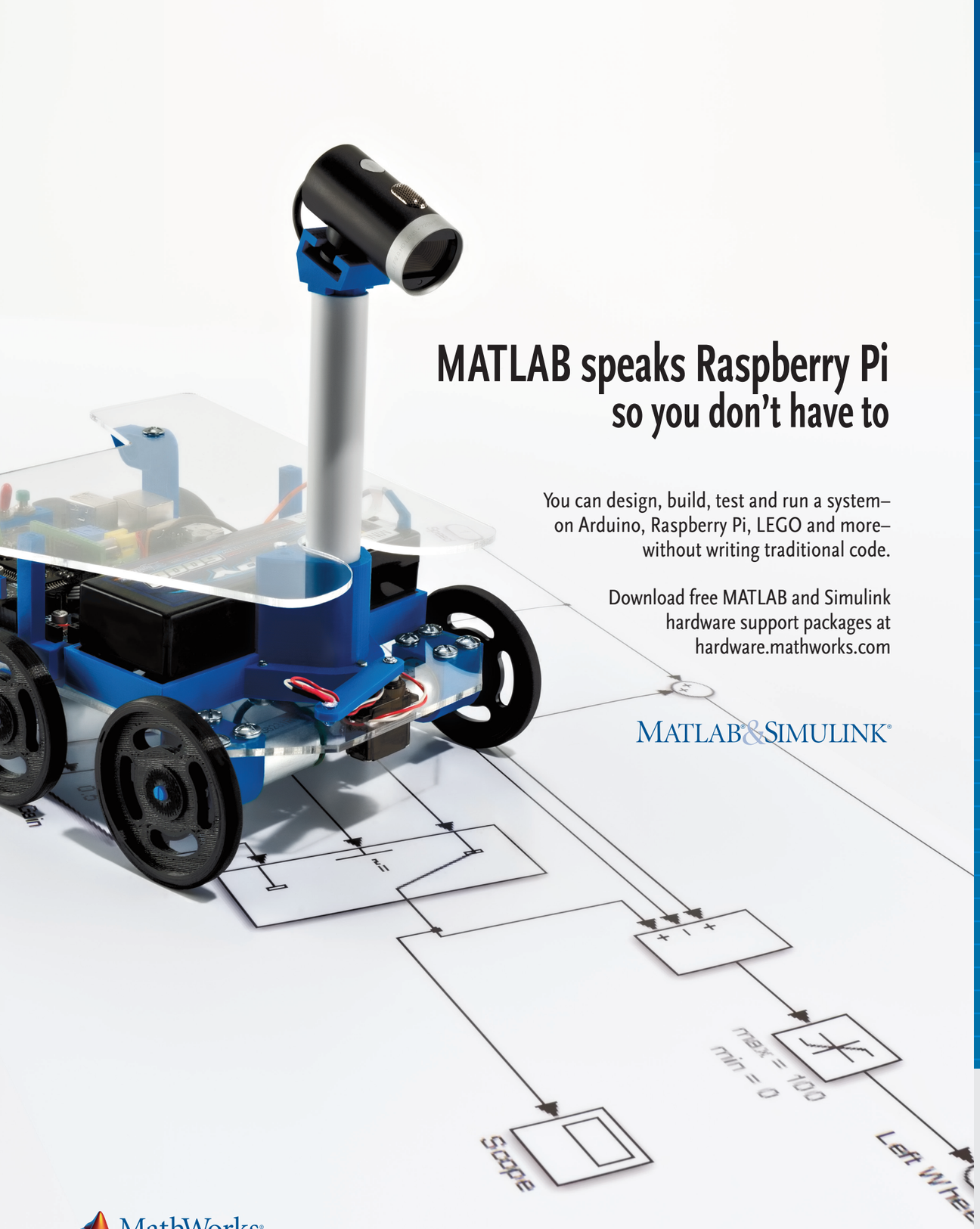
MathWorks News&Notes

The Magazine for the MATLAB® and Simulink® Community

Designing a Nonlinear Feedback Controller for the DARPA Robotics Challenge

ALSO IN THIS ISSUE

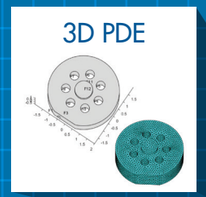
MathWorks®

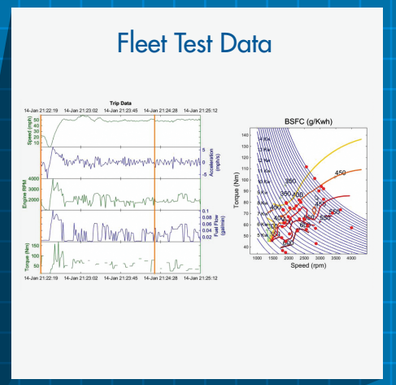


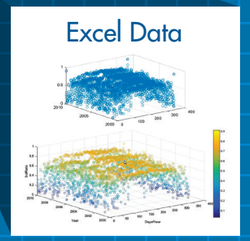
Explore MATLAB and Simulink Webinars

Join a live session or browse the library of recordings in 19 languages.

mathworks.com/recordedwebinars







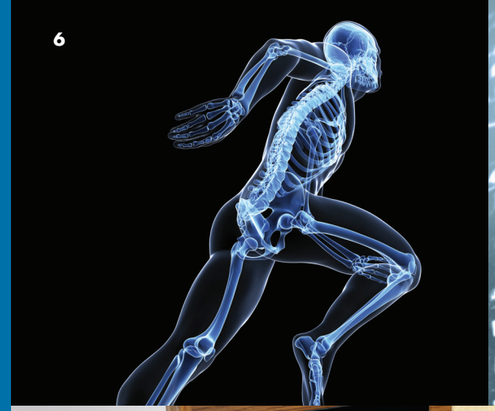
Tips and examples from technical experts on topics including:

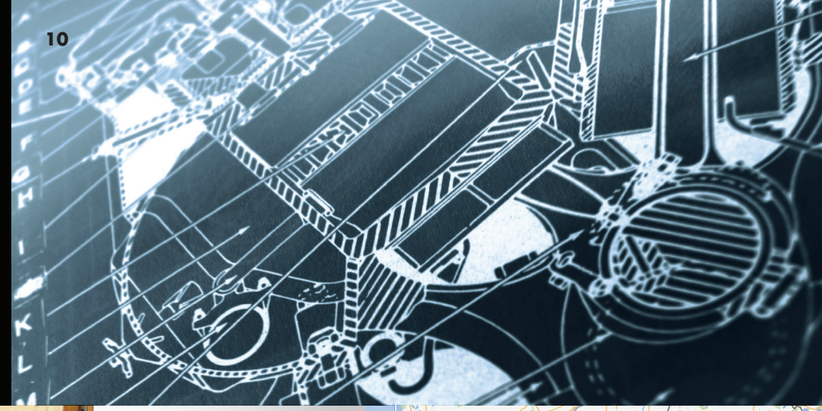
- Data analytics
- Real-time simulation and testing
- Physical modeling
- Signal processing and communications
- Verification, validation, and test
- Robotics
- Image processing and computer vision
- MATLAB and Simulink in academia
- Machine learning
- GPU and parallel computing
- Control system design and analysis
- Code generation and verification

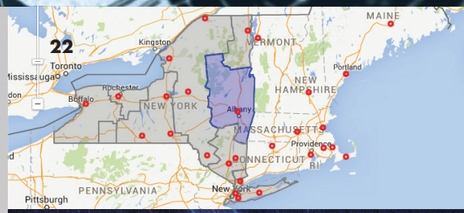
MathWorks News&Notes

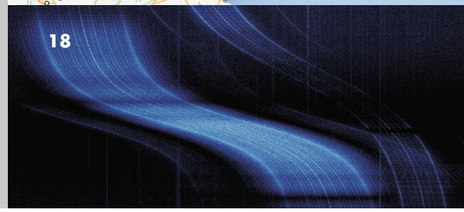
>> FEATURES

>> DEPARTMENTS









>> ABOUT THE COVER

The cover shows MIT's ATLAS robot in action. Built by Boston Dynamics and specially designed to negotiate rough terrain, ATLAS can walk, use tools, climb using hands and feet, and pick its way through congested spaces. The robot includes 28 hydraulically actuated degrees of freedom. Its articulated sensor head is equipped with stereo cameras and a laser range finder. Dr. Russ Tedrake and his colleagues at MIT designed control software for ATLAS in MATLAB and Simulink.

Managing Editor
Linda Webb

Editor
Rosemary Oxenford

Art Director
Robert Davison

Graphic Designer
Chris Roth

Technical Writer
Jack Wilber

Production Editor
Julie Cornell

Printer
DS Graphics

Production Staff
J. Gareau, K. Larkin, L. MacDonald, M. Murray, A. Pollack

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

Contributors and Reviewers
S. Abe, A. Ananthan, T. Atkins, G. Bourdon, C. Buhr, G. Campa, E. Charry, B. Chou, S. DeLand, E. Dillaber, G. Dudgeon, R. Dudgeon, A. Filion, J. Ghidella, D. Jaffry, W. Jin, K. Karnofsky, S. Kodial, T. Kush, C. Lagunowich, T. Lennon, D. Lim, J. Little, K. Lorenc, P. Maloney, T. Mathieu, JM Modisette, K. Motter, D. Ning, T. Otani, P. Pilotte, S. Popinchalk, O. Pujado, M. Ricci, D. Rich, B. Root, G. Rouleau, R. Rovner, G. Sandmann, J. Schlosser, B. Tannenbaum, R. Tedrake, G. Thomas, A. Turevskiy, N. Vasi, S. Velilla, Vijayalayan R, U. Wahner, S. Zaranek

Subscribe
mathworks.com/subscribe

Comments
mathworks.com/contact

Find Us Online

MathWorks

©2015 The MathWorks, Inc.
MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

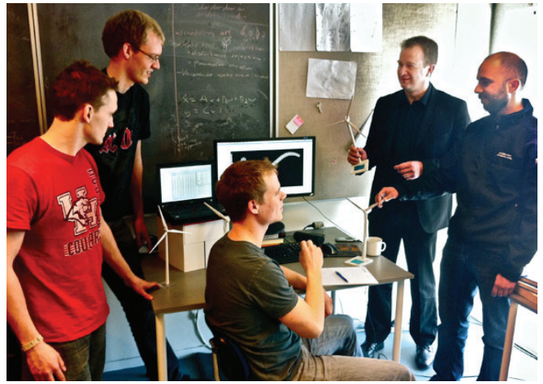
MIX
From responsible sources
FSC® C084277

Printed on 30% post-consumer waste materials

Made in the U.S.

The MATLAB Enabled Campus

Universities worldwide provide faculty, researchers, and students with anytime, anywhere access to MATLAB® and Simulink®. With MATLAB available to everyone on campus, these schools are fostering interdisciplinary study and research, supporting problem-based learning, and giving students hands-on, industry-relevant experience.



AALBORG UNIVERSITY

Using problem-based learning to turn students into active learners

Founded more than 20 years ago, the Problem-Based Learning (PBL) program at Aalborg University (AAU) in Denmark has become a model for universities around the world. From their first semester, AAU students use Model-Based Design to solve real-world problems in the energy, aerospace, and marine industries. Students have completed European Space Agency projects in which they used MATLAB and Simulink to model and simulate monitoring systems that detect faults during a spacecraft's re-entry. They are currently using MATLAB and Simulink to model, simulate, and prototype control systems and condition-monitoring algorithms for the European AEOLUS wind farm project.

