmapfile` object to map dynamic memory to files on a hard disk. Using this approach, I can easily work with 10,000 x 10,000 matrices and yet use only 120 bytes in memory. With this class, access to the values of a variable is the same whether the variable is a MATLAB matrix or a `cl_memmapfile` object.

The SonarScope interface takes advantage of reusable class objects and OO design patterns enabled by MATLAB to provide drag-and-drop capabilities, a property editor, keyboard shortcuts for panning and zooming, and custom menus.

## Building a Standalone Application

Many of the researchers, scientists, and engineers who use SonarScope are familiar with MATLAB. However, not every user is a MATLAB expert, nor do all users have MATLAB installed on their workstations. To deliver a solution to these users, I used MATLAB Compiler to create standalone 32-bit and 64-bit versions of SonarScope for Windows® and Linux® operating systems.

With its detailed and customizable data processing capabilities, SonarScope excels at helping engineers verify, calibrate, and troubleshoot their MBES hardware. When MBES systems produce extraneous artifacts that show up in the data, engineers use SonarScope to examine the data dynamically and identify the source of the artifacts. In one case, to minimize the number of spikes in their bathymetry data, an MBES manufacturer replaced the bottom detection algorithm they were using with the one from SonarScope.

Development of SonarScope is active and ongoing. I continue to update the code to take advantage of new features as they become available in MATLAB, and to enable me to collaborate effectively with more research colleagues worldwide. ▪

## Learn More

**SonarScope**
*www.ifremer.fr/fleet/acous_sism/sonarscope/index.html*

**Video: Deploying Applications with MATLAB**
*mathworks.com/deploying-applications-matlab*

**Introduction to Object-Oriented Programming in MATLAB**
*mathworks.com/oop-in-matlab*

# Simulating a Piezoelectric-Actuated Hydraulic Pump Design at Fraunhofer LBF and Ricardo

By H. Atzrodt, Fraunhofer LBF, and Dr. A. Alizadeh, Ricardo Deutschland GmbH

Hydraulic pumps are often the focus of automotive engineers attempting to reduce energy losses. For example, power steering systems have progressed from purely hydraulic to electro-hydraulic and even purely electric to make the system more efficient. Engineers at Fraunhofer LBF and Ricardo have focused on incorporating innovative technologies into hydraulic pumps to improve system performance and efficiency.

Working together, we developed a novel pump design that leverages piezoelectric actuators, offering both a significant reduction in energy losses and improved response time. Simulation enabled us to verify that the design met specifications. It also reduced development time by 50%.

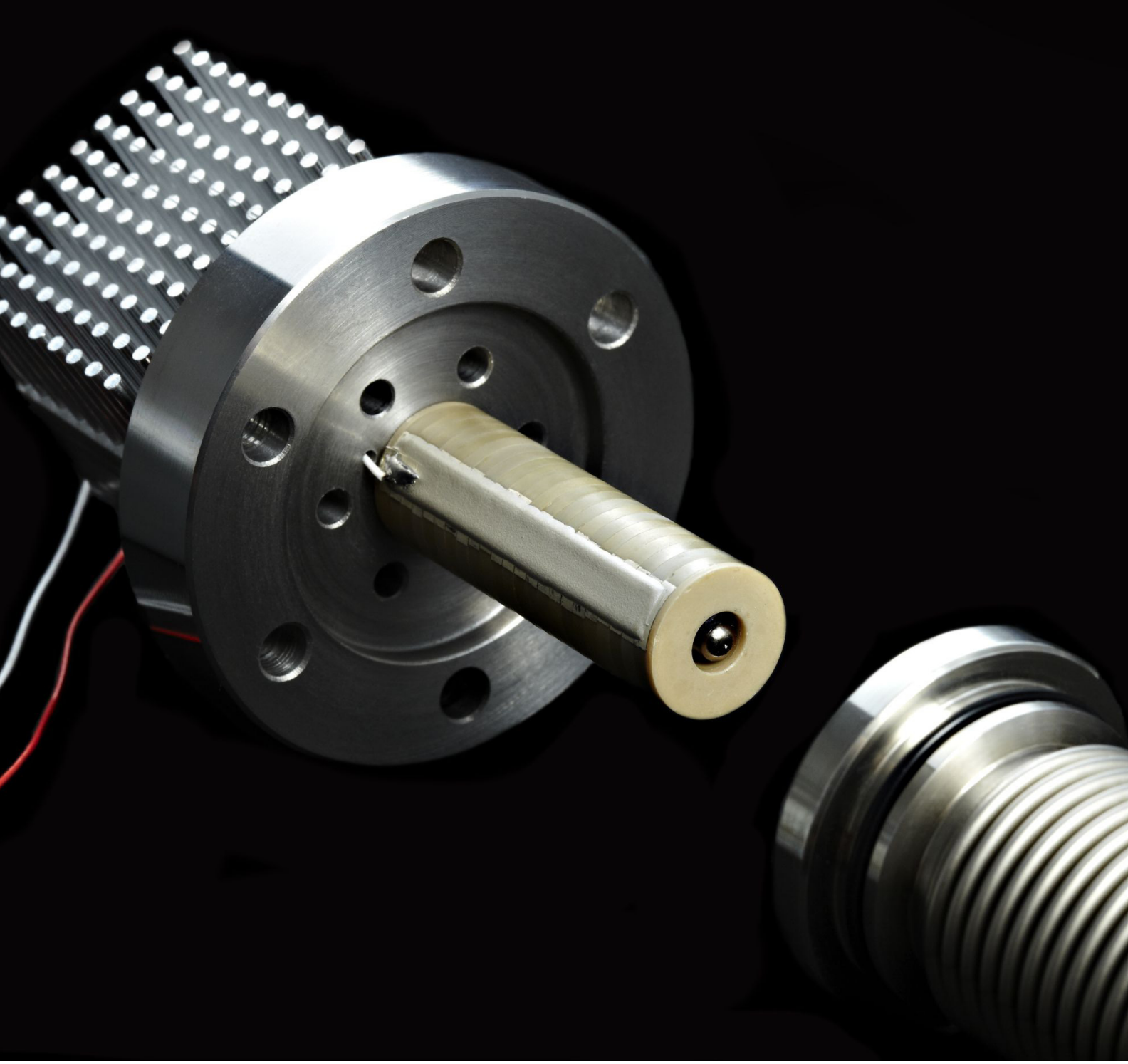
In automotive applications, piezo-actuated hydraulic pumps (Figure 1) have mainly been used in applications that require low flow rates and pressures. For automotive control systems, pressures of 20–200 bar are necessary, which is much higher than the pressure produced by pump designs currently in series production.

To create a design that produces the pressures required for automotive control systems, it was critical to simulate the entire system, including the pump, valves, and actuator (Figure 2). We could not rely solely on separate simulations performed in domain-specific tools, because improving the efficiency of a system that unites multiple physical domains requires system-level optimization. The ability to calibrate and validate the models was also critical, because the models we use range from extremely detailed finite element models to system-level. The high-level system models help us keep run times as short as possible and enable wide-ranging parameter variations. Using Model-Based Design to develop the pump enabled us to increase the output pressure of existing designs by a factor of 10.

## Selecting the Application and Pump Architecture

For our new design, we chose the pump used in selective catalytic reduction (SCR). SCR systems are used in diesel engines, where strict emission and fuel economy requirements are motivating the development of efficient emission controls. The pump injects a liquid reductant into the automobile exhaust stream, which promotes the conversion of nitrous oxide into diatomic nitrogen ($N_2$) and water. The reductant in our system is AdBlue®, a urea solution. Our pump must produce an output pressure of 50 bar to deliver a high-quality spray that will improve the mixture.

Using SimHydraulics® we were able to test different pump architectures to find the one most likely to enable a piezoelectric actuator to develop the pressure required while maintaining its fast response. To minimize costs, we wanted an architecture that uses only one pump rather than designs requiring an additional feed pump to achieve the required pressure. Tests in SimHydraulics let us refine the pump specification by determining the input



*Hydraulic pump driven by a piezoelectric actuator for use in automotive applications.*

pressure and spring stiffness for the input and output valve springs.

During architecture selection, we tested the sensitivity of the design to various component parameters. We determined that the ability of the pump to meet specifications was heavily influenced by the design of the clamped disc spring, which forms the seal for the pump chamber. SimHydraulics simulations were helpful in determining the spring design requirements (minimum and maximum spring rate, geometry, and disc stiffness).

## Incorporating a Detailed Design into the System-Level Model

Finite element analysis was useful when we designed the clamped disc spring. To find a design that offered the minimum bending volume, maximum delivery chamber volume, and minimum stiffness, we used a tool developed in-house by Ricardo. Design of Experiments methods were used to identify the optimal design. We incorporated the spring behavior into the SimHydraulics model to see if the overall system still met the design requirements with the new clamped disc spring.

The check valves are crucial to the performance of the pump. Engineers working on the project have extensive experience with these types of valves, so they customized the standard SimHydraulics components to obtain exactly the effects they wanted to capture in their model. Simulation analysis showed that the eigenfrequency of the pump valves needed to be very close to the operating frequency of the pump to avoid interactions with the natural frequency of the system.

At the start of the design, the team used ideal actuator models to ensure that the required force stayed within a range that a piezoelectric actuator could produce. Next, we introduced Simulink® models of piezoelectric stack actuators to verify that the dynamic performance of the actuator would still enable the pump to meet system-level requirements. The amplifier and power supply were included in the model to add the electrical portion of the system into the simulation.

FIGURE 1. *Test bench for a hydraulic pump driven by a piezoelectric actuator.*

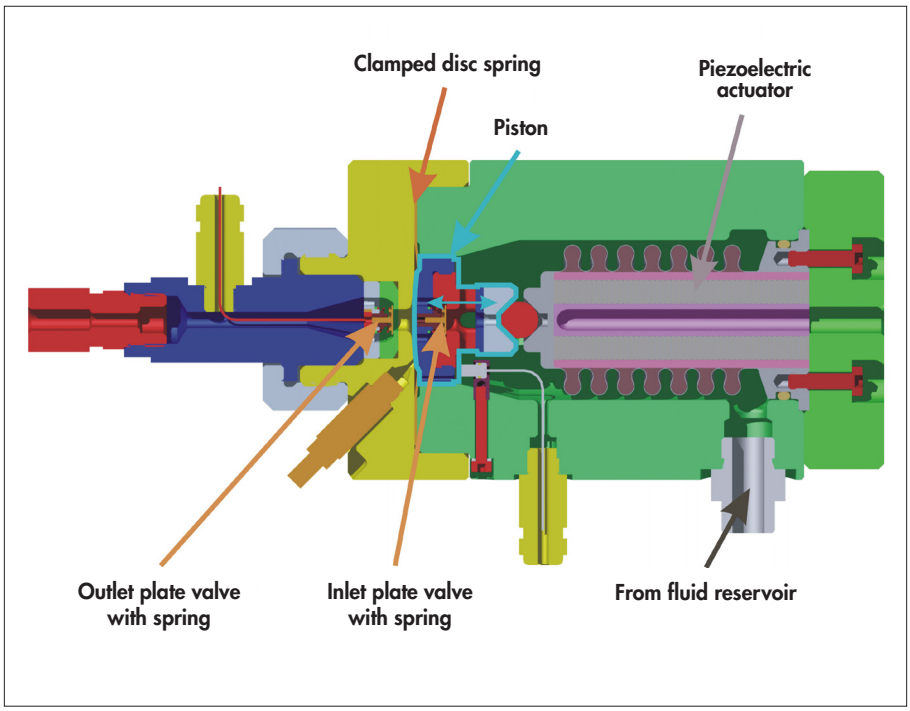



FIGURE 2. *Diagram of the piezoelectric-actuated pump consisting of a piezoelectric actuator, clamped disc spring, hydraulic fluid, valves, and electrical actuation.*

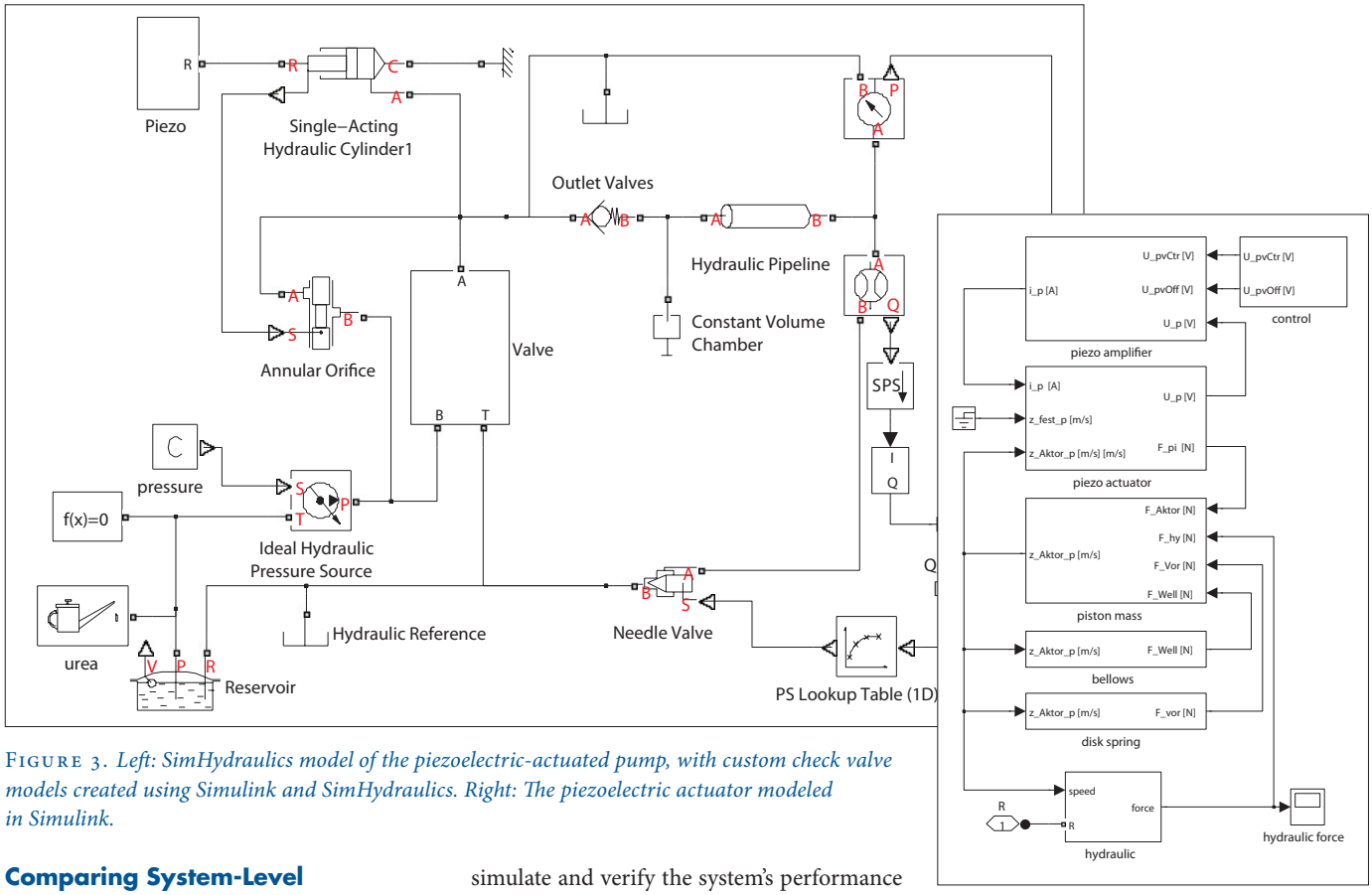



FIGURE 3. *Left: SimHydraulics model of the piezoelectric-actuated pump, with custom check valve models created using Simulink and SimHydraulics. Right: The piezoelectric actuator modeled in Simulink.*

## Comparing System-Level Performance to the Specification

Engineers simulated the entire system within the Simulink environment (Figure 3). The complexity of the hydraulic system, with multiple valves, fluid compressibility, hysteresis, and multiple flow paths, would have been extremely difficult to model without the blocks that SimHydraulics provides. The team used MATLAB® to postprocess simulation results and ensure the design was meeting system requirements.