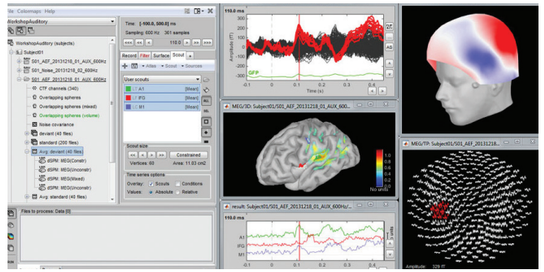
mathworks.com/aalborg

POLITECNICO DI TORINO

Inviting industry experts to teach students complex embedded system development

Companies in the region around Politecnico di Torino need automotive and avionics engineers capable of developing complex, high-integrity embedded software. To help meet this need, the university introduced *Model-Based Software Design*, a course for fifth-year students that combines lectures and practical exercises with seminars conducted by local engineers. Students learn embedded system design by building and simulating an executable model; rigorously validating, testing, and debugging it; and generating code for an embedded target.

mathworks.com/torino

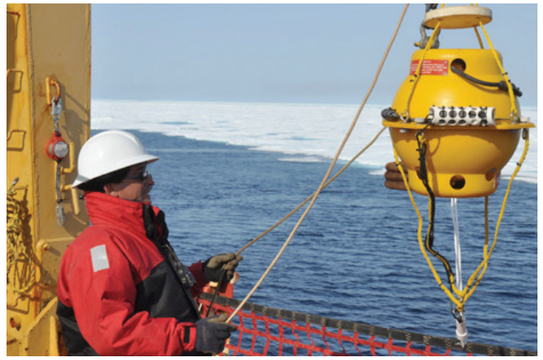


MCGILL UNIVERSITY

Enabling neuroscience researchers to visualize and process vast amounts of EEG/MEG data

Electroencephalography (EEG) and magnetoencephalography (MEG) instruments capture electrical activity from the entire volume of the brain at a rate of 1000 times per second, yielding roughly 100 MB of data per minute. Researchers at the McConnell Brain Imaging Center at McGill University, with colleagues at Case Western Reserve University and the University of Southern California, developed Brainstorm, an open-source, MATLAB based software application that neuroscience researchers with no programming experience can use to visualize and process large volumes of EEG/MEG data. Researchers can interact directly with their data, contribute new plug-ins, and exchange ideas and code prototypes with other Brainstorm users.

mathworks.com/mcgill



CORNELL UNIVERSITY

Accelerating bioacoustics data analysis with high-performance computing

Bioacoustics scientists study animal populations by analyzing animal sounds recorded in oceans, jungles, forests, and other natural environments. Bioacoustics Research Program scientists at the Cornell Laboratory of Ornithology have developed a MATLAB based platform for analyzing terabytes of data generated by passive acoustic monitoring systems. The team saved years of development time, cut analysis time from weeks to hours, and developed algorithms that enable researchers to assess the effect of man-made noise on natural environments and monitor endangered animal populations.

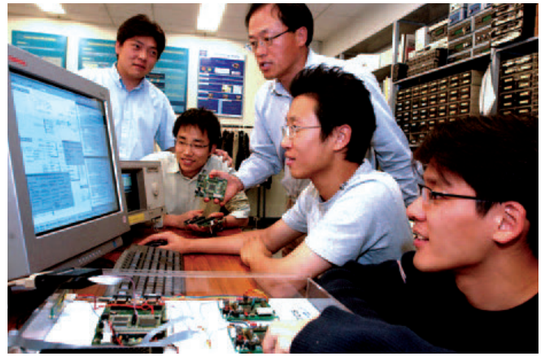
mathworks.com/cornell

UNIVERSITY OF MELBOURNE

Adopting a multidisciplinary curriculum to improve student outcomes

To prepare engineering students for careers spanning a wide range of disciplines, the Melbourne School of Engineering at the University of Melbourne adopted an educational model in which students complete a broad-based curriculum to earn an undergraduate degree in three years and a specialized Master of Engineering degree two years later. Students use MATLAB and Simulink to learn and apply concepts in basic linear algebra, control systems, signal processing, and mechatronics. Professors have coupled MATLAB with hands-on activities, enabling students to explore new concepts with immediate, visual feedback as they progress from theory to numerical computation and lab experiments.

mathworks.com/melbourne



HANYANG UNIVERSITY

Shortening ECU development times while delivering practical skills to future engineers

Like their competitors worldwide, Korean automotive manufacturers are facing increased development costs due to growing technological complexity, stricter fuel economy and safety regulations, and more diverse customer needs. The Automotive Control and Electronics Laboratory (ACE Lab), a research institute affiliated with Hanyang University in South Korea, shortened development times and lowered costs by replacing their traditional control development process with Model-Based Design. As part of its collaboration with ACE Lab, Hanyang University incorporated MATLAB and Simulink into class assignments and exercises at the graduate and undergraduate levels to help students acquire the skills and practical experience needed by Korea's future automotive engineers.

mathworks.com/hanyang

LEARN MORE

MATLAB Campus License
mathworks.com/campus-license

Developing Motion Analysis Algorithms for Medical, Sports, and Occupational Safety Applications

By Edgar Charry, dorsaVi

>> ELITE ATHLETES WANT TO RETURN TO MATCH PLAY AS QUICKLY

as possible after injury. As a result, they often convince themselves and their physicians that they are fit to play before they have fully recovered. Advanced technologies such as optical tracking systems, which capture motion, and force plates, which measure ground reaction forces, enable athletes and trainers to determine when the athlete can safely return to full activity.

These technologies provide metrics and insights that are impossible to obtain by simply observing the athlete, but they have several drawbacks. In addition to being costly, they typically depend on specially trained technicians and lengthy setup procedures. Further, because they often require the athlete to perform in a constrained environment, such as on a treadmill, they make it difficult to assess natural movement.

At dorsaVi, my colleagues and I have developed wearable, wireless motion analysis devices that precisely measure and track movement while the athlete moves freely in any environment (Figure 1).

ViPerform incorporates inertial measurement units (IMUs) and magnetometers, as well as electromyography sensors for measuring muscle activity. Sensor data is transmitted to a recording and feedback device (RFD), which can be worn on the arm or carried in a pocket. The RFD sends the data to a PC, where it is processed and displayed by the dorsaVi software package.

At the heart of the dorsaVi technology are proprietary algorithms that filter and analyze raw sensor data, providing information that can be used to evaluate knee control, lower back range of motion, hamstring activity, hip and core control, and running performance. By developing and testing the algorithms in MATLAB® and developing portable C code with MATLAB Coder™, we cut development time by almost half compared with our previous approach, which involved hand-coding in C#.

Developing Algorithms for Objective Analysis

Objective analysis is made possible by sophisticated algorithms that process raw sensor data and present results in a meaningful form. For example, an algorithm may identify large negative spikes in the accelerometer output that occur on each step when an athlete is running. From the timing and magnitude of these spikes, the algorithm can calculate the runner's cadence and the ground reaction force of each step, respectively. This analysis may detect an asymmetrical cadence or unequal ground reaction forces for each leg, revealing an inability or unwillingness for the athlete to apply equal force through both legs. Similarly, analyzing knee deviation, flexion, and rotation as the athlete squats or hops can guide recovery from ACL and other knee injuries. Even healthy athletes can benefit from such analysis, using feedback from the algorithms to refine their technique or optimize the efficiency of key movements.

When we implemented our algorithms directly in C#, we had to develop our own low-level signal processing and graphing functions. By switching to MATLAB, we have sped up development by using built-in functions for data visualization and discrete Fourier transforms. Today, we use Signal Processing Toolbox™ and Wavelet Toolbox™ to further streamline development. For example, we use Signal Processing Toolbox to design and apply Butterworth and other infinite impulse response (IIR) filters to sensor data. We identify peaks and troughs in sensor signals by computing the continuous wavelet transform (CWT) coefficients of the signals using Wavelet Toolbox. Identifying these patterns in signals is essential to pinpointing gait events, such as heel strike and toe off, for our running analysis.

FIGURE 1. *ViMove sensors in a running test.*

Verifying and Testing the Algorithms

After developing an algorithm in MATLAB, we verify it by comparing its results with results produced by an independent measuring device, such as an optical motion tracking system or a force plate. For example, we compare the timing of gait events detected by our algorithm to the timing of the ground force reaction spikes generated by the force plates. In this phase, we inspect plots and graphs created in MATLAB to identify and then correct potential deficiencies in the algorithm (Figure 2).

Once we are satisfied that the algorithm is performing well on a limited set of test scenarios, we run more than 1000 tests in MATLAB using recorded data from athletes around the world. We continue to fine-tune and optimize the algorithm until it meets our requirements for accuracy and performance.

Generating C++ Code from Our MATLAB Algorithms

DorsaVi software, which serves as the primary interface for viewing sensor data and evaluating an athlete's performance, is written primarily in C#. When we first began using MATLAB to develop signal processing algorithms, our workflow still relied on C++ and C# programmers to implement the production version of the algorithm that was incorporated into the dorsaVi software. This approach was inefficient because it involved duplication of effort. It could take an additional month to reprogram and retest algorithms that had already been developed and tested in MATLAB.

To eliminate this inefficiency and shorten project delivery times, we decided to use

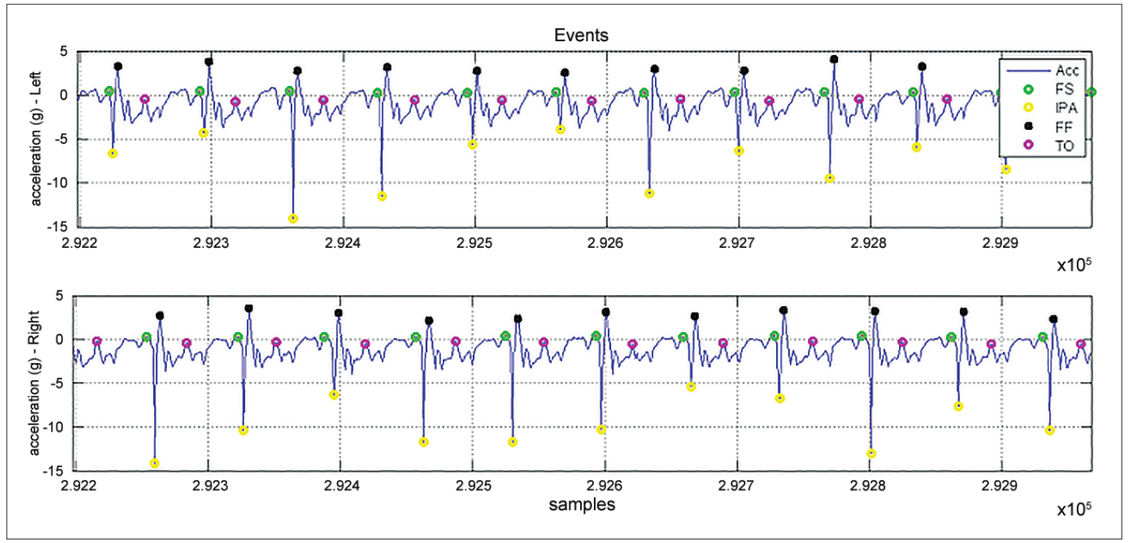


FIGURE 2. *MATLAB plot showing results of a running test.*

MATLAB Coder to generate C++ code from our verified MATLAB algorithms. We prepare our algorithms for code generation by initializing all variables and looking for opportunities to optimize loops. We verify that our algorithm is ready for code generation by generating a MEX function that wraps the compiled code, and then invoking the MEX function in place of the original MATLAB algorithm. After generating the C++ code from our algorithm, we compile the algorithm into a DLL. The C# programmers load this DLL into the dorsaVi software.

Translating the MATLAB algorithm into production code previously took up to a month. With MATLAB Coder it now takes a day or two. As a final step, we perform full system testing of the algorithms of the dorsaVi software. To date, we have not found any defects introduced during code generation. The extensive testing we perform in MATLAB enables our C# programmers to spend their time developing new features for the dorsaVi software instead of recoding our algorithms in C# and then retesting them.

Reusing Algorithms and Responding to Customer Requests

We have reused several algorithms initially developed for ViPerform in our ViMove and ViSafe products, designed for clinical and occupational health applications, respectively. My team is currently working on new algorithms to analyze data from sensors placed on different body parts, as well as algorithm development and enhancement requests from existing customers.

With MATLAB and MATLAB Coder, we can respond quickly to customer requests. In a recent release, we updated two algorithm modules, created two new ones, and delivered all four with a confidence that would have been impossible using our old workflow. ▪

LEARN MORE

iSonea Develops Mobile App and Server Software for Wheeze Detection and Asthma Management
mathworks.com/isonea

Analyzing Fitness Data from Wearable Devices in MATLAB
mathworks.com/fitness-data

Optimizing a Diesel Engine Aftertreatment System with MATLAB and GT-SUITE

By Seth DeLand, MathWorks, and Ryan Dudgeon, Gamma Technologies

>> OVER THE PAST DECADE, DIESEL ENGINE AFTERTREATMENT SYSTEMS have been developed rapidly to comply with strict diesel emission standards—which have also changed rapidly during this period. Today's diesel aftertreatment systems are highly complex components, and designing and optimizing aftertreatment systems is an essential part of the engine development process.

A well-calibrated aftertreatment system keeps emissions at regulation levels while enabling the engine to target its most efficient operating points for fuel economy. To meet this goal, engineers need to perform simultaneous optimization of multiple subsystems, including the engine, the controls, the vehicle, and the aftertreatment system.

This article describes a workflow for simulating and optimizing the ammonia oxidation catalyst (AOC) component of a diesel engine aftertreatment system. In this workflow MATLAB® and Simulink® are used to optimize a model built and simulated in GT-SUITE.

Performing Parameter Estimation with MATLAB

We begin by setting up an aftertreatment model in GT-SUITE. System components are linked together in sequence. For a diesel engine these components can include a diesel oxidation catalyst (DOC), a diesel particulate filter (DPF), a selective catalytic reduction (SCR) subsystem, and an ammonia oxidation catalyst (AOC) (Figure 1).

Gases coming from the engine cylinders pass through the DOC and DPF, where CO, unburned hydrocarbons, and soot are eliminated. They then flow to the SCR subsystem, which removes NO_x by using upstream injection of aqueous urea solution, or diesel exhaust fluid (DEF). This fluid breaks down to produce ammonia, which reduces or removes NO_x in the SCR. A final catalyst, the AOC, oxidizes excess ammonia before it leaves the tailpipe.

Having good parameter values for exhaust aftertreatment models is necessary to ensure that we get meaningful results from the simulation match the experimental (measured) data (Figure 2). The optimization problem is to minimize the difference between the simulated data and the measured data by finding optimal values for the 14

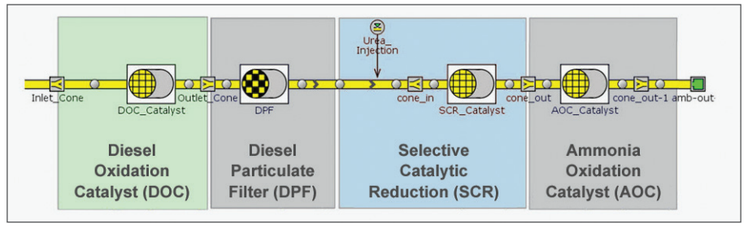


FIGURE 1. *A typical diesel exhaust system, modeled in GT-SUITE.*

model. However, few of these parameters can be directly measured. The steps outlined below describe how to use parameter estimation to calibrate the AOC component in the GT-SUITE model using experimental data. These same steps can be applied to each component in the aftertreatment system.

The behavior of the AOC can be modeled in GT-SUITE using six global reactions, with the temperature dependence of each reaction rate represented as an Arrhenius equation with two rate constants.

There are 14 independent variables (parameters) to be estimated: the two rate constants for each of the six reactions, and two global inhibition constants for the AOC.

We want to find values for these 14 parameters so that the results of the GT-SUITE reaction parameters.

The controller will be implemented in Simulink. Simulink also provides a conduit for the optimizer to access GT-SUITE model parameters. We begin by adding a Simulink harness block to the GT-SUITE model (Figure 3). This block is used to communicate with a Simulink model. The Simulink model uses a corresponding GT-SUITE model block to pass values for the 14 reaction parameters to GT-SUITE.

We run hundreds of cosimulations with Simulink and GT-SUITE using different parameter values. To find a set of values that minimizes the differences between the simulated and the measured results, we write a simple MATLAB script that calls the Global Optimization Toolbox `MultiStart`

Tightening Regulations and Other Challenges

Regulated emissions include carbon monoxide (CO), unburned hydrocarbons, particulate matter or soot, and nitrous oxides (NO_x). An exhaust aftertreatment system reduces these emissions by means of the catalytic reactors and filters that make up the exhaust system between the engine and the tailpipe. Because some of these reactors require precious metals, such as platinum and palladium, they represent a significant cost for manufacturers.

In the on-road and off-road vehicle industries, regulation of harmful emissions continues to tighten. EPA Tier 4 regulations for nonroad diesel engines call for a further 90% reduction in exhaust emissions and require in-use diesel fuel to reduce sulfur levels by more than 99%.

Greenhouse gas emissions, most notably carbon dioxide (CO_2), are now coming under closer scrutiny. European regulatory bodies have been regulating CO_2 emissions for some time. Meanwhile, the EPA is tightening restrictions on heavy-duty CO_2 emissions, and developing nations such as China and India are rolling out their own targets. The solution to reducing CO_2 emissions is to increase vehicle fuel economy.

Reducing emissions and increasing fuel economy are difficult objectives to achieve simultaneously. To meet these competing requirements, the aftertreatment system and the engine must be designed at the same time since each system affects the performance of the other. For example, adding a filter or a catalyst to the exhaust system creates higher back-pressure for the engine, reducing engine efficiency and fuel economy. Similarly, engine design decisions regarding exhaust gas recirculation, fuel injection parameters, air-fuel ratio, and warm-up strategies affect aftertreatment system performance and, therefore, vehicle emissions.

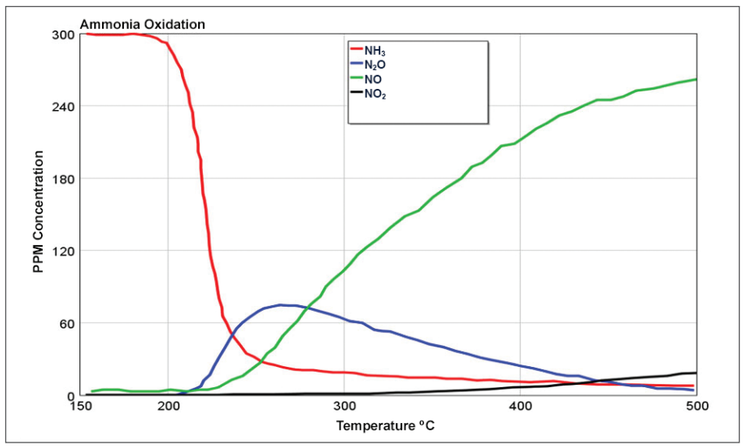


FIGURE 2. *Plot showing experimental data for concentrations of NH_3, N_2O, NO, and NO_2 leaving the AOC as a function of temperature.*

solver, which searches for the global minimum of a constrained nonlinear multivariable function.

Because each simulation takes several seconds to run, it can take about an hour for the optimization to find a good fit. This process can be accelerated by using Parallel Computing Toolbox™ to execute the simulations in parallel on multiple computing cores. Note that `MultiStart` uses a gradient-based optimization solver, and it works best when the objective function is smooth. If the objective function is nonsmooth, or has discontinuities, MATLAB has several other global optimization solvers that can be used.

The optimization gives us a set of parameter values that enable the model to produce results that closely match the results measured in the lab. By updating the GT-SUITE model of the AOC with these values and repeating this process for other aftertreatment system components, we ensure that our system-level simulations will produce accurate and meaningful results.

Performing System-Level Optimization

Now that we have a well-calibrated aftertreatment system model, we can perform a variety of system-level studies such as analyzing interactions between emissions, engine back-pressure, and fuel economy, or exploring controller strategies for diesel exhaust fluid injection. We can also identify optimal sizes for aftertreatment system components, including the AOC. Because the AOC uses platinum as its catalytic material, if it is too big, it will be too expensive. If it is too small, however, it will not meet emissions requirements.

To analyze the tradeoff between NO_x emissions and AOC cost, we need to solve a multiobjective optimization problem in which the variables are the SCR length, the AOC length, and the AOC catalytic material loading.

To set up this optimization problem, we build a vehicle model that includes engine, aftertreatment system, driver control (which takes the vehicle through the New European Driving Cycle (NEDC)), and vehicle. As with the parameter estimation problem, we build a Simulink model with an interface to the GT-SUITE model (Figure 4). A controller in the Simulink model manages the urea dosage in the GT-SUITE model based on the measured NO_x, temperature, and ammonia slip (the ammonia passing through the SCR). This setup supports closed-loop cosimulation using the GT-SUITE model as the plant.

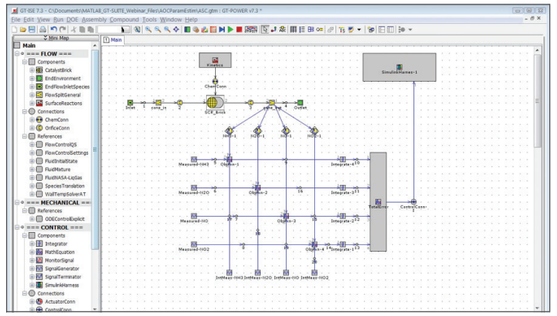


FIGURE 3. *GT-SUITE model of the AOC with a Simulink harness block (upper right).*

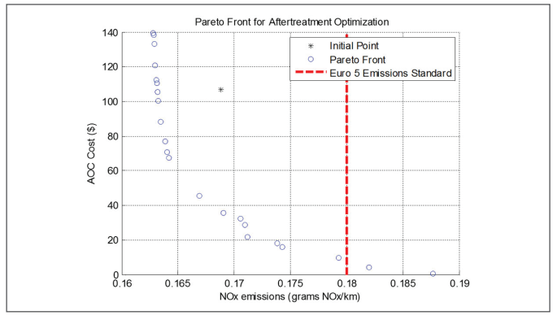


FIGURE 5. *Plot of the Pareto front for aftertreatment optimization produced by the MATLAB multiobjective genetic algorithm solver.*

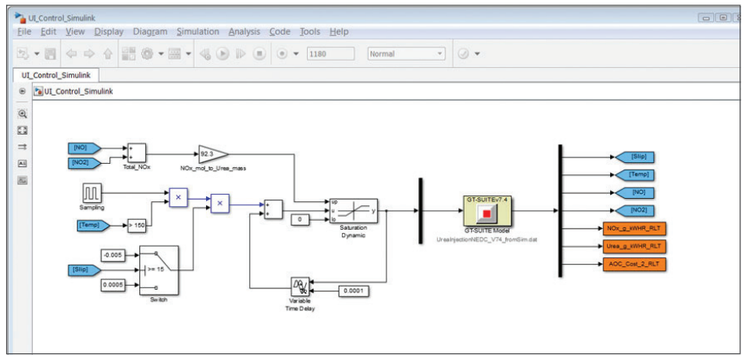


FIGURE 4. *Simulink model with an interface to the GT-SUITE vehicle model.*

We write a MATLAB script that uses a multiobjective genetic algorithm solver to tune parameters in the GT-SUITE model. The solver runs thousands of simulations with different combinations of SCR length, AOC length, and AOC loading, and evaluates their cost and NO_x emissions on the NEDC.