Simulation showed that we would need to adjust the command signal and amplifier to achieve the desired pressure profiles. The results also showed that the architecture with the optimized design for the clamped disc spring met the pressure requirements for the piezoelectric-actuated pump.

We verified our simulation results on actual hardware. The first prototype that we built verified the simulation results and met the pump specifications. Without the ability to simulate and verify the system's performance in a single environment, we would have had to rely on separate simulation tools and hardware prototypes. It would have taken twice as long to create the design, and it would not have been possible to increase the output of the pump by a factor of 10, which was required for this design.

## Future Development

Simulation is essential for development programs at Fraunhofer LBF and Ricardo. With Simscape™ and SimHydraulics, we can optimize designs of multidomain systems. We are currently building a second prototype for our piezoelectric-actuated pump based on further refinements to the design, and we expect to see even better performance in the future. ▪

## Learn More

**Designing Pitch and Yaw Actuators for Wind Turbines**
*mathworks.com/wbnr37405*

**Parameterization of Directional and Proportional Valves in SimHydraulics**
*mathworks.com/parameterization-valves*

**Video: Modeling a Hydraulic Actuation System**
*mathworks.com/hydraulic-actuation*

**Download: Hydraulic Valve Parameters from Data Sheets and Experimental Data**
*mathworks.com/download-hydraulic-valve*

# Improving the Efficiency of RF Power Amplifiers with Digital Predistortion

By George Vella-Coleiro, CommScope

When operating at near-peak efficiency, the RF power amplifiers commonly used in wireless base stations distort the signal they amplify. The distortions not only affect signal clarity, they also make it difficult to keep the signal within its assigned frequency band. Base station operators risk violating FCC and international regulatory agency standards if they cannot keep spurious amplifier emissions from interfering with adjacent frequencies. Today's WCDMA and LTE carriers have wider bandwidths than their predecessors, increasing the likelihood of interference from spurious emissions.

To reduce these emissions and achieve more linear amplifier output, base station operators can reduce the power output of the amplifier, but this practice also reduces efficiency. Amplifiers operating below peak efficiency dissipate more energy and get hot, sometimes requiring costly cooling equipment to prevent overheating.

At CommScope, we develop digital predistortion (DPD) systems that provide a way to operate amplifiers efficiently while improving linearity and minimizing spurious emissions. DPD alters the signal before it is amplified, counteracting the amplifier's distortion to produce a clearer output signal. Unlike their analog counterparts, DPD systems operate in the digital domain, enabling engineers to build flexible and adaptive solutions that produce a cleaner output signal.

We have developed, implemented, and patented DPD technology that enables wireless base stations to operate more efficiently while complying with stringent regulatory requirements. We rely on MATLAB® and Simulink® to characterize power amplifiers and their distortion, model and simulate DPD designs, and verify our hardware implementations.

## Characterizing the Power Amplifier

DPD addresses two types of distortion. Type 1 distortion results from curvature of the amplifier transfer function. Also known as amplitude modulation–to–amplitude modulation (AM-AM) and amplitude modulation–to–phase modulation (AM-PM) distortion, this effect is not a function of signal bandwidth. In contrast, Type 2 distortion, also known as memory effect, is a function of signal bandwidth, becoming more prominent as bandwidth increases.

The first step in designing a DPD system is to characterize fully the Type 1 and Type 2 distortion caused by the amplifier to which it will be coupled. At CommScope, we use MATLAB to characterize power amplifiers in the lab. After generating the baseband waveform with MATLAB, we download it to a signal generation chain that produces the signal to the amplifier. A second chain captures the amplifier output and feeds it back into MATLAB (Figure 1).

## Tackling Type 1 Distortion

After running tests of the actual amplifier in the lab, we use MATLAB to compare the input waveform to the output and determine how the amplifier is distorting the signal.

*A wireless base station.*

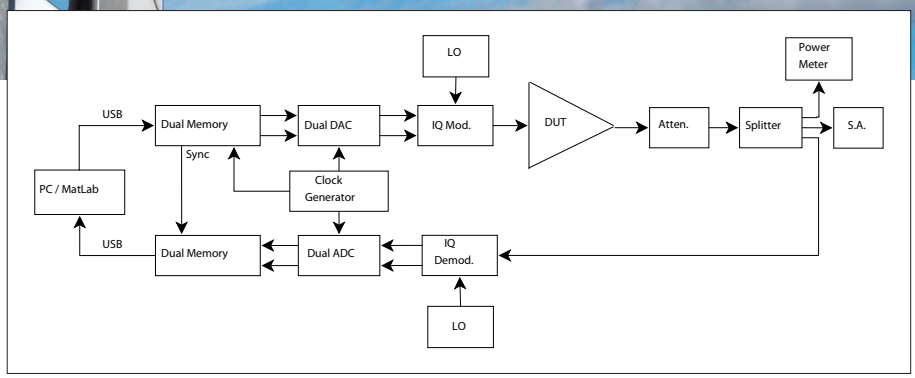
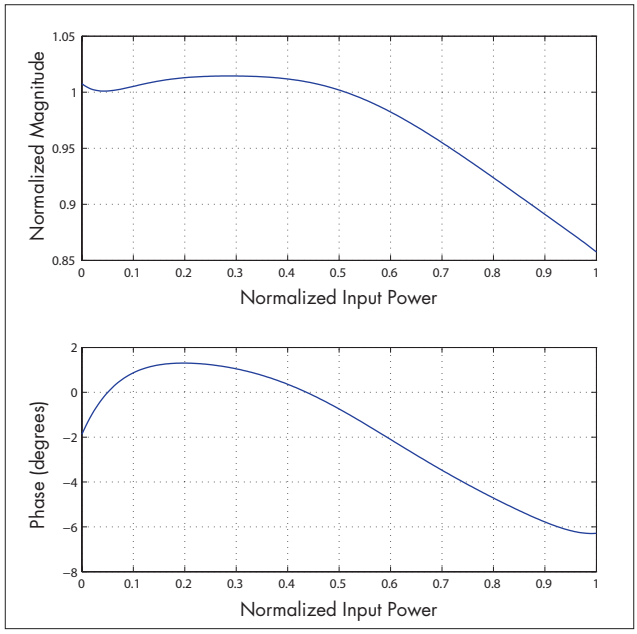



FIGURE 1. *Diagram of the lab setup for amplifier characterization.*



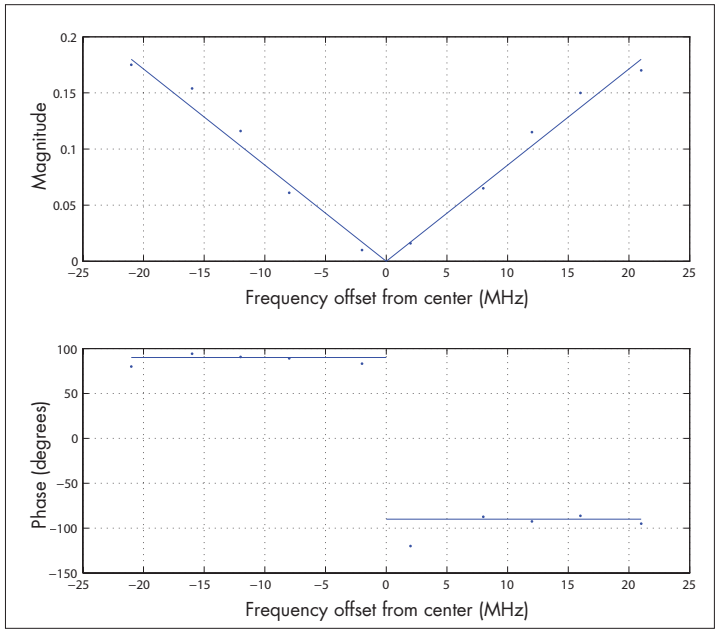




FIGURE 2 (LEFT). *Magnitude transfer function (top) and phase transfer function (bottom) showing Type 1 distortion. The raw data has been fitted to a pair of polynomials to average the fluctuations and produce smooth curves.* FIGURE 3 (RIGHT). *Magnitude and phase of Type 2 distortion.*

This data processing includes time, gain, and phase alignment of the two signals. Based on our analysis, we generate a plot of the amplifier's transfer function to visualize the Type 1 distortion (Figure 2).

If the amplifier did not distort the signal at all, the normalized magnitude would be plotted as a horizontal line with a value of 1, and the phase would be constant at 0 degrees. Instead, however, the normalized magnitude and phase of a typical amplifier vary as a function of input power. We obtain the complex gain of the amplifier by dividing the output samples by the input samples in MATLAB. The Type 1 DPD correction is simply the inverse of this complex gain.

## Characterizing Type 2 Distortion

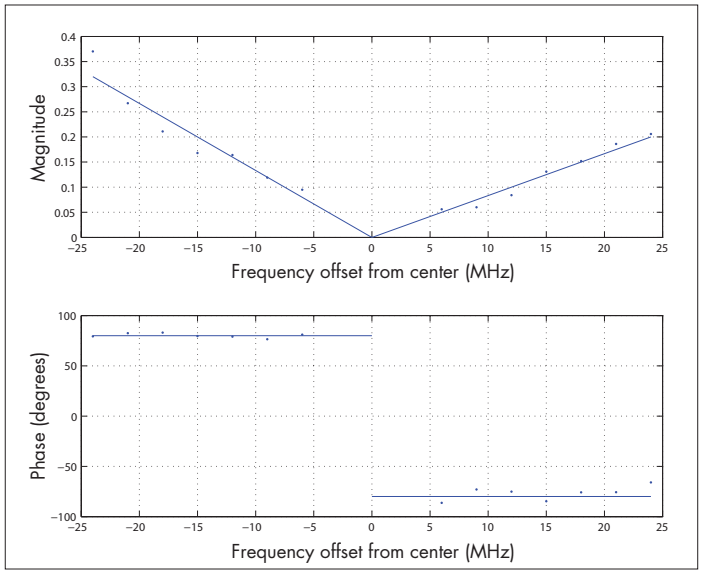
Characterizing Type 2 distortion requires a more sophisticated approach, because this type of distortion depends on signal bandwidth. We begin by driving the amplifier with a signal generated in MATLAB that consists of two narrow-bandwidth carriers separated by a gap. After applying the Type 1 DPD to remove AM-AM and AM-PM effects, we add third-order predistortion with adjustable magnitude and phase. Using MATLAB optimization functions, including `fminbnd` and `fminsearch`, we optimize the magnitude and phase to minimize the third-order distortion exhibited by the amplifier, first on one side of the center frequency and then on the other. We repeat this process for multiple carrier spacings to map fully the amplifier's Type 2 distortion as a function of frequency. Using third-order predistortion is sufficient to characterize the amplifier because the frequency dependence of the higher-order distortions follows that of the third-order distortion.

Using a series of MATLAB generated waveforms with different spacing between the two carriers, the magnitude and phase of Type 2 distortion is measured for various frequencies across the bandwidth of interest. Then we generate a plot in MATLAB to visualize the results (Figure 3). The magnitude of the distortion increases as the frequency moves away from the center, supporting our observation that Type 2 distortion worsens as the bandwidth of the signal increases. Likewise, the 180-degree jump in the phase plot supports our observation that minimizing distortion on one side of the center frequency makes it worse on the opposite side.