In a multiobjective optimization problem like this one, no single solution simultaneously optimizes each objective. There are many optimal solutions along the Pareto front (Figure 5).

The blue circles in Figure 5 represent optimal solutions along the Pareto front, making it easy to see the tradeoff between NO_x emissions and AOC cost. These solutions minimize cost for a given NO_x emissions limit—or, from another perspective, minimize emissions for a given set cost. Since our goal is to keep emissions below a specific limit, we select the solution along the Pareto front that is below that limit and has the lowest cost. For example, if the emissions target is 0.18 g/km, then the minimal AOC cost will be under $20.

Simulating an entire drive cycle thousands of times can be a lengthy process—to run all the necessary simulations for one optimization took approximately five days on a single processor. Fortunately, genetic algorithms are well-suited to acceleration using parallel computing techniques. We used Parallel Computing Toolbox and MATLAB Distributed Computing Server™ to run the simulations on a 64-core computing cluster. Because we had 64 cores available, we selected a population size of 64 for the genetic algorithm, making it possible for the algorithm to simultaneously evaluate 64 points in each iteration or generation. On the computing cluster, our optimization took about four hours.

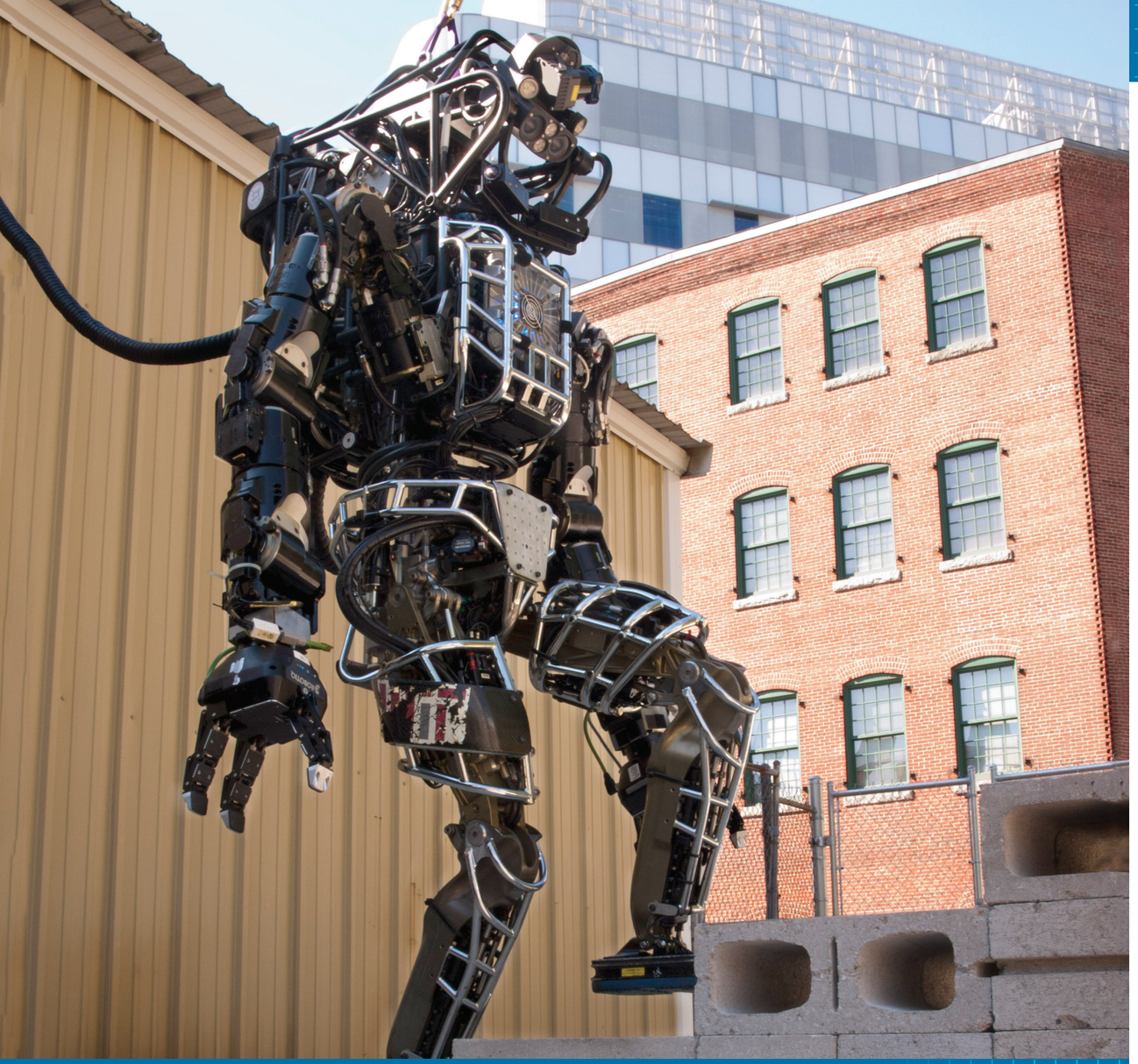
Extending this Approach

The approach described in this article can be applied to the tuning of additional parameters, or it can be used to support additional objectives such as minimizing vehicle weight. For example, it can be used to trade off control strategies and optimize controller parameters to minimize diesel exhaust fluid consumption—and as a result, reduce the ownership cost of the vehicle. By using the vast numbers of computational resources available today, engineers can quickly investigate design changes and optimize several parameters at a system level. ▪

LEARN MORE

Diesel Engine Aftertreatment System Development Using MATLAB and GT-SUITE
37:02
mathworks.com/video-93110

Optimizing Performance and Fuel Economy of a Dual-Clutch Transmission Powertrain
mathworks.com/dual-clutch



Designing a Nonlinear Feedback Controller for the DARPA Robotics Challenge

By Russ Tedrake, Massachusetts Institute of Technology

>> IN DECEMBER 2013, A HUMANOID ROBOT BROKE THROUGH A wall, cleared debris from a doorway, unspooled and connected a firehouse, and drove a utility vehicle through an obstacle course at the Homestead-Miami Speedway. The robot's control software was developed in MATLAB® and Simulink® by a team from Massachusetts Institute of Technology participating in the DARPA Robotics Challenge (DRC). This competition is designed to spur research into developing robots that can work in hazardous areas with task-level autonomy. A robot with task-level autonomy can be instructed to perform simple tasks, such as turning a steering wheel or grasping a handle, which it then carries out on its own.

From the time we received our robot to the day of the competition, we had less than five months to develop, debug, and test our controller algorithms. MATLAB and Simulink helped us keep to this aggressive schedule. We were able to prototype highly sophisticated, optimization-based controllers at a pace that would have been impossible with C or another low-level language.

Making Robots Move with Grace and Efficiency

Our work at the DRC was a continuation of my research into making robots move gracefully in the real world. My goal is to design legged robots that move as skillfully as ballerinas and unmanned aerial vehicles that fly like birds. These are fundamental problems for the field of robotics; they also force us to solve hard nonlinear control problems that will have applications in many other domains.

Watch carefully the next time you see a bird fly past the window and land on a branch. That little bird is casually but dramatically outperforming some of the best control systems ever designed by humans. During a "perching" maneuver, birds rotate their wings and bodies so that they are almost perpendicular to the direction of travel and to oncoming airflow. This maneuver increases the aerodynamic drag on the bird, both by increasing the surface area exposed to the flow and by creating a low-pressure pocket of air behind the wing.

Viscous and pressure forces combine for the desired rapid deceleration, but the maneuver has important consequences: The wings become "stalled," meaning that they experience a dramatic loss of lift and, potentially, of control authority. The aerodynamics become unsteady (time-varying) and nonlinear, making the aerodynamic forces difficult to model and predict accurately. Yet birds perch with apparent ease. By comparison, helicopters and vertical take-off and landing (VTOL) airplanes require considerable time and energy to land on a target. Similarly, few jet pilots would be willing to fly between skyscrapers, yet owls and hawks navigate dense forests with ease.

Control systems that mimic such feats must make sophisticated logical decisions about how and where the robot will move. Although the equations describing the system kinematics and dynamics are nonlinear, these equations have exploitable structure. For example, many can be described in terms of rational polynomial equations. This means that we can write efficient algorithms that use numerical algebraic geometry and convex optimization to assess system stability over a wide range of operating conditions.

When we set out to land a small glider on a wire, we used MATLAB and Simulink to implement a relatively standard approach to trajectory optimization and time-varying linear feedback. Using this approach we produced controllers that could consistently land the plane as long as it was always launched from the same location and at the same velocity. By quickly evaluating the stability of this system using the polynomial approach, represented as a stability "funnel" (Figure 1), we were able to design a feedback library that would reliably land the aircraft on the perch from a wide range of initial conditions. Now we can simply throw the airplane towards the wire from any position, and it will always find its way to the perch.

Why MATLAB?

As we began work for the DARPA Robot Challenge, some of my colleagues questioned the use of an interpreted language such as MATLAB in a real-time feedback control loop. They were concerned that the MATLAB algorithms would not execute quickly enough, and

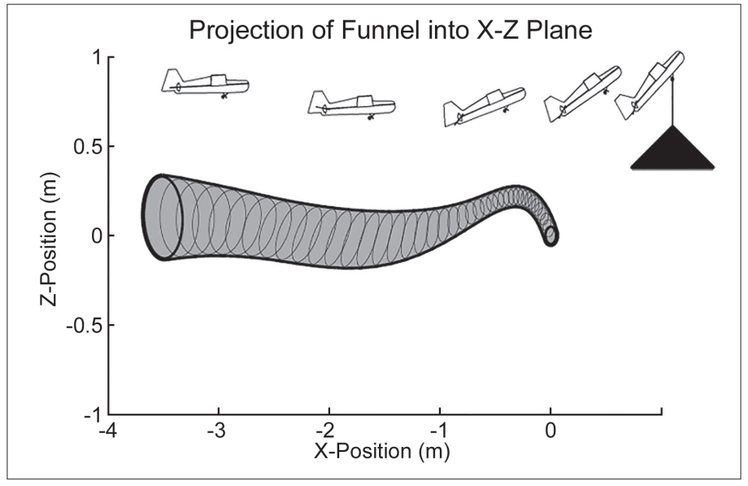


FIGURE 1. *Funnel-shaped area from which the glider can reliably land on its perch.*

that interruptions by just-in-time-compiling, garbage collection, or background processes would cause jitters and affect timing.

After carefully considering these issues, I determined that our MATLAB algorithms would be able to run on a PC at 300 Hz or faster, and with enough timing accuracy to meet the needs of our control design. I knew that if we needed to speed up critical or slow components, we could use C++ code within our MATLAB simulations. While higher sampling rates and more reliable timing make control design easier, I believe that controllers can and should be designed with sufficient robustness to handle lower rates and less accurate timing. Our human nervous system manages complex movements with relatively low bandwidth and high latencies, and I strive to develop controllers that can do the same.

The numerous advantages of using MATLAB and Simulink became apparent as development progressed. For example:

- MATLAB is one of the best tools available for rapid prototyping algorithms based on linear algebra.
- Most of the commercial optimization solvers we use have a MATLAB interface that makes it easy to invoke them from MATLAB code.
- MATLAB and Simulink provide numerous ways to visualize data, simulation results, and the motion of virtual robots.
- Simulink enables us to develop sophisticated models, incorporate MATLAB classes as S-functions, apply ODE solvers, and simulate hybrid systems and systems that combine continuous and discrete components.
- Many of the students on our DRC team had used MATLAB in their undergraduate or graduate studies in controls, communications, and signal processing.

The First DRC Event: Guiding a Simulated Robot in a Virtual Environment

Teams that did not have their own robots for the DRC were invited to participate in the Virtual Robotics Challenge, which tested software teams' ability to control a simulated robot as it completed three tasks in a virtual environment. Seven finalists from this round would move forward to the DRC trials using an ATLAS robot provided by DARPA. ATLAS is a humanoid, hydraulically powered robot created by Boston Dynamics.

The three tasks, closely related to the eight tasks that the real robot would complete later in the competition, were quite complex. For example, they required the robot to walk over rough, variable terrain and manipulate a fire hose (Figure 2).

For the DRC, we needed to make the whole-body motion planning and control algorithms for ATLAS execute quickly enough to run in real time. Operating in an unfamiliar environment, the robot would be commanded to execute a task, and the controller would need to immediately plan the motion of the entire robot. We achieved that in MATLAB by exploiting what we knew about the robot's structure and the equations used to describe it—just as we had done in earlier research projects. For mechanical systems like ATLAS, we know that energy is conserved, that the mass matrix is positive, and that the center of mass dynamics is uniquely determined by the influence of gravity and contact forces between the robot and the environment. The equations have important sparsity patterns: The dynamics of the right hand are only coupled to the dynamics of the left foot through the mass matrix.

We had just eight months from the Virtual Challenge kickoff meeting to the actual competition. With that aggressive timeline, we had to develop quickly. In MATLAB we were able to rapidly develop sophisticated control ideas, prototype them, and debug

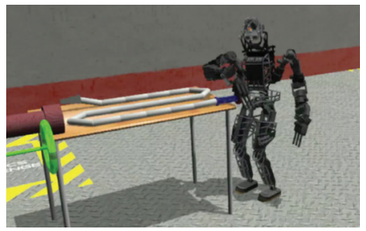


FIGURE 2. *The virtual ATLAS robot manipulating a fire hose in the simulation environment.*

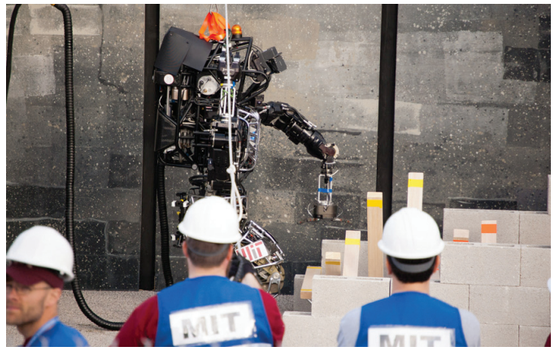


FIGURE 3. *The ATLAS robot removing debris from a doorway at the DRC trials.*

FIGURE 4. *MIT team members assessing how to fit ATLAS into the DRC utility vehicle.*

them visually, and that was much more important than having code that ran 2% or even 20% faster.

Two months after being named one of the seven winners of the Virtual Robotics Challenge, we received our ATLAS robot.

Taking ATLAS from a Virtual Environment to the Real World

Once again, a tight timeline made rapid development imperative. In this second phase of the competition we had just five months to program ATLAS to perform eight tasks, including walking on uneven terrain, climbing a ladder, clearing debris from a doorway, breaking through a wall, turning a valve, connecting a fire hose, and driving a utility vehicle. Humans were allowed to direct the robot, but only via a low bandwidth communications channel, making some degree of task-level autonomy essential.

The ability to quickly implement and debug algorithms with MATLAB and Simulink proved instrumental to our ability to produce a controller capable of guiding ATLAS through such complex tasks.

To plan ATLAS' movements and perform other required tasks during the competition, we had five or six separate MATLAB and Simulink processes running simultaneously on desktop PCs. These processes communicated with the ATLAS robot via UDP using Lightweight Communications and Marshalling (LCM), a set of libraries designed for real-time systems that requires data marshalling and message passing.

Two tasks in particular underscored the value of the task-level autonomy that we had achieved with MATLAB and Simulink. When our ATLAS robot was clearing debris from the doorway, a board it had just moved fell across its feet. ATLAS excels at many tasks, but the kinematics of the robot makes touching its own toes very challenging. Still, our team was able to overcome the unanticipated setback, going "off script" to successfully direct ATLAS to remove the board from its feet before clearing the remaining debris (Figure 3).

There was another challenge. We knew that ATLAS barely fit in the car it had to drive (Figure 4), but we had no practical means of attempting the driving task near the MIT campus. We had just 30 minutes to experiment with the car and ATLAS before the 30-minute task began. No team before us had managed to move the car off the starting line. Due to the robustness of the solution we had developed using MATLAB and Simulink, we ended up spending 45 minutes getting ATLAS into the car but were then able to have it turn the wheel, depress the accelerator, and drive halfway down the course before our time expired.

Preparing for the DRC Finals

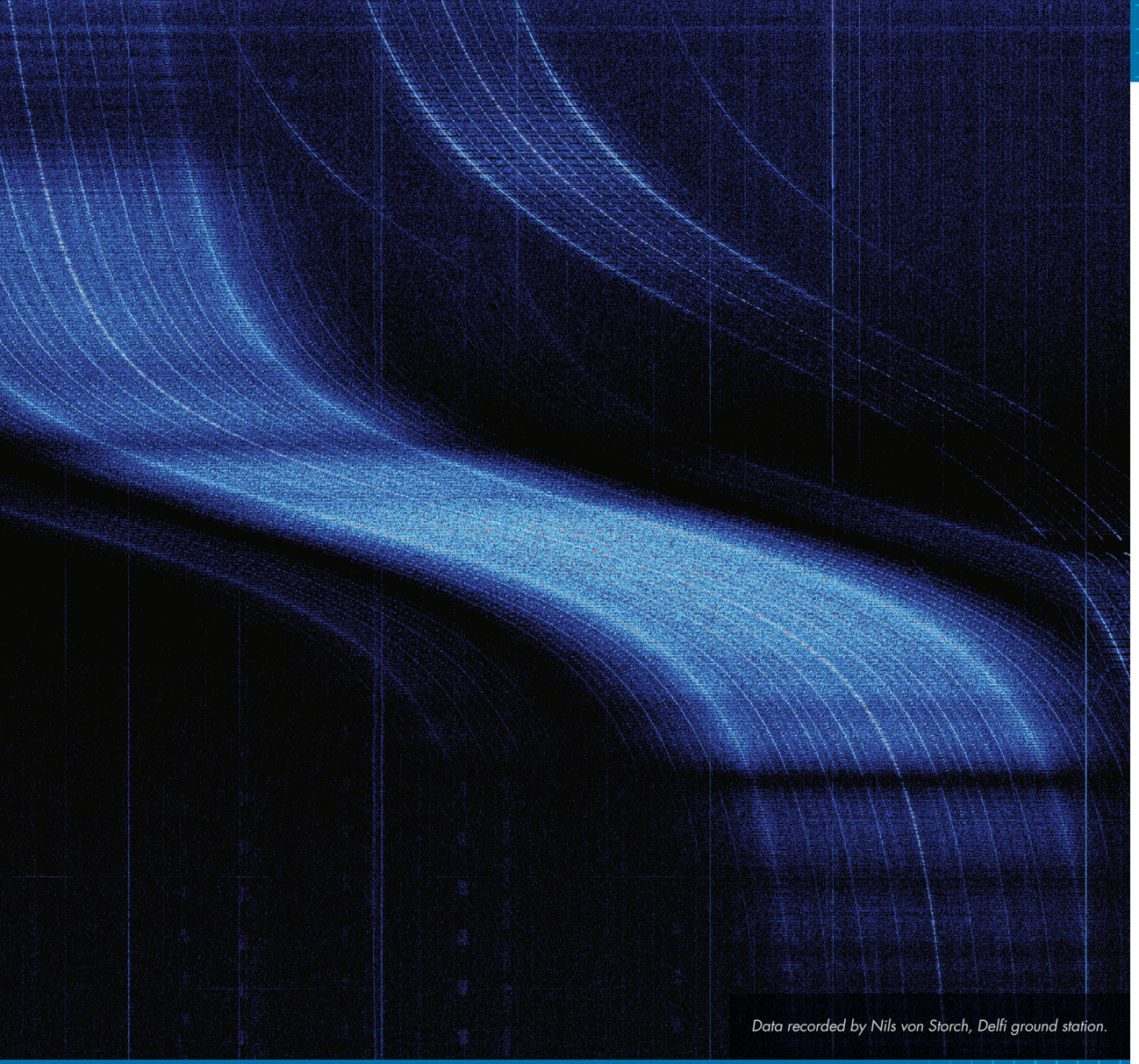
We placed fourth in the DRC trials, and qualified for the DRC finals held in June 2015. In this phase of the competition, the robots complete a series of physical tasks with degraded communications between the robots and the teams that operate them. This challenge places an even greater importance on task-level autonomy.

Having put an immense amount of effort into software engineering, our team has made a large portion of the software we've written for the DRC available as an open-source distribution called Drake. Drake is a general planning and control toolbox for nonlinear dynamical systems. It includes a rich dynamics engine for rigid body systems with frictional contact, trajectory motion planning, and nonlinear verification, as well as multiple hardware interfaces and methods for visualization, estimation, and parameter identification. ▪

LEARN MORE

Mobile Robot Simulation for Collision Avoidance with Simulink *45:02*
mathworks.com/video-90193

MIT DARPA Robotics Challenge Team
drc.mit.edu



Data recorded by Nils von Storch, Delfi ground station.

Teaching Hands-On Satellite Tracking and Communication to Delft University of Technology Undergraduates

By Bart Root, TU Delft

>> ON APRIL 28, 2008, A SATELLITE DESIGNED AND CONSTRUCTED by Delft University of Technology (TU Delft) students and faculty was launched into orbit. Its mission was to provide students with hands-on training on a real spacecraft project. The Delfi-C3 satellite (Figure 1) was equipped with thin film solar cells, autonomous sun sensors, and communications systems, but it lacked an off switch. This omission turned out to be fortuitous—seven years after launch, the satellite is still in operation, and we use it to teach undergraduate and master's aerospace engineering students satellite tracking and communications.

When I was asked to participate in teaching *Satellite Tracking and Communications* in our new Space Minor program, I wanted to give students a chance to work directly with real ground station hardware for orbit determination. My colleagues and I built a working ground station on campus dedicated to this purpose. In the course, students use MATLAB® to estimate when the satellite will pass over the ground station, plot the satellite's ground track, and analyze signal data captured from the satellite.

MATLAB is ideal for this work because it enables students to handle the vast amount of data involved. MATLAB supports a project-oriented curriculum much more effectively than low-level languages such as C++ and Java® because it enables students to get started on projects quickly, and readily understand the work of their team members.

The best part of teaching is the moment when a student's expression changes from "What are you talking about?" to "Now I understand this!" MATLAB makes it easier for students to get to that moment because they can interactively experiment, explore, and visualize new concepts.

Building a Working Ground Station

I worked with another researcher and an undergraduate on the initial design for the ground station and on a funding proposal to purchase equipment, which included UHF, VHF, and GPS antennas, as well as a software-defined radio (SDR), GPS clock, radio, and computer.