We developed a mathematical model of the amplifier and its distortion. The general form of this model is

$$Y = X + \underbrace{X f_1(P)}_{\text{Type 1}} + \underbrace{d\,\{X f_2(P)\} / dt}_{\text{Type 2}}$$

where X is the amplifier input, Y is the amplifier output, and P is the instantaneous envelope power. Modeling the Type 2 distortion requires a differentiation with respect to time in order to match the characteristics shown in Figure 3 (a magnitude that varies linearly with frequency offset from center, and a phase that changes by 180 degrees at zero offset frequency). Using MATLAB, we fit the parameters of this model to the data gathered during the characterization process. The parameters we fit are complex values that are used to multiply the third-order



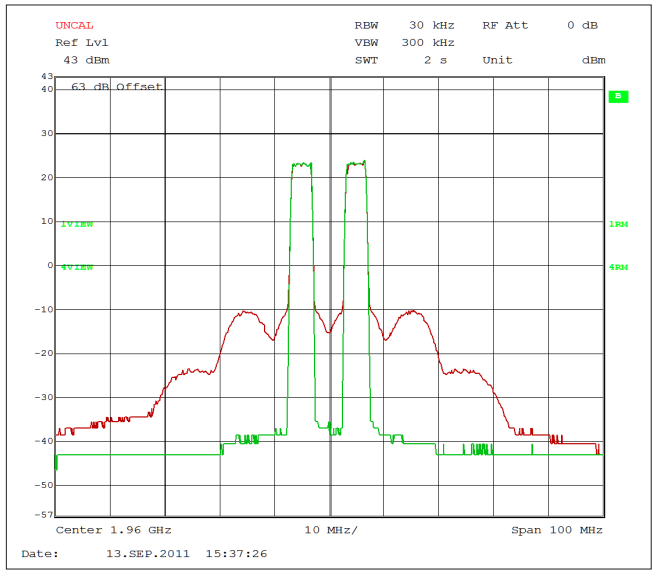




FIGURE 4 (LEFT). *Magnitude and phase of Type 2 distortion for a Doherty amplifier.* FIGURE 5 (RIGHT). *Doherty amplifier spectra for WCDMA carriers. Red = the original output spectrum. Green = the spectrum after applying Type 1 and Type 2 digital predistortion.*

and fifth-order Type 2 distortion (higher orders of distortion are typically not required). We verify that the model accurately reflects the behavior of the amplifier by supplying the amplifier and the model with the same input and comparing the output.

Once we have an accurate model of the amplifier, we develop a DPD model that corrects the distortion caused by the amplifier.

## The Doherty Complication

Our initial DPD development process focused on class AB amplifiers, which were a staple of the industry until a few years ago. As WCDMA and LTE systems became more popular, engineers began exploring amplifier designs capable of efficiently handling signals with high peak-to-average ratios. At CommScope, we returned to an old idea: the Doherty amplifier. First described in 1936, the Doherty amplifier is now the predominant choice in wireless transmitters for WCDMA and LTE systems.

With class AB amplifiers, the magnitude of the Type 2 distortion is symmetric about the center frequency, whereas with Doherty amplifiers, it is not (Figure 4). Also, in a Doherty amplifier the change of phase angle is not 180 degrees.

Handling this asymmetry requires a minor modification to the mathematical model of the amplifier:

$$Y = X + Xf_1(P) + P[d\{Xf_2(P)\} / dt] + N[d\{Xf_3(P)\} / dt]$$

where X is the amplifier input, Y is the amplifier output, P is the instantaneous envelope power, *P* is a positive frequency pass filter, and *N* is a negative frequency pass filter.

This model includes separate terms for positive and negative frequencies. We use frequency pass filters to apply these terms to the appropriate frequencies. We designed the filters using the Filter Design and Analysis Tool in Signal Processing Toolbox™. Figure 5 compares the original output spectrum of a Doherty amplifier for two WCDMA carriers with the output spectrum after applying Type 1 and Type 2 digital predistortion.

## Making the Most of the Models

The MATLAB models that we created for RF power amplifier DPD are exceptionally versatile and useful for many other groups in CommScope, both in the U.S. and elsewhere. We used MATLAB Compiler™ to create standalone versions of the models. Now, other teams within CommScope can use the models even on computers that do not have MATLAB installed.

CommScope engineers continue to push the boundaries of power amplifier efficiency with seventh-order predistortion and other technologies. As with the development of our original DPD technology, MATLAB is essential to this process because it lets us rapidly try different approaches and identify the best one before committing our designs to hardware. ▪

## Learn More

**Download: Mixed-Signal Library for Simulink**
*mathworks.com/mixed-signal-library*

**Verification of High-Efficiency Power Amplifier Performance at Nujira**
*mathworks.com/power-amplifier*

# University of Adelaide Undergraduates Design, Build, and Control an Electric Diwheel Using Model-Based Design

By Dr. Ben Cazzolato, University of Adelaide

In their final year, honors undergraduates at the University of Adelaide are encouraged to complete a year-long capstone course in which they apply the skills and knowledge acquired in their coursework to a practical, hands-on project. Ideas for the capstone projects come from industry, faculty, or the students themselves. For aerospace and mechanical engineering students, the projects often require the design and implementation of real-time control systems and real hardware systems—past projects have included self-balancing scooters, unicycles, and robots, as well as automated reversing systems for double tractor-trailers.

Completed projects are demonstrated at an annual exhibition open to the public. One project that repeatedly garners attention from the media and from other schools is EDWARD, or Electric DiWheel with Active Rotation Damping. Powered by on-board batteries, the diwheel's two electric motors are capable of propelling the vehicle at speeds of up to 40 kph.

The original diwheel was designed and constructed three years ago by a team of four engineering students. Since then, two more teams have significantly enhanced and refined the diwheel.

All three teams used Model-Based Design with MATLAB® and Simulink®. Model-Based Design supports a workflow that incorporates system modeling, control design, simulation, code generation, and rapid prototyping (Figure 1). This workflow lets students focus on system issues instead of low-level C coding. As a result, they achieve a great deal with relatively little prior experience, time, or capital expenditure. By the end of the year they've completed an entire engineering project, from analysis of a problem through delivery of an embedded system.

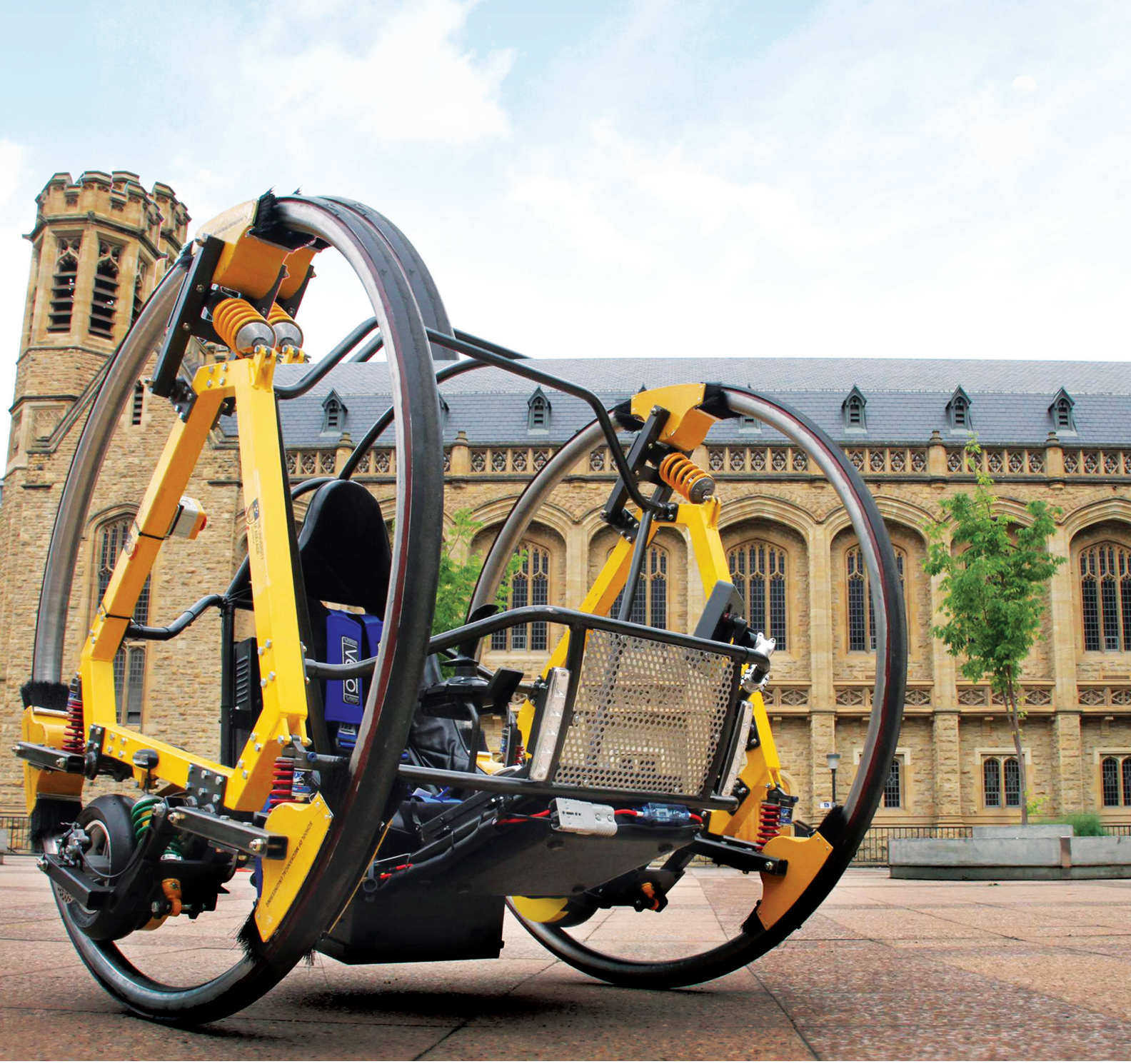
## Advantages of the Diwheel as a Capstone Project

The diwheel has all the characteristics of a good capstone project. Most important from the students' point of view, perhaps, is that the diwheel is fun to drive—students report that it gives them the same adrenaline rush as a roller coaster. The mechanical design is interesting but not too detailed to grasp. The relatively simple design requires sophisticated controls, engaging the students in feedback control, safety, and other fundamental engineering issues. In a practical vehicle, the center of gravity would be kept as low as possible to increase stability. We deliberately selected a high center of gravity for the diwheel, to increase the control design challenge.

Media coverage of the diwheel project has brought additional, unexpected benefits: By demonstrating what our students can achieve after only four years of study, the press reports have attracted new students and more support to the engineering program at the University of Adelaide.

## Integrating MATLAB and Simulink into the Curriculum

Engineering students at the University of Adelaide begin preparing for the capstone projects as early as their second year, when they take their first controls courses. Based on MATLAB and Simulink, these courses



*The electric diwheel designed and built by the University of Adelaide undergraduates.*

cover classical, digital, proportional-integral-derivative (PID), and state-space control with an emphasis on rapid prototyping processes in the latter years. The University of Adelaide has a Total Academic Headcount license, which makes it easy for students to access the MathWorks products they need for their assignments from anywhere on or off campus.

The outstanding outcomes that we see in the controls-based capstone projects are a direct result of integrating MATLAB and Simulink into our controls courses and focusing on rapid prototyping throughout the program. Students do not have to use Model-Based Design for their capstone project; they are free to choose the engineering approach they think will work best. Without modern rapid prototyping tools, however, the diwheel project and others like it would simply not be possible. Students would be unable to simulate their systems or ensure their controllers were safe. The groups that try to write their code manually in C never get anywhere near as far as those who use Model-Based Design.

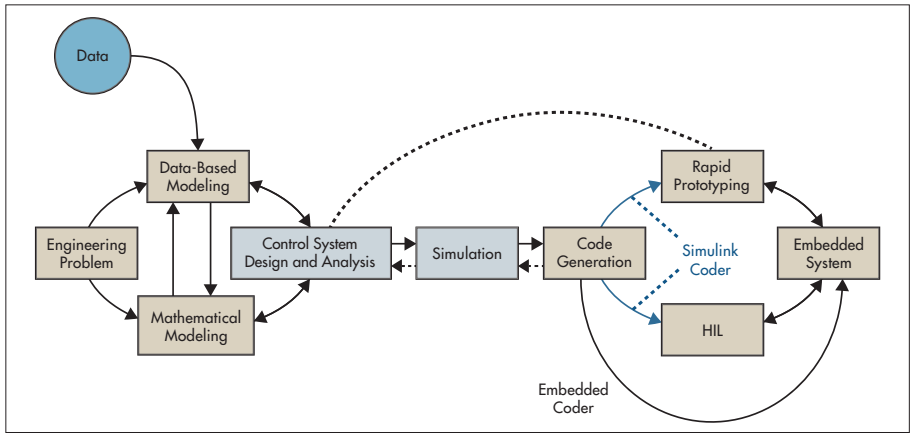



FIGURE 1. *A typical controller development workflow for capstone projects.*

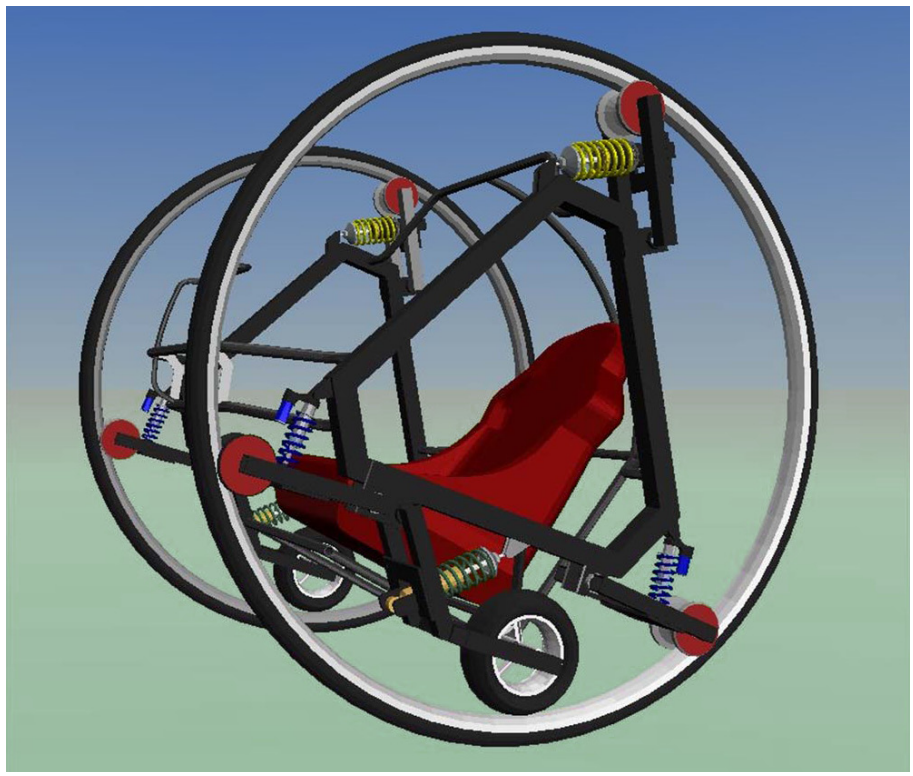


FIGURE 2. *A virtual reality model of the diwheel.*

## Year 1: From Idea to Working Diwheel

Because the students who built the original diwheel were starting from scratch, one of their first tasks was to create the mechanical design. They developed a 2D plant model of the diwheel in Simulink and ran simulations to better understand its tumble and slosh behavior. To derive the system dynamics, they developed the Lagrangian formulation in MATLAB. Later, a graduate student developed an equivalent model in SimMechanics™. The undergraduate team and I used this model to validate the simulation results.