We installed the antennas on the roof of the tallest building on campus (Figure 2). Because the land around TU Delft is very flat, these antennas, positioned at an altitude of just 100 meters, can pick up signals from satellites at almost minus two degrees elevation—below the horizon—enabling us to track the satellite for long periods of time.

Simulating Basic Satellite Communications

To help students in *Satellite Tracking and Communications* understand the signal processing theory they learn in lecture, we start with a simple hardware project in which they use MATLAB, two Arduino® microcontrollers, and basic electronic components to construct a simulated satellite communications link.

In this setup, one Arduino acts as the ground station and the other as the satellite. Students learn how to encode and decode information. They also get firsthand experience in information bit-loss and signal gain changes by observing what happens to the communications link when the Arduinos move toward and away from one another.

Planning a Satellite Pass

Before they can capture signals from the Delfi-C3 satellite at the ground station, students must determine when the satellite will pass over their position. For one assignment, students use MATLAB to calculate the satellite's approximate position based on two-line element (TLE) data provided by the U.S. Air Force, which tracks all objects in Earth's orbit.

To help them with this assignment, we demonstrate in lecture how to propagate an orbit from a force model and a state vector. MATLAB makes it easy to show students

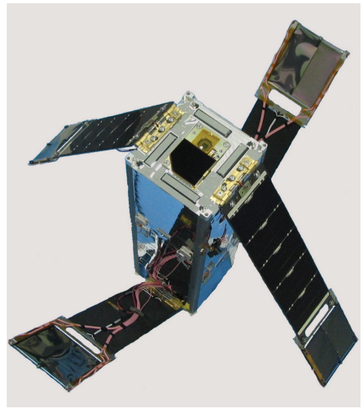


FIGURE 1. *The Delfi-C3 satellite, designed and constructed at TU Delft.*

Euler, Runge-Kutta, and Adams-Bashforth integration methods for orbit propagation. They put these concepts into practice using ordinary differential equation solvers and a variety of integrators in MATLAB. By the end of this exercise they can propagate virtually anything—from ballistic trajectories on Earth to satellite orbits in space.

Recording and Analyzing Satellite Data

About half an hour before the time calculated for the satellite to pass, the students come to the ground station to familiarize themselves with the antennas and equipment.

As the satellite approaches, the students' excitement grows, and they are thrilled when they hear the satellite's signal over the radio, its frequency decreasing during the whole pass due to the Doppler effect. We record the signals as the satellite passes, and students download the resulting binary data files to their computers for processing.

Working in MATLAB the students apply filters to remove noise introduced by the amateur radio enthusiasts who sometimes share the satellite's bandwidth or other sources of error. One of the biggest challenges with the data processing is finding the relatively few relevant data points for the signal in the gigabytes of data that were recorded. Students write MATLAB scripts to extract these points and perform a Fourier transform to see the characteristic S-shaped Doppler curve of the recorded signals (Figure 3).

Once the students know how the frequency changes, they use MATLAB to calculate the satellite's velocity with respect to the ground station and then use the results for orbit determination. The students also calculate the time of closest approach (TCA), the exact instant that the satellite passes directly overhead. They take pride in producing results that are more accurate than the TLE data they used for pass planning.

At one point we managed to simultaneously track Delfi-C3 and its sister satellite, Delfi-n3Xt, which was also designed at TU Delft. The signals from Delfi-n3Xt were strong because it was passing over the ground station. The signals from Delfi-C3 were weaker because it was over Iceland, but the satellite was still visible from the ground station (Figure 4).

FIGURE 2. *Left to right: Ground station UHF, GPS, and VHF antennas.*

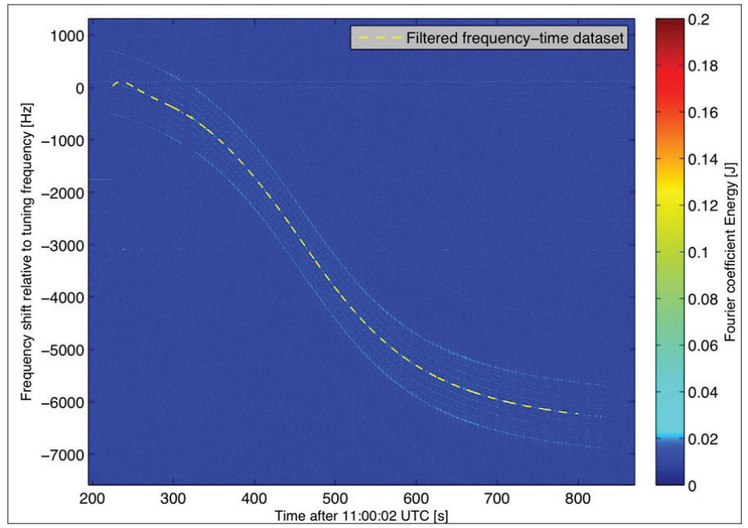


FIGURE 3. *A waterfall plot of the recorded signal from Delfi-C3. The characteristic S-shaped Doppler curve shows higher frequencies as the satellites approach the ground station and lower frequencies as they move away.*

Satellite tracking results are much more accurate when there are multiple ground stations. We designed our system to be affordable and easy to implement so that other universities could set up stations, gradually developing a worldwide network of satellite tracking stations. We are also investigating the feasibility of turning our ground station into a virtual lab that online students can access remotely. Here, software for operating the station and MATLAB scripts to extract tracking data will be available for download. Documents on how to design and build your own ground station will also be available.

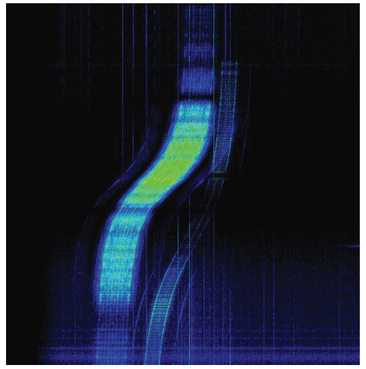


FIGURE 4. *A waterfall plot showing the recorded transmissions of Delfi-C3 as it passed over Iceland and its sister satellite, Delfi-n3Xt, as it passed over the ground station at TU Delft.*

Seeing the Bigger Picture

While we cannot match the satellite tracking accuracy of large space agencies with much more expensive equipment, I am impressed with what a few students using a home-grown station installed on a rooftop can achieve. Many of our students have gone on to positions at NASA, the European Space Agency, and aerospace companies. Engineers working in industry tell me that TU Delft graduates are in high demand because they have a reputation for seeing the bigger picture of engineering problems and rapidly coming up with solutions. MATLAB plays a big role in this reputation because our students use it throughout their studies to quickly try new ideas and show other engineers how their solutions will work. ▪

LEARN MORE

Hardware for Project-Based Learning (Student Project Support)
mathworks.com/hardware-project-based

Teaching Engineering Through Theory, Simulation, and Experimentation *37:00*
mathworks.com/video-81805

Visualizing Gravitational and Geophysical Phenomena

In his blog, DeepEarthScience, Bart Root writes about astrodynamics, space missions, geophysics, and other scientific topics. In one post, he writes about using MATLAB to calculate and plot the gravitational field of the 67P/Churyumov–Gerasimenko comet based on data from the ESA Rosetta satellite that is orbiting it (Figure 5).

Root also blogs about a project that is breathing new life into the work of Dutch geophysicist Vening Meinesz, who was a professor at TU Delft in the 1930s. Vening Meinesz invented a gravimeter, which he used on several submarine expeditions to measure Earth's gravitational field. Root and a group of undergraduates entered data that Vening Meinesz had recorded in his notebooks into spreadsheets. He then used MATLAB to analyze and plot the data for a new website chronicling Vening Meinesz' work.

Oceanic trenches were particularly interesting to Vening Meinesz. His measurements of the gravity anomalies near the Romanche Trench between Africa and South America produced results comparable with those obtained from state-of-the-art instruments today. Working in MATLAB and Mapping Toolbox™, Root plotted Vening Meinesz' path across the trench and compared the gravity anomalies he calculated with those produced by a high-resolution global gravity model currently used by scientists (Figure 6).

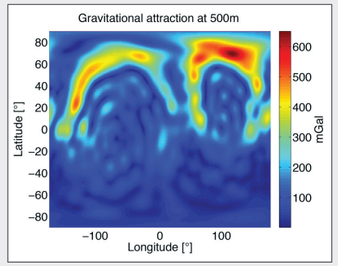


FIGURE 5. *Magnitude of the gravity vector for the 67P/Churyumov–Gerasimenko comet at 500 m altitude.*

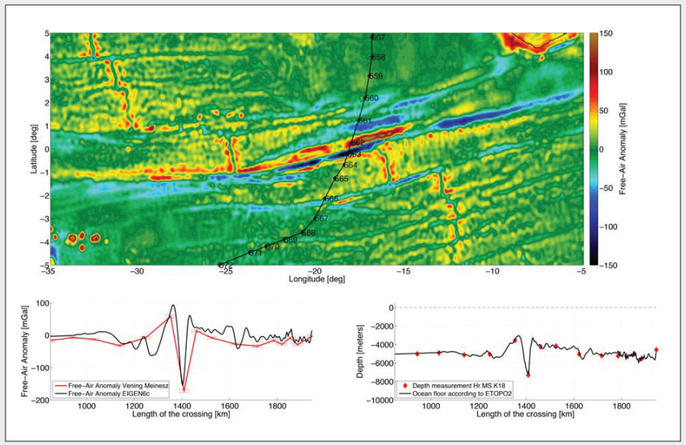


FIGURE 6. *Top: MATLAB plot tracing Vening Meinesz' path across the Romanche trench. Bottom: Plots comparing Vening Meinesz' gravity anomaly and depth measurements with current measurements.*

Data-Driven Insights with MATLAB Analytics: An Energy Load Forecasting Case Study

By Seth DeLand and Adam Filion, MathWorks

>> ENERGY PRODUCERS, GRID OPERATORS, AND TRADERS MUST MAKE decisions based on an estimate of future load on the electrical grid. As a result, accurate forecasts of energy load are both a necessity and a business advantage.

The vast amounts of data available today have made it possible to create highly accurate forecast models. The challenge lies in developing data analytics workflows that can turn this raw data into actionable insights. A typical workflow involves four steps, each of which brings its own challenges:

1. Importing data from disparate sources, such as web archives, databases, and spreadsheets
2. Cleaning the data by removing outliers and noise, and combining data sets
3. Developing an accurate predictive model based on the aggregated data using machine learning techniques
4. Deploying the model as an application in a production environment

In this article, we will use MATLAB® to complete the entire data analytics workflow for a load forecasting application. Using this application, utility analysts can select any region in the state of New York to see a plot of past energy load and predicted future load (Figure 1). They can use the results to understand the effect of weather on energy loads and determine how much power to generate or purchase. Given that the State of New York alone consumes several billions of dollars of electricity per year, the result can be significant for power generation companies.

Importing and Exploring Data

This case study uses two data sets: energy load data from the New York Independent System Operator (NYISO) website, and weather data—specifically, the temperature and dew point—from the National Climatic Data Center.