Working in Simulink, the students developed a controller that used input from a gyroscope to control the slosh. They ran closed-loop simulations of the controller and plant models together to verify the functionality of the control system. Simulink 3D Animation™ enabled them to visualize the simulation results in a 3D environment (Figure 2). Compared with plots of raw data, these visualizations made it much easier for the students to see how their control algorithms would perform once implemented on the diwheel. After validating the controller model, they generated C code using Simulink Coder™, and deployed the compiled code to a microprocessor installed on the diwheel.

After just one year of development, the students had designed and built a working diwheel that they could safely drive—a significant achievement.

## Year 2: Inversion Control and Other Enhancements

In the second year of the diwheel project, an entirely new group of students picked up

FIGURE 3. *The diwheel with its inversion controller engaged.*

where the first group left off. A key objective was to build two new controllers, one to enable the diwheel pilot to swing upside down and a second to keep the diwheel in an unstable inverted state indefinitely (Figure 3).

Before they could develop these controllers, the students needed a plant model of the diwheel that incorporated yaw as well as pitch. They used Symbolic Math Toolbox™ to derive the Lagrangian dynamics for a fully coupled, three-degree-of-freedom model that incorporated a yawing component. Using Simulink, Control System Toolbox™, and this plant model, they developed models of the swing-up controller, an improved slosh controller, and the inversion controller.

The new control systems took advantage of upgraded hardware on the diwheel, including new accelerometers, speed sensors, and gyros, as well as more powerful motors. Like their predecessors, this group of students used Simulink Coder to generate code from their models, but they targeted dSPACE rapid prototyping hardware instead of a microcontroller. The dSPACE hardware made it easier to add an on-board display screen that showed speed, battery charge levels, and the status of various states. After conducting initial tests with the new diwheel, the students performed basic system identification using measured data to improve the accuracy of the plant model, which enabled further refinements to the controller models.

## Year 3 and Beyond

A third group of students has begun building on the diwheel design created by last year's group. Once again, many of the hardware components have been updated, including the power electronics and batteries. These hardware updates require the redesign of some control algorithms. The students are also working on improving overall reliability and performance, as well as fixing known problems from earlier versions. These are exactly the kinds of activities that would be necessary in the real world, where engineering teams are frequently required to improve existing designs, including designs taken over from another group or a supplier.

Model-Based Design with MATLAB and Simulink enables University of Adelaide students to take on more ambitious and more interesting capstone projects because they can model, design, verify, and implement their real-time control systems in a single environment. They do not have to learn disparate tools or spend much of the year writing and debugging C code. As a result, they acquire a deep understanding of controls that will help them in virtually any career path they choose—indeed, some of the more accomplished students have the skills of seasoned controls engineers well before they graduate. ▪

## Learn More

**Video: EDWARD - Electric Diwheel With Active Rotation Damping**
*http://youtu.be/Uf6Gh-hPDeo*

**Hardware for Project-Based Learning**
*mathworks.com/academia/hardware-resources*

**Engaging Students in Hands-on Control System Design at the University of Arizona**
*mathworks.com/engaging-students*

# GPU Programming in MATLAB

By Jill Reese and Sarah Zaranek, MathWorks

Multicore machines and hyper-threading technology have enabled scientists, engineers, and financial analysts to speed up computationally intensive applications in a variety of disciplines. Today, another type of hardware promises even higher computational performance: the graphics processing unit (GPU).

Originally used to accelerate graphics rendering, GPUs are increasingly applied to scientific calculations. Unlike a traditional CPU, which includes no more than a handful of cores, a GPU has a massively parallel array of integer and floating-point processors, as well as dedicated, high-speed memory. A typical GPU comprises hundreds of these smaller processors (Figure 1).

The greatly increased throughput made possible by a GPU, however, comes at a cost. First, memory access becomes a much more likely bottleneck for your calculations. Data must be sent from the CPU to the GPU before calculation and then retrieved from it afterwards. Because a GPU is attached to the host CPU via the PCI Express bus, the memory access is slower than with a traditional CPU.[1] This means that your overall computational speedup is limited by the amount of data transfer that occurs in your algorithm. Second, programming for GPUs in C or Fortran requires a different mental model and a skill set that can be difficult and time-consuming to acquire. Additionally, you must spend time fine-tuning your code for your specific GPU to optimize your applications for peak performance.

This article demonstrates features in Parallel Computing Toolbox™ that enable you to run your MATLAB® code on a GPU by making a few simple changes to your code. We illustrate this approach by solving a second-order wave equation using spectral methods.

## Why Parallelize a Wave Equation Solver?

Wave equations are used in a wide range of engineering disciplines, including seismology, fluid dynamics, acoustics, and electromagnetics, to describe sound, light, and fluid waves.

An algorithm that uses spectral methods to solve wave equations is a good candidate for parallelization because it meets both of the criteria for acceleration using the GPU:

***It is computationally intensive.*** The algorithm performs many FFTs and IFFTs. The exact number depends on the size of the grid (Figure 2) and the number of time steps included in the simulation. Each time step requires two FFTs and four IFFTs on different matrices, and a single computation can involve hundreds of thousands of time steps.

***It is massively parallel.*** The parallel fast Fourier transform (FFT) algorithm is designed to "divide and conquer" so that a similar task is performed repeatedly on different data. Additionally, the algorithm requires substantial communication between processing threads and plenty of memory bandwidth.

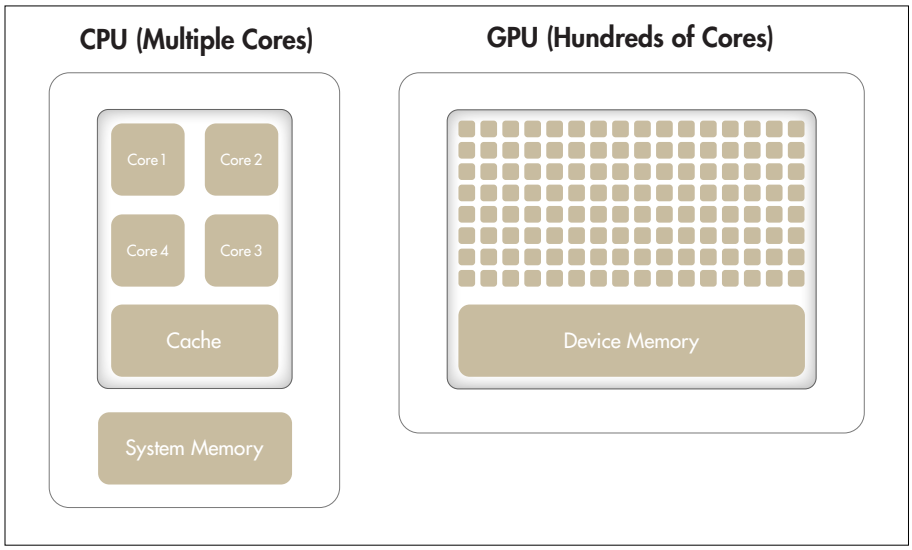



FIGURE 1. *Comparison of the number of cores on a CPU system and a GPU.*

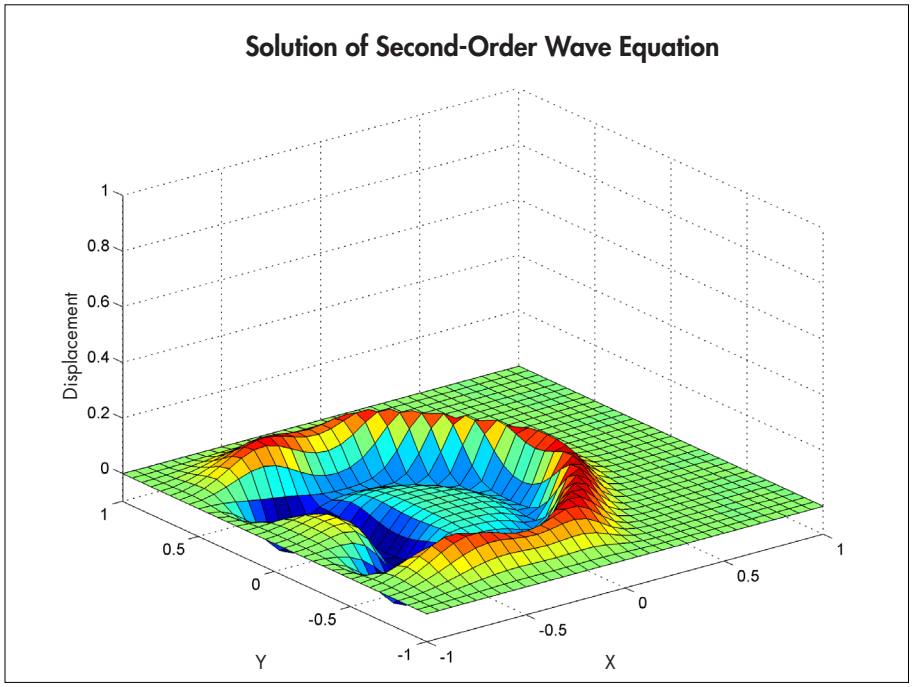



FIGURE 2. *A solution for a second-order wave equation on a 32 x 32 grid.*

The inverse fast Fourier transform (IFFT) can similarly be run in parallel.

## GPU Computing in MATLAB

Before continuing with the wave equation example, let's quickly review how MATLAB works with the GPU.

FFT, IFFT, and linear algebraic operations are among more than 100 built-in MATLAB functions that can be executed directly on the GPU by providing an input argument of the type GPUArray, a special array type provided by Parallel Computing Toolbox. These GPU-enabled functions are *overloaded*—in other words, they operate differently depending on the data type of the arguments passed to them.

For example, the following code uses an FFT algorithm to find the discrete Fourier transform of a vector of pseudorandom numbers on the CPU:

```
A = rand(2^16,1);
B = fft (A);
```

To perform the same operation on the GPU, we first use the `gpuArray` command to transfer data from the MATLAB workspace to device memory. Then we can run `fft`, which is one of the overloaded functions on that data:

```
A = gpuArray(rand(2^16,1));
B = fft (A);
```

The `fft` operation is executed on the GPU rather than the CPU since its input (a GPUArray) is held on the GPU.

The result, B, is stored on the GPU. However, it is still visible in the MATLAB workspace. By running `class(B)`, we can see that it is a GPUArray.

```
class(B)
ans =
parallel.gpu.GPUArray
```

We can continue to manipulate B on the device using GPU-enabled functions. For example, to visualize our results, the `plot` command automatically works on GPUArrays:

```
plot(B);
```

To return the data back to the local MATLAB workspace, you can use the `gather` command; for example:

```
C = gather(B);
```

`C` is now a double in MATLAB and can be operated on by any of the MATLAB functions that work on doubles.

In this simple example, the time saved by

executing a single FFT function is often less than the time spent transferring the vector from the MATLAB workspace to the device memory. This is generally true, but is dependent on your hardware and size of the array. Data transfer overhead can become so significant that it degrades the application's overall performance, especially if you repeatedly exchange data between the CPU and GPU to execute relatively few computationally intensive operations. It is more efficient to perform several operations on the data while it is on the GPU, bringing the data back to the CPU only when required.[2]

Note that GPUs, like CPUs, have finite memories. However, unlike CPUs, they do not have the ability to swap memory to and from disk. Thus, you must verify that the data you want to keep on the GPU does not exceed its memory limits, particularly when you are working with large matrices. By running `gpuDevice`, you can query your GPU card, obtaining information such as name, total memory, and available memory.

## Implementing and Accelerating the Algorithm to Solve a Wave Equation in MATLAB

To put the above example into context, let's implement the GPU functionality on a real problem. Our computational goal is to solve the second-order wave equation

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

with the condition u = 0 on the boundaries. We use an algorithm based on spectral methods to solve the equation in space and a second-order central finite difference method to solve the equation in time.

Spectral methods are commonly used to solve partial differential equations. With spectral methods, the solution is approximated as a linear combination of continuous basis functions, such as sines and cosines. In this case, we apply the Chebyshev spectral method, which uses Chebyshev polynomials as the basis functions.

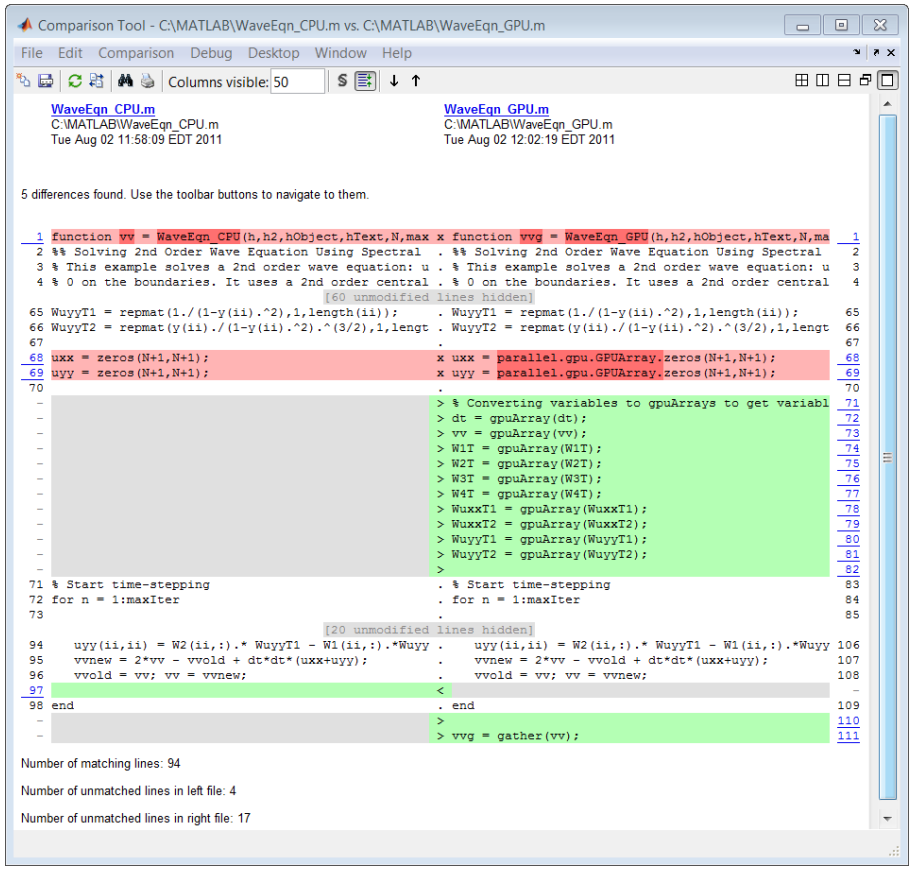



FIGURE 3. *Code Comparison Tool showing the differences in the CPU and GPU versions of the code. The GPU and CPU versions share over 84% of their code in common (94 lines out of 111).*

At every time step, we calculate the second derivative of the current solution in both the x and y dimensions using the Chebyshev spectral method. Using these derivatives together with the old solution and the current solution, we apply a second-order central difference method (also known as the leap-frog method) to calculate the new solution. We choose a time step that maintains the stability of this leap-frog method.