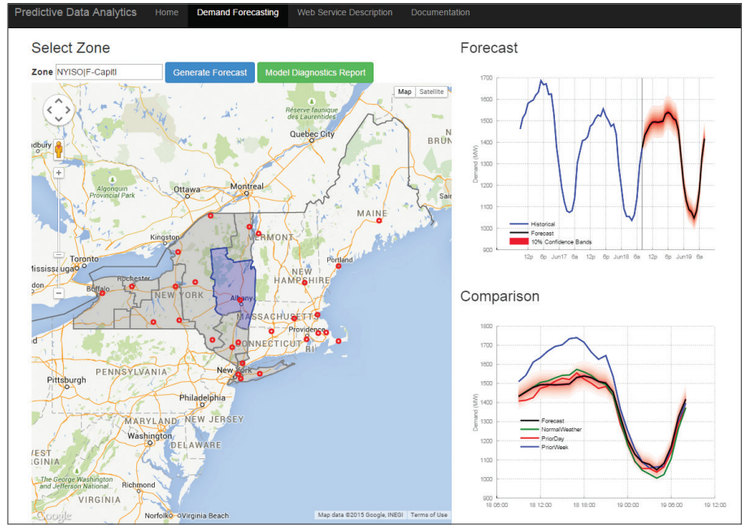


FIGURE 1. *MATLAB application for energy demand forecasting for New York.*

NYISO publishes monthly energy data in a ZIP file containing a separate comma-separated value (CSV) file for each day. The typical approach for working with data spread across several files is to download a sample file, explore it to identify the data values to be analyzed, and then import those values for the complete data set.

The Import Tool in MATLAB lets us select columns in a CSV file and import the selected data into a variety of MATLAB data structures, including vectors, matrices, cell arrays, and tables. The energy load CSV contains a time stamp, a region name, and a load for the region. With the Import Tool, we select CSV file columns and a target format. We can either import the data from the sample file directly or generate a MATLAB function that imports all files that match the format of the sample file (Figure 2). Later we can write a script that invokes this function to programmatically import all the data from our source.

Once the data has been imported, we generate preliminary plots to identify trends, reformat time and date stamps, and perform conversions—for example, by swapping the rows and columns in the data table.

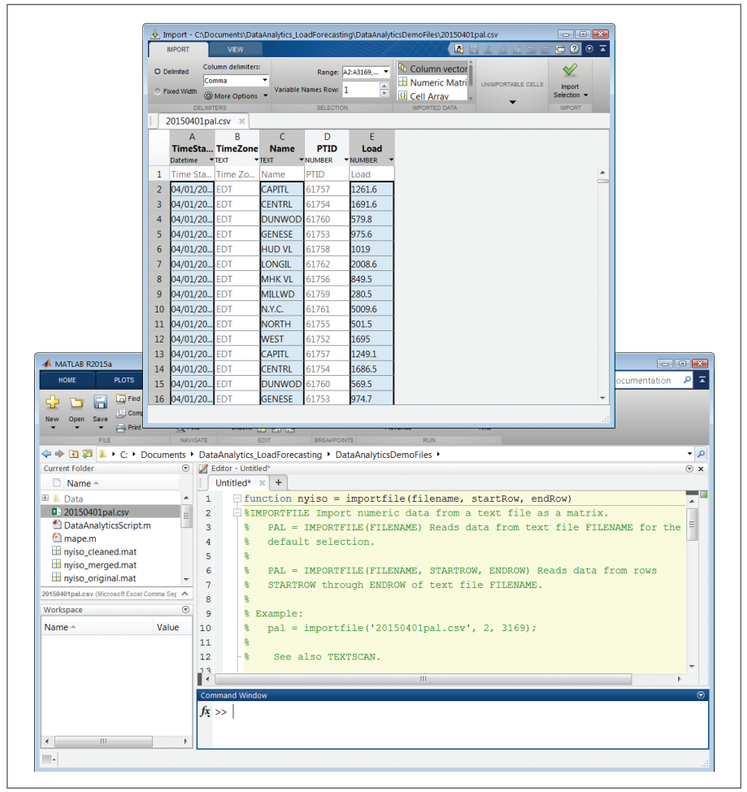


FIGURE 2. *CSV data selected for import and an automatically generated MATLAB function for importing the data.*

Cleaning and Aggregating the Data

Most real-world data contains missing or erroneous values, and before the data can be explored, these must be identified and addressed. After reformatting and plotting the NYISO data, we notice spikes in load that fall outside the normal cyclical rise and fall of demand (Figure 3). We must decide whether these spikes are anomalous and can be ignored by the data model, or whether they indicate a phenomenon that the model should account for. We choose to examine only normal cyclical behavior for now; we can address the spikes later if we decide that our model needs to account for such behavior.

There are several ways to automate the identification of the spikes. For example, we can apply a smoothing spline and pinpoint the

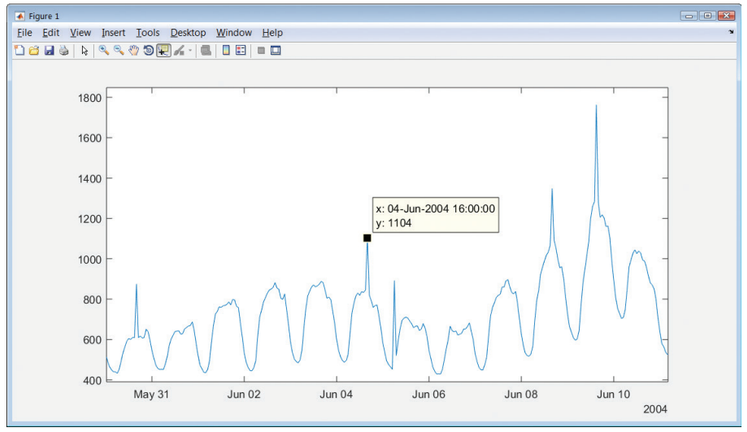


FIGURE 3. *Plot of energy load showing anomalous spikes in demand.*

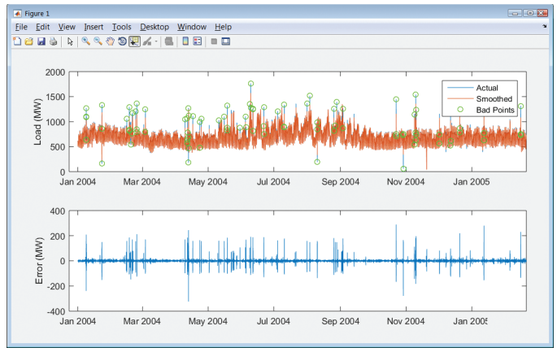


FIGURE 4. *Top: Plot of actual load and smoothed load with anomalies circled. Bottom: Plot of the difference between actual and smoothed values.*

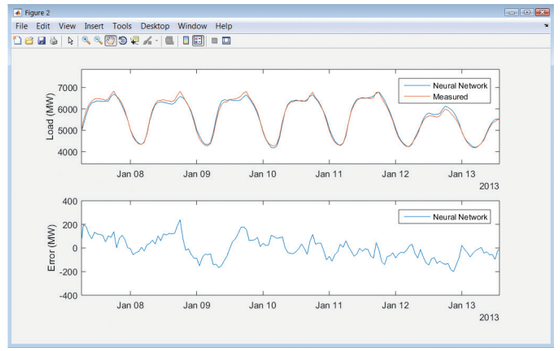


FIGURE 5. *Top: Plot of measured load and load predicted from a neural network. Bottom: Plot comparing measured and predicted values.*

spikes by calculating the difference between the smoothed and original curves (Figure 4).

After removing the anomalous points from the data, we must decide what to do about the missing data points introduced by their removal. We could simply ignore them; this has the advantage of reducing the size of the data set. Alternatively, we could substitute approximations for the missing values in MATLAB by interpolating or using comparable data from another sample, taking care not to bias the data. For the purposes of estimating load, we will ignore the missing values. We will still have enough "good" data to create accurate models.

After cleaning the temperature and dew point data using similar techniques, we aggregate the two data sets. Both data sets are stored in MATLAB table data types. We apply a table join in MATLAB by invoking the `outerjoin` function. The result is a single table giving us easy access to the load, temperature, and dew point for each time stamp.

Building a Predictive Model

MATLAB provides many techniques for modeling data. If we know how different parameters influence the energy load, we might use statistics or curve fitting tools to model the data with linear or nonlinear regression. If there are many variables, the underlying system is particularly complex, or the governing equations are unknown, we could use machine learning techniques such as decision trees or neural networks.

Since load forecasting involves complex systems with many variables to be considered, we'll opt for machine learning—specifically, *supervised learning*. In supervised learning, a model is developed based on historical input data (the temperature) and output data (the energy load). After the model is trained, it is used to predict future behavior. For energy load forecasting, we can use a neural network and Neural Network Toolbox™ to complete these steps. The workflow is as follows:

1. Use the Neural Fitting app in MATLAB to:
   a. Specify the variables that we believe are relevant in predicting the load, including hour of day, day of week, temperature, and dew point
   b. Select lagging indicators, such as the load from the previous 24 hours
   c. Specify the target, or the variable we want to predict—in this case, the energy load
2. Select the data set that we want to use to train the model, as well as a data set that we reserve for testing.

For this example, we opted for just one model. For most real-word applications, you would try several different machine learning models and evaluate their performance on training and test data. Statistics and Machine Learning Toolbox™ provides a variety of machine learning approaches, all using a similar calling syntax, making it easy to try out different approaches. The toolbox also includes the Classification Learner app for interactively training supervised learning models.

When the training is complete, we can use the test data to see how well the model performs on new data (Figure 5).

To automate the steps of setting up, training, and testing the neural network we use the Neural Fitting app to generate MATLAB code that we can invoke from a script.

To test the trained model, we run it against the data that we held in reserve and compare its predictions with the actual measured data. Results show that the neural network model has a mean absolute percent error (MAPE) of less than 2% on the test data.

When we first run our model against a test data set, we notice a few instances where the model's predictions diverge significantly from the actual load. Around holidays, for example, we see deviations from predicted behavior. We also notice that the model's prediction for load on October 29, 2012, in New York City is off by thousands of megawatts (Figure 6). A quick Internet search shows that on this date Hurricane Sandy disrupted the grid across the

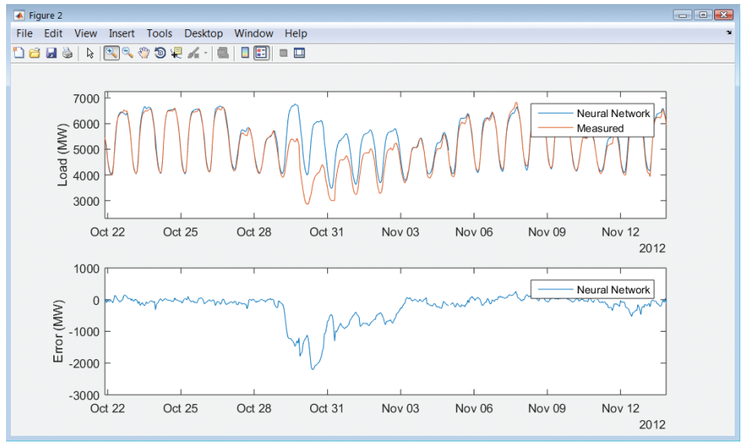


FIGURE 6. *Plot of measured load and predicted load for New York City on October 29, 2012.*

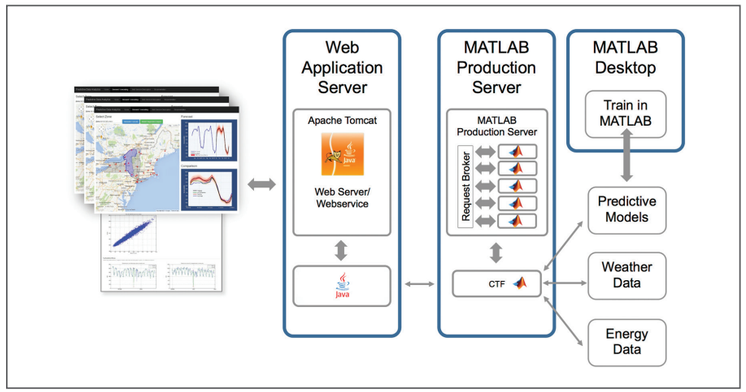


FIGURE 7. *Data analytics in MATLAB deployed in a production environment with Apache Tomcat and MATLAB Production Server.*

region. It makes sense to adjust the model to handle holidays, which are regular and therefore predictable occurrences, but a storm like Sandy is a one-off event and therefore difficult to account for.