The MATLAB algorithm is computationally intensive, and as the number of elements in the grid over which we compute the solution grows, the time the algorithm takes to execute increases dramatically. When executed on a single CPU using a 2048 x 2048 grid, it takes more than a minute to complete just 50 time steps. Note that this time already includes the performance benefit of the inherent multithreading in MATLAB. Since R2007a, MATLAB supports multithreaded computation for a number of functions. These functions automatically execute on multiple threads without the need to explicitly specify commands to create threads in your code.

When considering how to accelerate this computation using Parallel Computing Toolbox, we will focus on the code that performs computations for each time step. Figure 3 illustrates the changes required to get the algorithm running on the GPU. Note that the computations involve MATLAB operations for which GPU-enabled overloaded functions are available through Parallel Computing Toolbox. These operations include FFT and

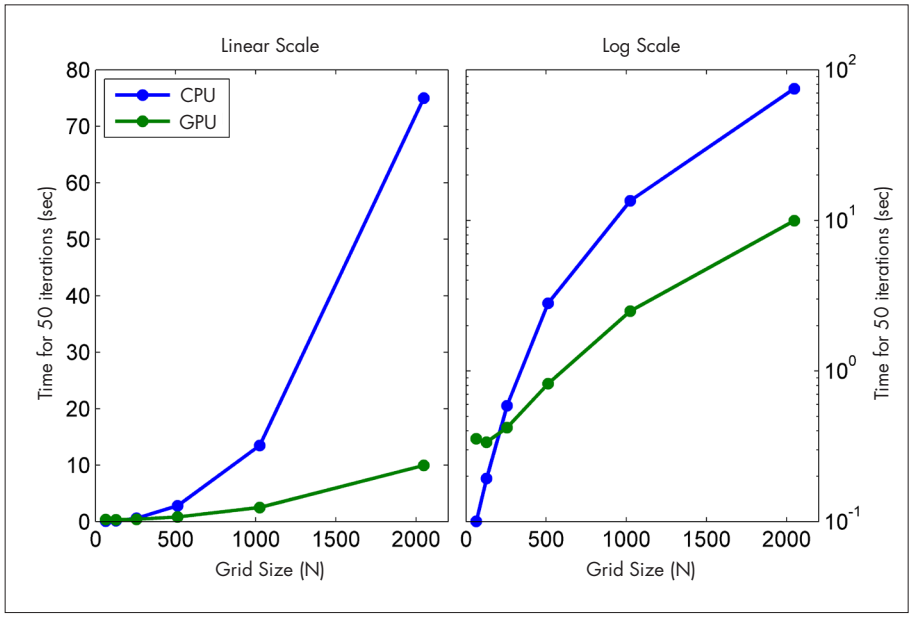



FIGURE 4. *Plot of benchmark results showing the time required to complete 50 time steps for different grid sizes, using either a linear scale (left) or a log scale (right).*

IFFT, matrix multiplication, and various element-wise operations. As a result, we do not need to change the algorithm in any way to execute it on a GPU. We simply transfer the data to the GPU using `gpuArray` before entering the loop that computes results at each time step.

After the computations are performed on the GPU, we transfer the results from the GPU to the CPU. Each variable referenced by the GPU-enabled functions must be created on the GPU or transferred to the GPU before it is used.

To convert one of the weights used for spectral differentiation to a GPUArray variable, we use

```
W1T = gpuArray(W1T);
```

Certain types of arrays can be constructed directly on the GPU without our having to transfer them from the MATLAB workspace. For example, to create a matrix of zeros directly on the GPU, we use

```
uxx = parallel.gpu.GPUArray.
zeros(N+1,N+1);
```

We use the `gather` function to bring data back from the GPU; for example:

```
vvg = gather(vv);
```

Note that there is a single transfer of data to the GPU, followed by a single transfer of data from the GPU. All the computations for each time step are performed on the GPU.

## Comparing CPU and GPU Execution Speeds

To evaluate the benefits of using the GPU to solve second-order wave equations, we ran a benchmark study in which we measured the amount of time the algorithm took to execute 50 time steps for grid sizes of 64, 128, 512, 1024, and 2048 on an Intel® Xeon® Processor X5650 and then using an NVIDIA® Tesla™ C2050 GPU.

For a grid size of 2048, the algorithm shows a 7.5x decrease in compute time from more than a minute on the CPU to less than 10 seconds on the GPU (Figure 4). The log scale plot shows that the CPU is actually faster for small grid sizes. As the technology evolves and matures, however, GPU solutions are increasingly able to handle smaller problems, a trend that we expect to continue.

## Summary

Engineers and scientists are successfully employing GPU technology, originally intended for accelerating graphics rendering, to accelerate their discipline-specific calculations. With minimal effort and without extensive knowledge of GPUs, you can now use the promising power of GPUs with MATLAB. GPUArrays and GPU-enabled MATLAB functions help you speed up MATLAB operations without low-level CUDA programming. If you are already familiar with programming for GPUs, MATLAB also lets you integrate your existing CUDA kernels into MATLAB applications without requiring any additional C programming.

To achieve speedups with the GPUs, your application must satisfy some criteria, among them the fact that sending the data between the CPU and GPU must take less time than the performance gained by running on the GPU. If your application satisfies these criteria, it is a good candidate for the range of GPU functionality available with MATLAB. ▪

[1]See Chapter 6 (Memory Optimization) of the NVIDIA "CUDA C Best Practices" documentation for further information about potential GPU-computing bottlenecks and optimization of GPU memory access.

[2]See Chapter 6 (Memory Optimization) of the NVIDIA "CUDA C Best Practices" documentation for further information about improving performance by minimizing data transfers.

## Learn More

**Introduction to MATLAB GPU Computing**
*mathworks.com/matlab-gpu-computing*

**Parallel Computing with MATLAB on Multicore Desktops and GPUs**
*mathworks.com/wbnr56334*

# Improving Simulink Simulation Performance

By Seth Popinchalk, MathWorks

Whatever the level of complexity of the model, every Simulink® user wants to improve simulation performance. This article shows how to make a model run faster or more efficiently. While every situation is unique, the techniques described here can be applied to a wide range of projects. Use them as a list of things to try whenever you need to increase the speed of your simulations.

## Selecting a Simulation Mode

In Simulink, three modes affect simulation performance: Normal, Accelerator, and Rapid Accelerator (Figure 1). As their names imply, Accelerator is faster than Normal, and Rapid Accelerator is faster still. Each increase in speed typically means sacrificing another capability—for example, flexibility, interactivity, or diagnostics. In many cases, if you can work without one of these capabilities—at least temporarily—simulation performance will improve.

## Accelerating the Initialization Phase

Large images and complex graphics take a long time to load and render. As a result, masked blocks that contain images might make your model less responsive. To accelerate the initialization phase of a simulation, remove complex drawings and images. If you don't want to remove an image, you can still improve performance by replacing it with a smaller, low-resolution version. To do this, use the Mask Editor and edit the icon drawing commands to change the image that is loaded by the call to `image()`.

When you update or open a model, Simulink runs the mask initialization code. If you have complicated mask initialization commands that contain many calls to `set_param`, consider consolidating consecutive calls to `set_param()` into a single call with multiple argument pairs. This can reduce the overhead associated with these calls.

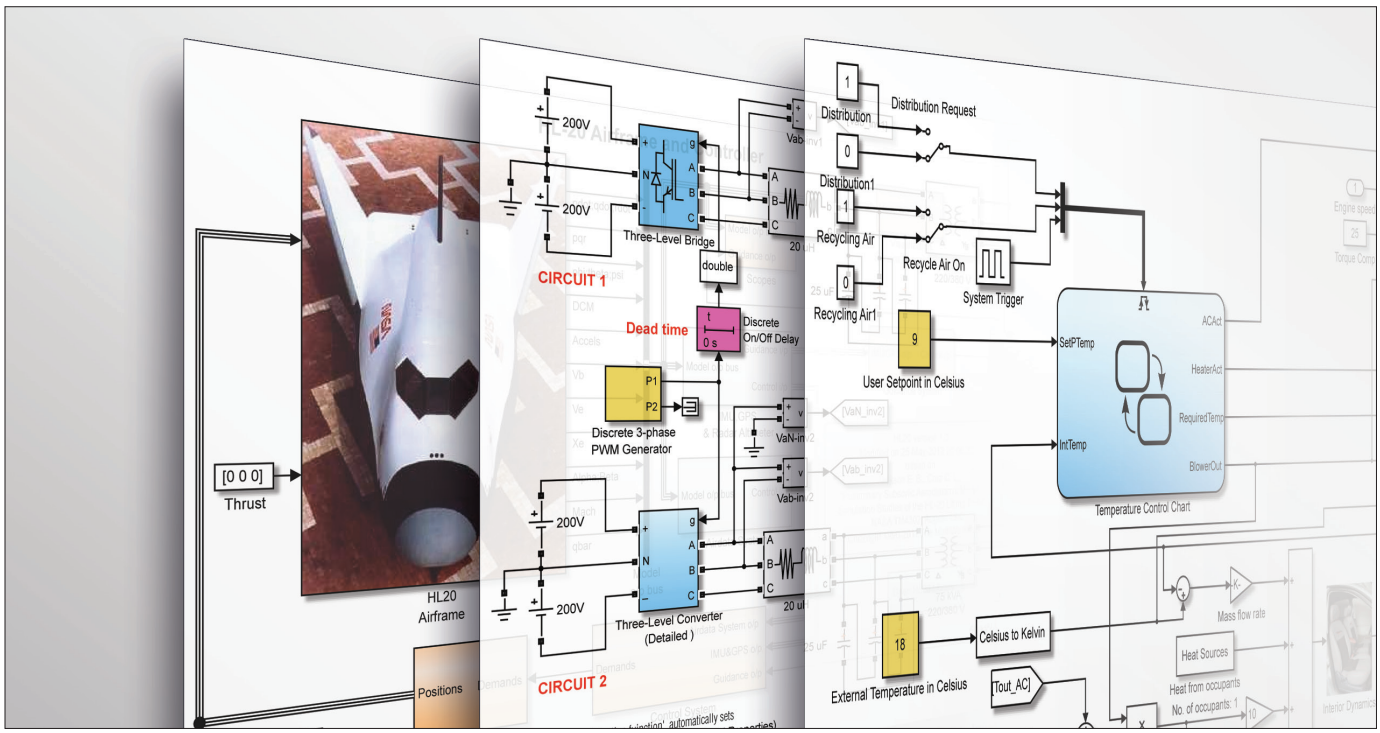
If you use MATLAB® scripts to load and initialize data, you can often improve performance by loading MAT-files instead. The drawback is that the data in a MAT-file is not in a human-readable form, and can therefore be more difficult to work with than a script. However, load typically initializes data much more quickly than the equivalent script.

## Reducing Interactivity

In general, the more interactive the model, the longer it will take to simulate. The tips in this section illustrate ways to improve performance by giving up some interactivity.

**Enable inline parameters optimization.** When you enable this optimization in the Optimization pane of the Configuration Parameters dialog box, Simulink uses the numerical values of model parameters instead of their symbolic names. This substitution can improve performance by reducing the parameter tuning computations performed during simulations.

**Disable debugging diagnostics.** Some enabled diagnostic features noticeably slow




*Simulation performance can be improved for models of every level of complexity.*

simulations. You can disable them in the Diagnostics pane of the Configuration Parameters dialog box.

**Disable MATLAB debugging and use BLAS library support.** After verifying that your MATLAB code works correctly, disable debugging support. In the Simulation Target pane of the Configuration Parameters dialog box, disable debugging/animation, overflow detection, and echoing expressions without semicolons.

If your simulation involves low-level MATLAB matrix operations, enable the Basic Linear Algebra Subprograms (BLAS) Library feature to make use of highly optimized external linear algebra routines.

**Disable Stateflow animations.** By default, Stateflow charts highlight the current active states and animate the state transitions that take place as the model runs. This feature is useful for debugging, but it slows the simulation. To accelerate simulations, either close all Stateflow charts or disable the animation. Similarly, if you're using Simulink 3D Animation™, SimMechanics™ visualization, FlightGear, or another 3D animation package, consider disabling the animation or reducing scene fidelity to improve performance.

**Adjust viewer-specific parameters and manage viewers through enabled subsystems.** If your model contains a scope viewer that displays a large number of data points and you can't eliminate the scope, try adjusting the viewer parameters to trade off fidelity for rendering speed. Be aware, however, that by using decimation to reduce the number of plotted data points, you risk missing short transients and other phenomena that would be obvious with more data points. You can place viewers in enabled subsystems to more precisely control which visualizations are enabled and when.

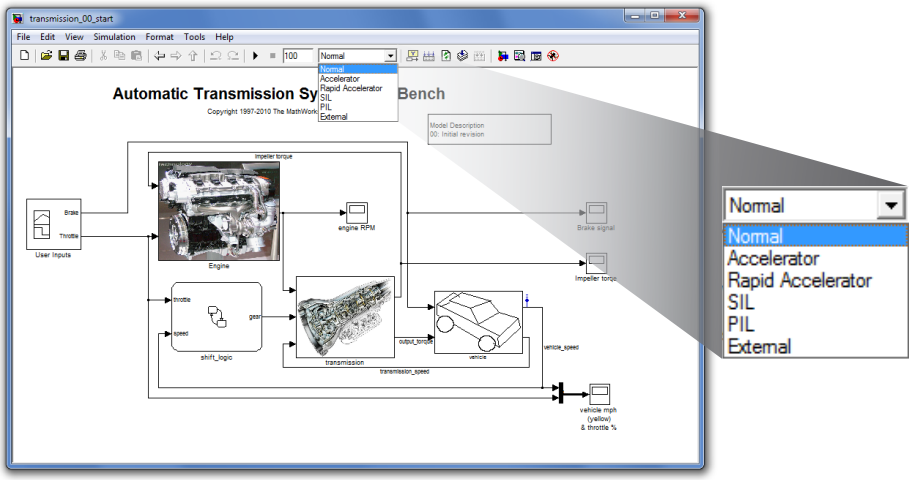



FIGURE 1. *Simulink simulation modes.*

## Modeling Made Easier with the New Simulink Editor

Simulink R2012b provides a new editor to simplify and accelerate modeling and simulation. Highlights include:

**Tabbed Navigation**
Reduce the number of open windows with tabs and automatic window reuse.

**Smart Signal Routing**
Connect one block to another and let Simulink determine the simplest signal line path without overlapping blocks and text.

**Simulation Rewind**
Step backwards and forwards through your simulation to facilitate model analysis.

**Explorer Bar to Navigate Model**
View the current level of model hierarchy and navigate through that hierarchy in one tool.

**Masked Subsystem Badges**
Identify masked subsystems and look under masks with one click.

**Menu Organized for Your Workflow**
Access menu items arranged to match the typical design workflow: design, analysis, code generation, and testing.

**Learn More About Simulink R2012b**
*mathworks.com/simulink-new-features*

## Reducing Model Complexity

Simplifying your model without sacrificing fidelity is an effective way to improve simulation performance. Here are three ways to reduce model complexity.

**Replace a subsystem with a lower-fidelity alternative.** In many cases, you can simplify your model by replacing a complex subsystem model with one of the following:

- A linear or nonlinear dynamic model created from measured input-output data using System Identification Toolbox™
- A high-fidelity, nonlinear statistical model created using Model-Based Calibration Toolbox™
- A linear model created using Simulink Control Design™
- A lookup table

You can maintain both representations of the subsystem in a library and use variant subsystems to manage them.

**Reduce the number of blocks.** When you reduce the number of blocks in your model, fewer blocks will need to be updated during simulations, leading to faster simulation runs. Vectorization is one way to reduce your block count. For example, if you have several parallel signals that undergo a similar set of computations, try combining them into a vector and performing a single computation. Another way is to simply enable the Block Reduction optimization in the Optimization > General section of the configuration parameters.

**Use frame-based processing.** In frame-based processing, samples are processed in batches instead of one at a time. If your model includes an analog-to-digital converter, for example, you can collect the output samples in a buffer and process the buffer with a single operation, such as a fast Fourier transform. Processing data in chunks in this way reduces the number of times that blocks in your model must be invoked. In general, scheduling overhead decreases as frame size increases. However, larger frames consume more memory, and memory limitations can adversely affect the performance of complex models. Experiment with different frame sizes to find one that maximizes the performance benefit of frame-based processing without causing memory issues.

## Choosing and Configuring a Solver

Simulink provides a comprehensive library of solvers, including fixed-step and variable-step solvers to handle stiff and nonstiff systems. Each solver determines the time of the next simulation step and applies a numerical method to solve ordinary differential equations that represent the model. The solver you choose and the solver options you specify will affect simulation speed.

**Select a solver that matches the stiffness of your system.** A stiff system has both slowly and quickly varying continuous dynamics. *Implicit solvers* are specifically designed for stiff problems, whereas explicit solvers are designed for nonstiff problems. Using nonstiff solvers to solve stiff systems is inefficient and can lead to incorrect results. If a nonstiff solver uses a very small step size to solve your model, it may be because you have a stiff system.

**Choose a variable-step or fixed-step solver based on your model's step size and dynamics.** Exercise caution when deciding whether to use a variable-step or fixed-step solver; otherwise, your solver could take additional time steps to capture dynamics that are not important to you, or it could perform unnecessary calculations to work out the next time step.

In general, simulations run with variable-step solvers are faster than those run with fixed-step solvers: You use fixed-step solvers when the step size is less than or equal to the fundamental sample time of the model. With a variable-step solver, the step size can vary because variable-step solvers dynamically adjust the step size. As a result, the step size for some time steps is larger than the fundamental sample time, reducing the number of steps required to complete the simulation.

As a rule, choose a fixed-step solver when the fundamental sample time of your model is equal to one of the sample rates. Choose a variable-step solver to capture continuous dynamics, or when the fundamental sample time of your model is less than the fastest sample rate.

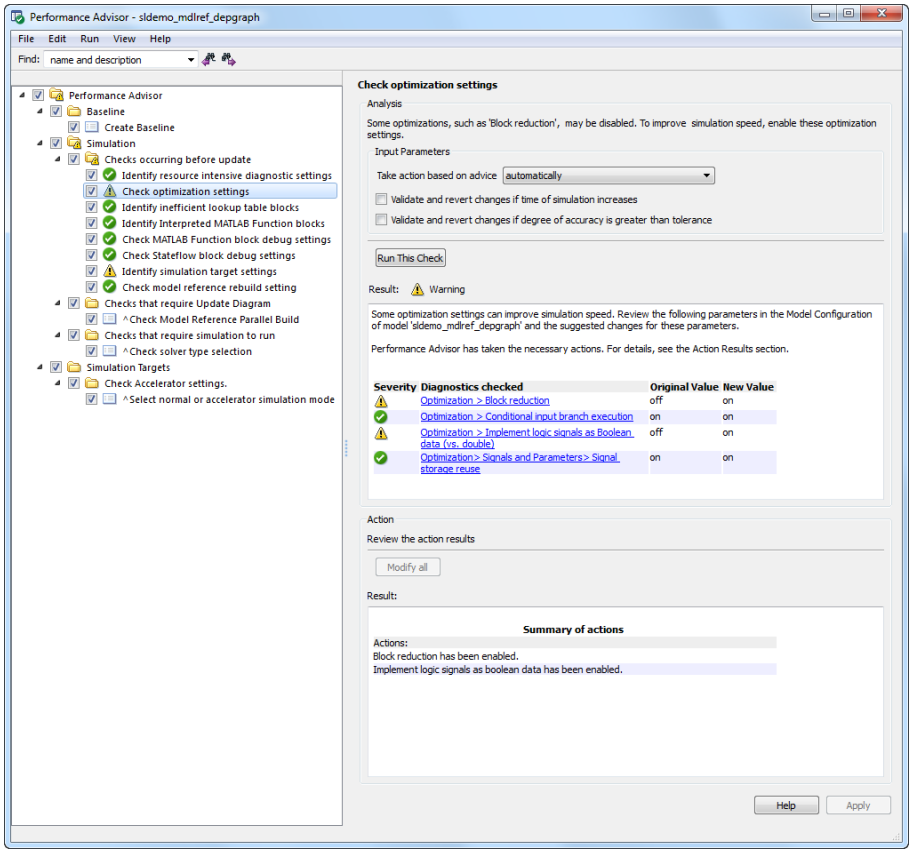



FIGURE 2. *The Simulink Performance Advisor.*

**Decrease the solver order.** Decreasing the solver order improves simulation speed because it reduces the number of calculations that Simulink performs to determine state outputs. Of course, the results become less accurate as the order of the solver decreases. The goal is to choose the lowest solver order that will produce results that meet your accuracy requirements.

**Increase the solver step size or the error tolerance.** Similarly, increasing the solver step size or increasing the solver error tolerance usually increases simulation speed at the expense of accuracy. Such changes should be made with care because they can cause Simulink to miss potentially important dynamics during simulations.

**Disable zero-crossing detection.** Variable-step solvers dynamically adjust the step size, increasing it when a variable changes slowly and decreasing it when a variable changes rapidly. This behavior causes the solver to take many small steps in the vicinity of a discontinuity because this is when a variable is rapidly changing. Accuracy improves, but it often comes with long simulation times.

To avoid the small time steps and long simulations associated with these situations, Simulink uses zero-crossing detection to accurately locate such discontinuities. For systems that exhibit frequent fluctuation betweens modes of operation—a phenomenon known as *chattering*—this zero-crossing detection can actually have the opposite effect and slow simulations. In these situations, it may be possible to adjust zero-crossing detection to improve performance.

## Saving the Simulation State

Engineers typically simulate a Simulink model repeatedly for different inputs, boundary conditions, and operating conditions. In many situations, these simulations share a common startup phase in which the model transitions from its initial state to some other state. An electric motor, for example, may be brought up to speed before various control sequences are tested.

Using the Simulink SimState feature, you can save the simulation state at the end of the startup phase and then restore it for use as the initial state for future simulations. This technique does not improve simulation speed per se, but it can reduce total simulation time for consecutive runs because the startup phase needs to be simulated only once.

## Simulink Performance Advisor

Starting with Simulink R2012b, you can use the Performance Advisor to check for conditions and configuration settings that might cause inefficient simulation performance (Figure 2). The Performance Advisor analyzes the model and produces a report that lists the suboptimal conditions or settings that it finds. It suggests better model configuration settings where appropriate, and provides mechanisms for fixing issues automatically or manually. The techniques recommended in this article, as well as other approaches, can be automatically tested in the Performance Advisor. ▪

## Learn More

**Guy and Seth on Simulink**
*http://blogs.mathworks.com/seth*

**Video: Speeding Up Simulink for Control System Applications**
*mathworks.com/speedsl-061710*

**Video: Speeding Up Simulink for Signal Processing Applications**
*mathworks.com/speedslspc-062210*

# Simulating Blackjack with MATLAB

By Cleve Moler, MathWorks

Blackjack is the most popular casino game in the world. Using a basic strategy, a knowledgeable player can reduce the casino's advantage to less than one-half of one percent. Simulating blackjack play with this strategy in MATLAB® is both an instructive programming exercise and a useful parallel computing benchmark.

Blackjack is also known as "21." The object is to get a hand with a value close to, but not more than, 21. Face cards are worth 10 points, aces are worth either 1 or 11, and all other cards are worth their numerical value. You play against the dealer. You each start with two cards. Your cards are dealt face up; one of the dealer's cards stays face down.

You signal "hit" to receive additional cards. When you are satisfied with your hand, you "stand." The dealer then reveals the hidden card and finishes the hand. If your total ever exceeds 21, you are "bust," and the dealer wins the hand without drawing any more cards.

The dealer has no choices to make, and must draw on hands worth less than 17 and stand on hands worth 17 or more. (A variant has the dealer draw to a "soft" 17, which is a hand with an ace counting 11.) If neither player goes bust, then the hand closest to 21 wins. Equal totals are a "push," and neither wins.

The fact that you can bust before the dealer takes any cards is a disadvantage that would be overwhelming were it not for three additional features of the game. On your first two cards:

- An ace and a face card or a 10 is a "blackjack," which pays 1.5 times the bet if the dealer does not also have 21
- You can "split" a pair into two separate hands by doubling the bet
- You can "double down" a good hand by doubling the bet and receiving just one more card

## Basic Strategy

Basic strategy was first described in the 1956 paper "The Optimum Strategy in Blackjack," published in the *Journal of the American Statistical Association* by four authors from the Aberdeen Proving Ground. It is now presented, with a few variations, on dozens of web pages, including Wikipedia. The strategy assumes that you do not retain information from earlier hands. Your play depends only on your current hand and the dealer's up card. With basic strategy, the house advantage is only about one half of one percent of the original bet.

My MATLAB programs, shown in the sidebar, use three functions to implement basic stategy. The function `hard` uses the array `HARD` to guide the play of most hands. The row index into `HARD` is the current total score, and the column index is the value of the dealer's up card. The return value is 0, 1, or 2, indicating "stand," "hit," or "double down." The other two functions, `soft` and `pair`, play similar roles for hands containing an ace worth 11 and hands containing a pair.

The most important consideration in basic strategy is to avoid going bust when the dealer has a chance of going bust. In our functions, the subarray `HARD(12:16,2:6)` is nearly all zero. This represents the situation where both you and the dealer have bad hands—your total and the dealer's expected total are each less than 17. You are tempted to hit, but you might bust, so you stand. The dealer will have to hit, and might bust. This is your best defense against the house advantage. With naïve play, which ignores the dealer's up card, you would almost certainly hit a 12 or 13. But if the dealer is also showing a low card, stand on your low total and wait to see if the dealer goes over 21.

## Card Counting

Card counting was introduced in 1962 in *Beat the Dealer*, a hugely popular book by Edward Thorp. If the deck is not reshuffled after every hand, you can keep track of, for example, the number of aces, face cards, and nonface cards that you have seen. As you approach the end of the deck, you may know that it is "ten rich"—the number of aces and face cards remaining is higher than would be expected in a freshly shuffled deck. This situation is to your advantage because you have a better than usual chance of getting a blackjack and the dealer has a better than usual chance of going bust. So you increase your bet and adjust your strategy.

Card counting gives you a mathematical advantage over the casino. Exactly how much of an advantage depends upon how much you are able to remember about the cards you have seen. Thorp's book was followed by a number of other books that simplified and popularized various systems. My personal interest in blackjack began with a 1973 book by John Archer. But I can attest that card counting is boring, error-prone, and not very lucrative. It is nowhere near

### MATLAB Functions for Basic Blackjack Strategy

```
function strat = hard(p,d)
   % Hands without aces.
   % hard(player,dealer)
   % 0 = stand
   % 1 = hit
   % 2 = double down
   % Dealer shows:
   %      2 3 4 5 6 7 8 9 T A
   HARD = [ ...
      1   x x x x x x x x x x
      2   1 1 1 1 1 1 1 1 1 1
      3   1 1 1 1 1 1 1 1 1 1
      4   1 1 1 1 1 1 1 1 1 1
      5   1 1 1 1 1 1 1 1 1 1
      6   1 1 1 1 1 1 1 1 1 1
      7   1 1 1 1 1 1 1 1 1 1
      8   1 1 1 1 1 1 1 1 1 1
      9   2 2 2 2 2 1 1 1 1 1
     10   2 2 2 2 2 2 2 2 1 1
     11   2 2 2 2 2 2 2 2 2 2
     12   1 1 0 0 0 1 1 1 1 1
     13   0 0 0 0 0 1 1 1 1 1
     14   0 0 0 0 0 1 1 1 1 1
     15   0 0 0 0 0 1 1 1 1 1
     16   0 0 0 0 0 1 1 1 1 1
     17   0 0 0 0 0 0 0 0 0 0
     18   0 0 0 0 0 0 0 0 0 0
     19   0 0 0 0 0 0 0 0 0 0
     20   0 0 0 0 0 0 0 0 0 0];
   strat = HARD(p,d);
end
```

```
function strat = soft(p,d)
   % Hands with aces.
   % soft(player-11,dealer)
   % 0 = stand
   % 1 = hit
   % 2 = double down
   % Dealer shows:
   %      2 3 4 5 6 7 8 9 T A
   SOFT = [ ...
      1   1 1 1 1 1 1 1 1 1 1
      2   1 1 2 2 2 1 1 1 1 1
      3   1 1 2 2 2 1 1 1 1 1
      4   1 1 2 2 2 1 1 1 1 1
      5   1 1 2 2 2 1 1 1 1 1
      6   2 2 2 2 2 1 1 1 1 1
      7   0 2 2 2 2 0 0 1 1 0
      8   0 0 0 0 0 0 0 0 0 0
      9   0 0 0 0 0 0 0 0 0 0];
   strat = SOFT(p,d);
end
```

```
function strat = pair(p,d)
   % Strategy for splitting pairs.
   % pair(paired,dealer)
   % 0 = keep pair
   % 1 = split pair
   % Dealer shows:
   %      2 3 4 5 6 7 8 9 T A
   PAIR = [ ...
      1   x x x x x x x x x x
      2   1 1 1 1 1 1 0 0 0 0
      3   1 1 1 1 1 1 0 0 0 0
      4   0 0 0 1 0 0 0 0 0 0
      5   0 0 0 0 0 0 0 0 0 0
      6   1 1 1 1 1 1 0 0 0 0
      7   1 1 1 1 1 1 1 0 0 0
      8   1 1 1 1 1 1 1 1 1 1
      9   1 1 1 1 1 0 1 1 0 0
     10   0 0 0 0 0 0 0 0 0 0
     11   1 1 1 1 1 1 1 1 1 1];
   strat = PAIR(p,d);
end
x = NaN; % Not possible
```

as glamorous—or as dangerous—as the recent Hollywood film "21" portrays it. And many venues now have machines that continuously shuffle the cards after each hand, making card counting impossible.

## The MATLAB Simulations

My original MATLAB program, written several years ago, had a persistent array that is initialized with a random permutation of several copies of the vector `1:52`. These integers represent both the values and the suits in a 52-card deck. The suit is irrelevant in the play, but is nice to have in the display. Cards are dealt from the end of the deck, and the deck is reshuffled when there are just a few cards left.

```
function card = dealcard
% Deal one card
persistent deck
if length(deck) < 10
   % Four decks
   deck = [1:52 1:52 1:52 1:52];
   % Shuffle
   deck = deck(randperm(length(deck)));
end
card = deck(end);
deck(end) = [];
```

This function faithfully simulates a blackjack game with four decks dealt without reshuffling between hands. It would be possible to count cards, but this shuffler has two defects: It does not simulate a modern shuffling machine, and the persistent array prevents some kinds of parallelization.

My most recent simulated shuffler is much simpler. It creates an idealized mechanical shuffler that has an infinite number of perfectly mixed decks. It is not possible to count cards with this shuffler.

```
function card = dealcard
% Deal one card
card = ceil(52*rand);
```

I have two blackjack programs, both available on MATLAB Central. One program offers an interface that lets you play one hand at a time. Basic strategy is highlighted, but you can make other choices. For example, Figures 1 and 2 show the play of an infrequent but lucrative hand. You bet $10 and are dealt a pair of 8s. The dealer's up card is a 4. Basic strategy recommends splitting the pair of 8s. This increases the bet to $20. The first hand is then dealt a 3 to add to the 8, giving 11. Basic strategy recommends always doubling down on 11. This increases the total bet to $30. The next card is a king, giving the first hand 21. The second hand is dealt a 5 to go with the 8, giving it 13. You might be tempted to hit the 13, but the dealer is showing a 4, so you stand. The dealer reveals a 10, and has to hit the 14. The final card is a jack, busting the dealer and giving you $30. This kind of hand is rare, but gratifying to play correctly.

My second program plays a large number of hands using basic strategy and collects statistics about the outcome.

## Accelerating the Simulations with Parallel Computing

I like to demonstrate parallel computing with MATLAB by running several copies of my blackjack simulator simultaneously using Parallel Computing Toolbox™. Here is the main program:

```
% BLACKJACKDEMO Parallel blackjack demo.
matlabpool
p = 4;      % Number of players.
n = 25000;  % Number of hands per player.
B = zeros(n,p);
parfor k = 1:p
   B(:,k) = blackjacksim(n);
end
plot(cumsum(B))
r = sum(B(n,1:p));
fmt = '%d x %d hands, total return = $ %d'
title(sprintf(fmt,p,n,r))
```

The `matlabpool` command starts up many *workers* (copies of MATLAB) on the cores or processors available in a multicore machine or a cluster. These workers are also known as *labs*. The random number generators on each lab are initialized to produce statistically independent streams drawn from a single overall global stream. The main program on the master MATLAB creates an array `B`, and then the `parfor` loop runs a separate instance of the sequential simulator, `blackjacksim`, on each lab. The results are communicated to the master and stored in the columns of `B`. The master can then use `B` to produce the plot shown in Figure 3. With "only" 25,000 hands for each player, the simulation is still too short to show the long-term trend. The computation time is about 11 seconds on my dual-core laptop. If I do not turn on the MATLAB pool, the computation uses only one core and takes almost 20 seconds.

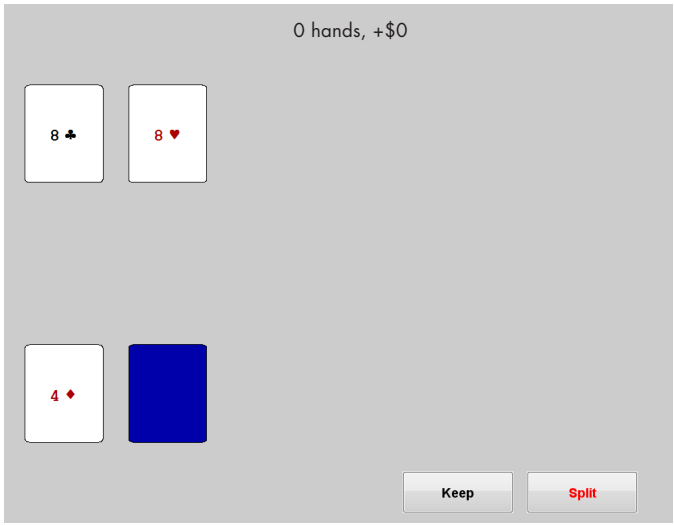



FIGURE 1. *Start of an atypical but important example: You are dealt a pair of 8s, and the dealer's up card is a 4. Basic strategy, highlighted in red, recommends splitting the pair.*

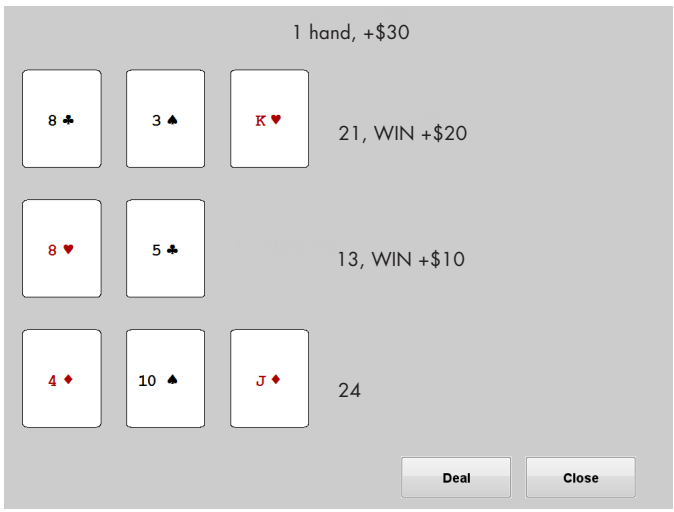



FIGURE 2. *The final outcome. After splitting, you double down on your first hand and stand on your second. The dealer goes bust, giving you a rare 3x win.*

This run can also produce the histograms shown in Figure 4. The cumulative return from the four players is the dot product of the two vectors annotating the horizontal axis. The discrepancy between the frequency of $10 wins and $10 losses is almost completely offset by the higher frequencies of the larger wins over the larger losses. The average

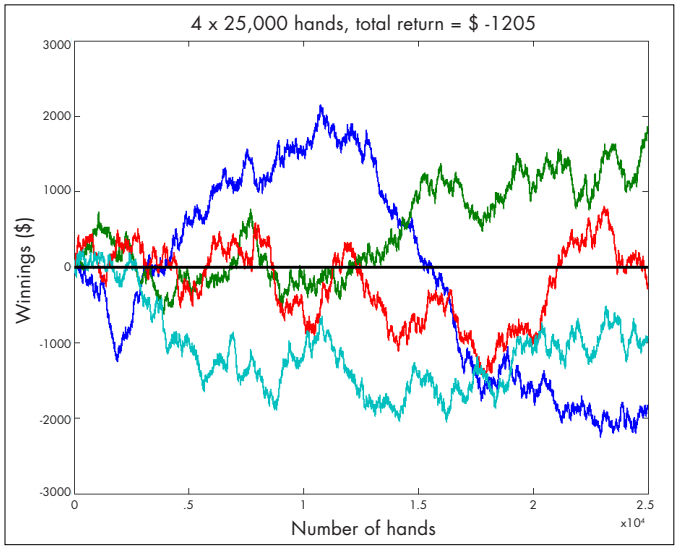



FIGURE 3. *Four players in a parallel simulation. Green wins, red nearly breaks even, cyan muddles through, and blue should have quit while he was ahead.*

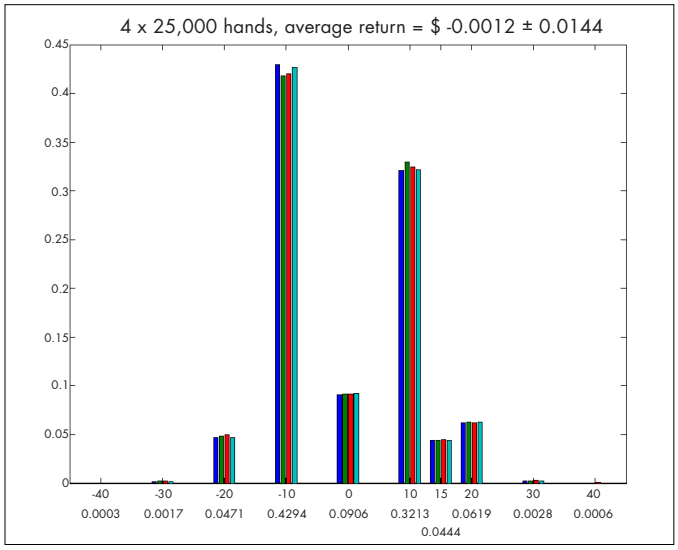



FIGURE 4. *Histograms for each player. The $10 bet is a push 9% of the time, a $10 win 32%, and a $10 loss 42%. Blackjacks yield $15 wins 4.5% of the time. The less frequent $20, $30, and $40 swings come from doubling down, splitting pairs, and doubling after splitting.*

return and a measure of its variation are shown in the title of the figure. We see that in runs of this length, the randomness of the shuffle still dominates. It would require longer runs with millions of hands to be sure that the expected return is slightly negative.

Blackjack can be a surrogate for more sophisticated financial instruments. The first graph in Figure 5 shows the performance of the Standard & Poor's stock market index during 2011. The second shows the performance of our blackjack simulation playing 100 hands a day for

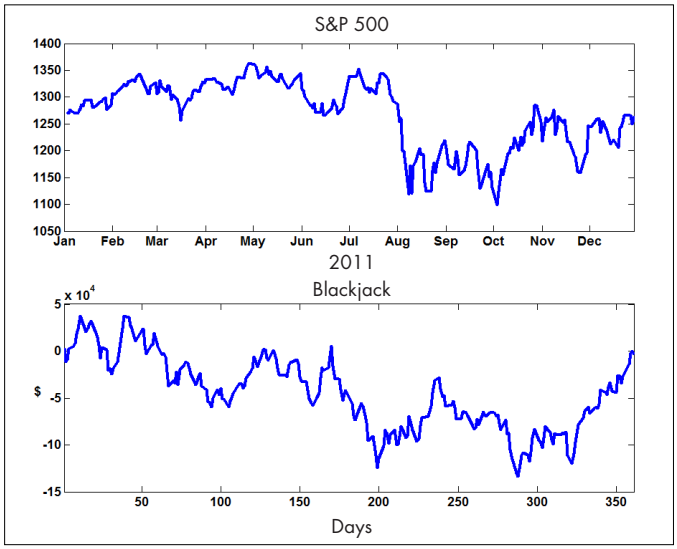



FIGURE 5. *The Standard & Poor's stockmarket index over one year versus 100 hands of blackjack per day for the same time. Short-term random flucuations dominate any long-term trend.*

each of the 252 days the stock market was open that year. The S&P dropped 14.27 points. Our blackjack simulation, which bet $10 per hand, lost $3860 over the same period. More important than these final results is the fact that both instruments show large fluctuations about their mean behavior. Over the short term, stock market daily behavior and card shuffling luck are more influential than any long-term trend. ▪

## References

Baldwin, R. R., Cantey, W. E., Maisel, H., and McDermot, J. P. "The Optimum Strategy in Blackjack," *Journal of the American Statistical Association*, 51:429-439 (Sept.), 1956.

Thorp, E. O. *Beat the Dealer*. New York: Random House, 1962.

Wikipedia. http://en.wikipedia.org/wiki/Blackjack

## Learn More

**Cleve's Corner Blog**
*http://blogs.mathworks.com/cleve*

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

**Download: MATLAB Programs for Simulating Blackjack**
*mathworks.com/simulating-blackjack*

# Writing MATLAB Functions with Flexible Calling Syntax

MATLAB® functions often have flexible calling syntax with required inputs, optional inputs, and name-value pairs. While this flexibility is convenient for the end user, it can mean a lot of work for the programmer who must implement the input handling. You can greatly reduce the amount of code needed to handle input arguments by using the `inputParser` object.

Our goal is to define a function with the following calling syntax:

```
function a = findArea(width,varargin)
% findArea(width)
% findArea(width,height)
% findArea(... 'shape',shape)
```

With `inputParser` you can specify which input arguments are required (`width`), which are optional (`height`), and which are optional name-value pairs (`'shape'`). `inputParser` also lets you confirm that each input is valid—for instance, that it is the right size, shape, or data type. Finally, `inputParser` lets you specify default values for any optional inputs.

`inputParser` is a MATLAB object. To use it, you first create an object and then call functions to add the various input arguments. Let's start by adding the required input argument, `width`:

```
p = inputParser;
addRequired(p,'width');
```

Our input parser ensures that the user has specified at least one input argument. We want to go one step further—to make sure that the user entered a numeric value. We include a validation function, which is a handle to the built-in function `isnumeric`:

```
p = inputParser;
addRequired(p,'width',@isnumeric);
```

The input parser will now generate an error if given a value for width that is not numeric.

When we add the optional height input, we also include a default value:

```
defaultHeight = 1;
addOptional(p,'height',defaultHeight,@isnumeric);
```

Adding support for the `'shape'` name-value pair is trickier, since we must make sure that the user entered either `'square'` or `'rectangle'`. We create a custom anonymous function that returns true if the input string matches, and false if it does not:

```
defaultShape = 'rectangle';
checkString = @(s)...
    any(strcmps,{'square','rectangle'});
addParamValue(p,'shape',defaultShape,checkString);
```

Now that we have defined our input parsing rules, we are ready to parse the inputs. We pass the function inputs to the `parse` function of the `inputParser`, using `{:}` to convert the input cell array `varargin` to a comma-separated list of input arguments. We get the validated user input values from the `Results` structure in our `inputParser` to use in the rest of the function:

```
parse(p,width,varargin{:});
width  = p.Results.width;
height = p.Results.height;
shape  = p.Results.shape;
```

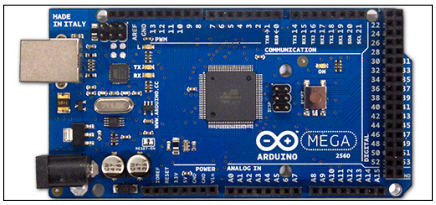
To see how the code snippets in this article fit together, read the input validation example in the `inputParser` documentation. You'll then be ready to use `inputParser` to offer flexible calling syntax in your own MATLAB functions. ▪

## Learn More

**inputParser Documentation**
*mathworks.com/inputparser*

# Project-Based Learning

Project-based learning uses active learning techniques and gives students direct exposure to hardware and software. By extending the approach to incorporate industry-standard software such as MATLAB and Simulink, instructors not only keep students motivated but also prepare them for a range of careers. Simulink enables these goals with built-in support for interfacing with low-cost hardware, including Arduino®, BeagleBoard, and LEGO® MINDSTORMS® NXT platforms. This built-in support is also available in MATLAB and Simulink Student Version.




## Mechatronics and Controls

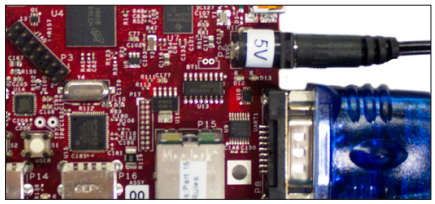
Arduino, when used with Simulink, enables students to develop algorithms that interface with a variety of peripherals and electronics for mechatronics and control systems. Students can develop software in Simulink, implement algorithms as standalone applications, and tune parameters while the application is running.

**Interactive Tutorial:** Control Systems
*mathworks.com/control-systems-tutorial*


**Control Systems Engineering, 6e**
by Norman S. Nise, John Wiley & Sons, Inc.

**Advanced Mechatronics: Monitoring and Control of Spatially Distributed Systems**
by Dan Necsulescu, World Scientific Publishing Co.

**Basic Engineering Circuit Analysis, 9e**
by J. David Irwin and R. Mark Nelms, John Wiley & Sons, Inc.



## Signal Processing

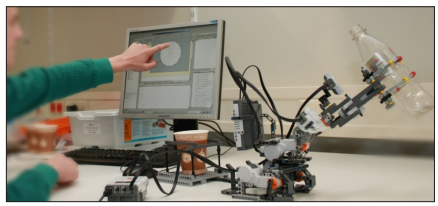
BeagleBoard provides a low-cost platform for teaching audio processing and computer vision applications. By using Simulink with BeagleBoard, students can verify their algorithms during simulation and then implement them as standalone applications.

**Interactive Tutorial:** Signal Processing
*mathworks.com/signal-processing-tutorial*


**Practical Image and Video Processing Using MATLAB**
By Oge Marques, John Wiley & Sons, Inc.

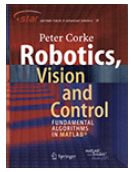
**Digital Signal Processing: A Computer-Based Approach, 4e**
By Sanjit K. Mitra, McGraw-Hill

**Multidimensional Signal, Image, and Video Processing and Coding, 2e**
By John W. Woods, Academic Press



## Robotics

LEGO MINDSTORMS NXT and Simulink offer an environment that lets students model, simulate, and design algorithms for a range of robotics applications. The platform is also widely used for teaching programming.




**Robotics, Vision and Control: Fundamental Algorithms in MATLAB**
By Peter Corke, Springer

**Mobile Robots: Navigation, Control and Remote Sensing**
By Gerald Cook, John Wiley & Sons, Inc.

**Simulation of Dynamic Systems with MATLAB and Simulink, 2e**
By Harold Klee and Randal Allen, CRC Press, Inc.

## Learn More

**Hardware for Project-Based Learning**
*mathworks.com/hardware-resources*

**Video: Enabling Project-Based Learning with MATLAB, Simulink, and Target Hardware**
*mathworks.com/enabling-project-based-learning*

**Academic Resources**
*mathworks.com/academia*

# Model-Based Design for Renewable Energy Systems

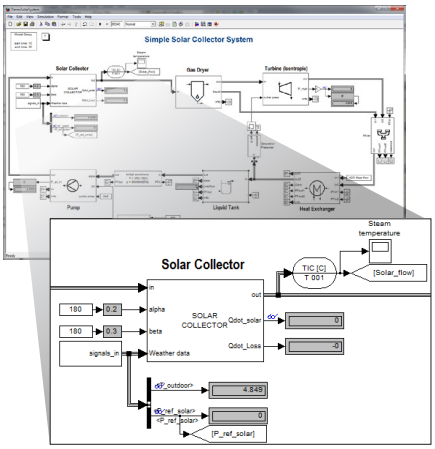
Third-party solutions enhance Model-Based Design for system engineering and embedded-system development in renewable energy applications. Libraries of specialized components enable engineers to model and simulate solar and fuel cell systems, conduct trade studies, and specify system architectures. Embedded targets let them generate code from Simulink® models for the PLC systems and microcontrollers used to control wind turbines, solar power inverters, and hydrogen fuel cell systems.

## Autonomie

Autonomie is a development environment for powertrain and vehicle modeling that supports the rapid evaluation of new technologies for improving fuel economy. It includes Simulink and Stateflow® models of electric, hybrid electric, plug-in hybrid, fuel cell, and other advanced powertrains. Autonomie models provide multiple levels of fidelity and abstraction, enabling the simulation of subsystems, systems, or entire vehicles.

**Argonne National Laboratory**

*www.autonomie.net*

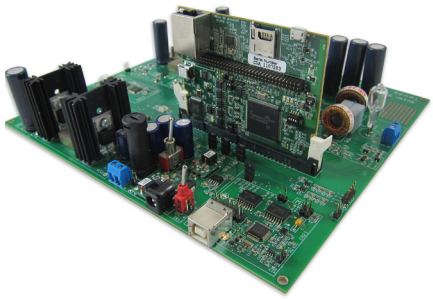



## Thermolib

Thermolib provides a set of Simulink blocks for modeling and simulating thermodynamic and thermochemical systems, including traditional and solar power plants, fuel cell vehicles, and HVAC systems. These building blocks enable engineers to create user-defined components or systems and simulate them within Simulink. Thermolib provides demos for common combustion processes, heat pumps and refrigeration cycles, fuel cells, gas turbines, and battery thermal management systems.

**EUtech Scientific Engineering GmbH**

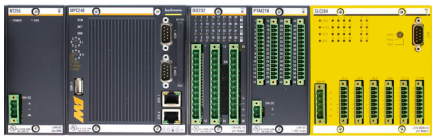
*www.eutech-scientific.de*



## C2000 MCU Family

Texas Instruments C2000™ MCU family includes integraed peripherals appropriate for solar power inverter control applications. Typical inverter control algorithms are based on reading solar module voltages through on-chip ADCs and controlling AC power output through high-resolution PWM outputs. TI offers Renewable Energy Developer's Kits containing C2000-based DC/AC inverters, as well as integrated development tools, including Code Composer Studio™ (CCS). Engineers can generate C and C++ code directly from their algorithms in Simulink and compile it with CCS using target I/O blocks for C2000 peripherals provided with Embedded Coder™.

**Texas Instruments** *www.ti.com/solar*



## M-Target for Simulink

Integrated into Bachmann's M1 automation systems, M-Target for Simulink provides blocksets of M1 I/O cards, as well as variable and parameter interchange blocks for developing advanced controls for wind turbine systems. Designed as a target for Simulink Coder™, it lets engineers automatically generate code and deploy their Simulink models directly to M1 controllers in hardware-in-the-loop systems or actual plants.

**Bachmann electronic GmbH**

*www.bachmann.info*

### Learn More

**Third-Party Products and Services**
*mathworks.com/connections*

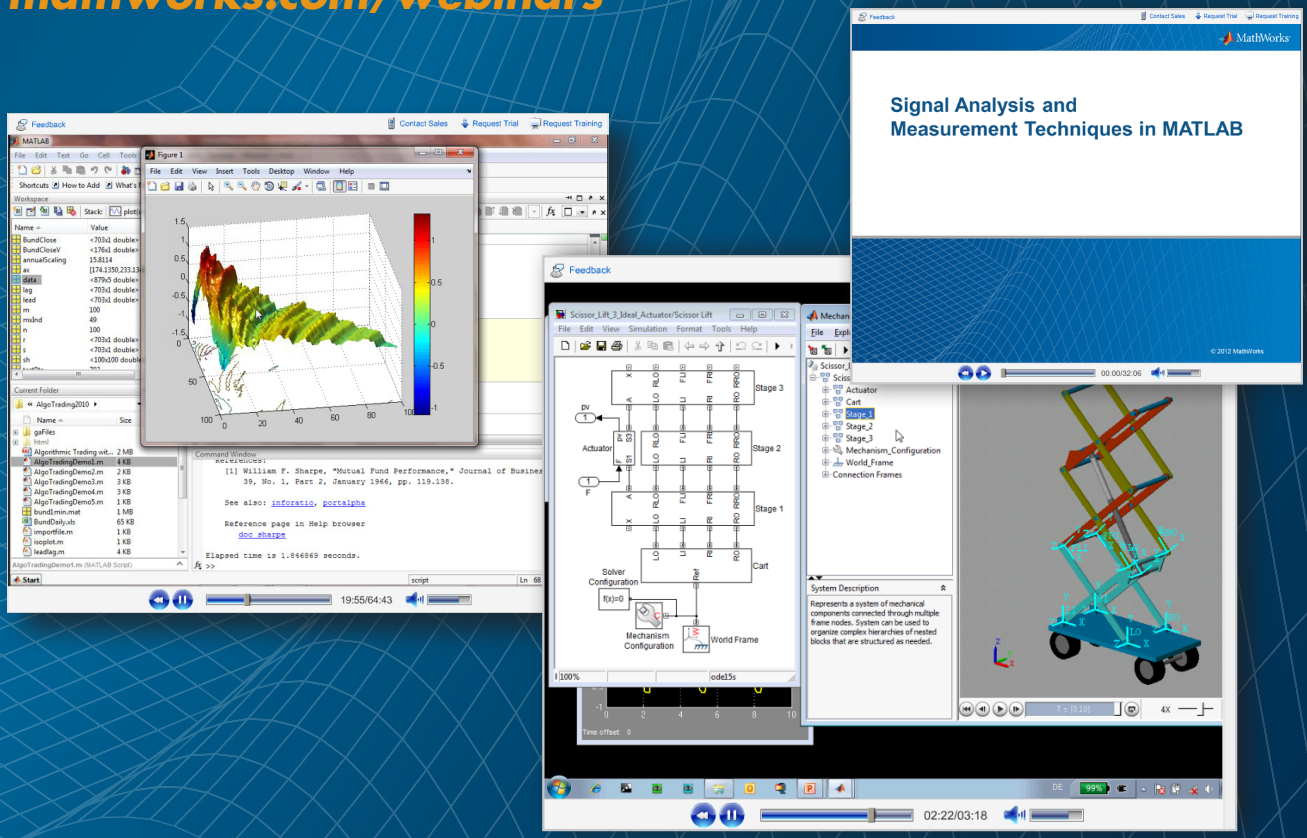
# Watch Live and Recorded Webinars in 19 Languages

## Technical tips and examples at your desk

Explore more than 300 topics, including:

- Data acquisition
- Data analysis
- Mathematical modeling
- Algorithm development
- GPU and parallel computing
- Desktop and web deployment
- System design and simulation
- Plant modeling and control design
- Discrete-event simulation
- Rapid prototyping
- Embedded code generation
- HDL code generation and verification
- Verification, validation, and test

***mathworks.com/webinars***



©2012 MathWorks, Inc.

MathWorks®
*Accelerating the pace of engineering and science*

# DISCOVER THE NEW LOOK AND FEEL *of* Simulink

TRY IT TODAY
visit mathworks.com/simulink-new-features


Starting with Simulink® Release 2012b, it's even easier to build, manage, and navigate your Simulink and Stateflow® models:

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

MATLAB® & SIMULINK®

92012v00 10/12