The process of developing, testing, and refining a predictive model often requires numerous iterations. Training and testing times can be reduced by using Parallel Computing Toolbox™ to run several steps simultaneously on multiple processor cores. For very large data sets you can scale up by running the steps on many computers with MATLAB Distributed Computing Server™.

Deploying the Model as an Application

Once the model meets our accuracy requirements, the final step is moving it into a production system. We have several options. With MATLAB Compiler™ we can generate a standalone application or spreadsheet add-in. With MATLAB Compiler SDK™ we can generate .NET, and Java® components. With MATLAB Production Server™ we can deploy the application directly into a production environment capable of serving a large number of users simultaneously.

For our load prediction tool, we made the data analytics developed in MATLAB accessible via a RESTful API, which returns both numerical predictions and plots that can be included in an application or report. With the Production Server Compiler app we specify the MATLAB functions that we want to deploy. The app automatically performs a dependency analysis and packages the necessary files into a single deployable component. Using MATLAB Production Server we deploy the component as a processing engine, making the analytics available to any software or device on the network, including web applications, other servers, and mobile devices (Figure 7).

Next Steps

The energy load forecast model developed here provides highly accurate forecasts that can be used by decision-makers via a web front end. Because the model has been validated over months of test data, we have confidence in its ability to give a 24-hour forecast within 2% of actual load.

The model could be expanded to incorporate additional data sources, such as holiday calendars and severe weather alerts. Because the entire data analytics workflow is captured in MATLAB code, additional sources of data can easily be merged with the existing data, and the model retrained. Once the new model is deployed to MATLAB Production Server, the algorithm behind the load forecasting application is automatically updated—end users don't even need to refresh the web page. ▪

LEARN MORE

Data Analytics with MATLAB 47:25
mathworks.com/video-99066

Big Data with MATLAB
mathworks.com/big-data

Best Practices for Implementing Modeling Guidelines in Simulink

By David Jaffry, MathWorks

>> EMBEDDED SYSTEMS CONTINUE TO BECOME LARGER AND MORE complex. At the same time, design teams are becoming more dispersed, both geographically and in terms of team members' skills and experience. In this challenging development environment, implementing modeling guidelines to ensure modeling consistency is vital. Modeling guidelines establish a homogenous approach within the design team, making it easier to reuse the models for new projects. Guidelines also help new members of the team become familiar with your development process.

For some projects, implementing modeling guidelines is not simply a best practice; it is a requirement. Software systems deployed in high-integrity applications in aerospace and other industries must satisfy rigorous development and verification standards. Industry standards such as ISO 26262, EN 50128, IEC 61508, and DO-178C make modeling guidelines a prerequisite (Figure 1).

This article describes best practices for creating, implementing, validating, and promoting new modeling guidelines. The approach recommended will enable you to efficiently apply the standard within the team and ease the qualification of your modeling guidelines.

Defining Rules: Considerations and Best Practices

Modeling guidelines can and should meet several requirements, including readability, design reuse, and problem-free exchange of Simulink® models. The modeling guidelines enable the verification not only of models and model objects but also other artifacts, including object properties and workspace variables.

For this reason a modeling guideline consists of a set of rules. Each guideline lists the checks applicable to that guideline and provides detailed recommendations for resolving issues.

It is important to define rules that can be clearly understood and automatically verified. If a rule cannot be verified automatically, you will need to weigh the importance of having the rule against the time required for its manual verification.

Topics		ASIL			
		A	B	C	D
1a	Enforcement of low complexity[a]	++	++	++	++
1b	Use of language subsets[b]	++	++	++	++
1c	Enforcement of strong typing[c]	++	++	++	++
1d	Use of defensive implementation techniques	o	+	++	++
1e	Use of established design principles	+	+	+	++
1f	Use of unambiguous graphical representation	+	++	++	++
1g	Use of style guides	+	++	++	++
1h	Use of naming conventions	++	++	++	++

FIGURE 1. *Modeling guidelines required by the ISO 26262 standard.*

Table 1 provides examples of how ambiguously worded or subjectively defined rules can be improved.

Before creating a new rule, consider using the rules available in Simulink, Simulink Verification and Validation™, Embedded Coder®, IEC Certification Kit, and DO Qualification Kit. Most of the rules in these products are qualifiable and can be used in certification processes. By reusing these existing rules, you reduce the effort of creating your own modeling guidelines not only in the short term but also in the long term, as the rules are maintained for each new release of MATLAB.

Rule	Issue	Recommended Change
The model shall be readable.	Subjective	Replace with "Each sheet of the model shall contain less than 50 blocks. Maximum depth of subsystems is 10."
The model shall generate code.	Incomplete	Complete with "The model shall be compliant with Embedded Coder target `ert.tlc`."
The model should be optimized for efficiency.	Vague	Specify the model parameters that should be optimized.

TABLE 1. *Sample rules with recommended improvements.*

FIGURE 2. *Sample modeling guideline (from High Integrity Systems Modeling rule).*

Each rule in your modeling guidelines should use a template that includes the following elements:

- Title
- Identifier
- Description
- Rationale
- Level (mandatory or optional)
- Application examples (at least one example that complies with the rule and at least one that violates the rule, preferably with a workaround to enable the design team to deal with it)
- A reference to the individual responsible for implementing the check (for an existing rule)

In addition, the modeling guidelines should specify the version(s) of MATLAB and Simulink to which the rules apply. Verification options change from release to release. For example, in R2012b some verification tools support bus of arrays but not array of buses. By associating the rationale with the product version, you avoid including obsolete rules in your guidelines. The modeling guideline should also include references to all external standards documents. Figure 2 shows a sample guideline.

Implementing the Modeling Guidelines

The Model Advisor in Simulink Verification and Validation provides a framework for enforcing modeling guidelines. The Model Advisor checks a Simulink model for inconsistencies and for model objects that do not comply with the guidelines. Each guideline lists the checks applicable to that guideline and provides detailed recommendations for resolving issues.

The tool proposes many checks that can be reused. Alternatively, you can create custom checks using an API available through Simulink Verification and Validation. A Fix

action can be associated to a check to correct any violations found.

All rules implemented in the Model Advisor should use the same template. For each failed check, include its location, the problem, and a suggested fix. It is a good idea to include a "See also" paragraph that links directly to the modeling guidelines documentation.

Generally, one rule defined in the modeling guidelines matches one check in Model Advisor. Each rule will be implemented in a separate MATLAB file. The Model Advisor API eases the formatting of the check by proposing which MATLAB functions to include, as well as where to include images, tables, and formatting. Figure 3 shows an example of a formatting check.

The Simulink or Stateflow® APIs will be used mainly during the authoring of the check to verify the Simulink model. For example:

```
% Look for all 'Inport' blocks
using find_system function
inportBlockList = find_
system(bdroot,'BlockType',
'Inport');

% Get name of all Inport Blocks
using get_param function
inportBlockName = get_
param(inportBlockList,'Name');
```

Table 2 shows examples of functions from the Simulink API that you can use to create your own checks and verify some properties of the Simulink model.

Notice that some checks require an update of the Simulink model, as they need to manage data types or dimensions of signals or ports. A compiled model is very useful because it identifies all port data types and port dimensions for each block.

The check definition must use the following format to make it post-compile:

```
rec.Title = 'Check if root inport
name begins with "in_" prefix';

setCallbackFcn(rec, @checkRootInp
ortName,'None','StyleOne');

rec.CallbackContext =
'PostCompile';
```

A check that requires an update to the model starts with the character "^". For ease of identification it is best to separate these checks from others (for example, by placing them at the end of the list of checks). Note that checks can share data for efficient computation using callbacks.

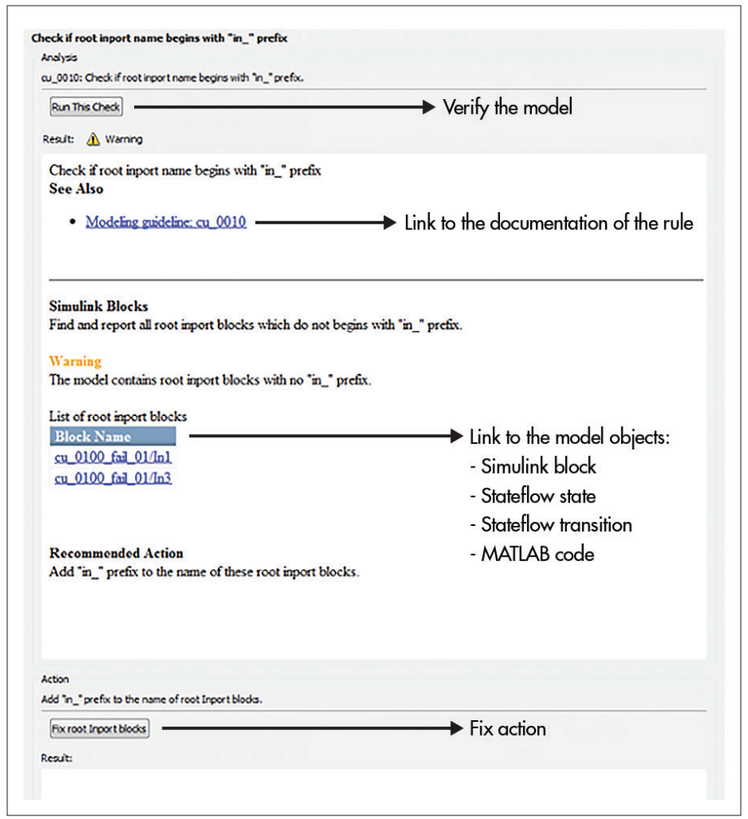


FIGURE 3. *Result of the check using the Simulink API.*

Function	Description
`find_system`	Finds systems, blocks, lines, ports, and annotations
`get_param`	Gets system and block parameter values
`bdroot`	Returns name of top-level Simulink system (useful during the writing of the check)
`gcb`	Gets pathname of current block (useful during the writing of the check)

TABLE 2. *Sample API functions.*

Validating the Modeling Guidelines

Provide failed and passed model examples for each rule. Keep these models as simple as possible to ensure that the check catches all violations.

You can use these models for the validation or certification of the modeling guidelines checks and for performing non-regression tests.

Figure 4 shows a sample architecture for test models.

Organizing and Promoting the Modeling Guidelines

The Configuration Editor in Model Advisor enables you to organize and keep only the rules you want to share. The custom configuration of Model Advisor is contained in a MAT-file that can be shared with others.

Figure 5 shows an example of modeling guidelines customized with the Configuration Editor. Notice that all unused checks have been removed from the Model Advisor interface.

Now you are ready to deploy the modeling guidelines to your design teams. The first step is to create a package that includes the following:

- Current version of your modeling guidelines
- Implementation of the checks
- Simulink models used to perform validation or non-regression tests
- Documentation
- Custom configuration or MAT-file created by the Model Advisor Configuration Editor
- MATLAB file `sl_customization.m` for deploying the custom configuration

It is a good idea to apply a subset of the modeling guidelines at the beginning of the project. Phased implementation will make it easier for the design team to adopt the new guidelines. It will also enable you to set checks relevant to a specific stage in your development process. ▪

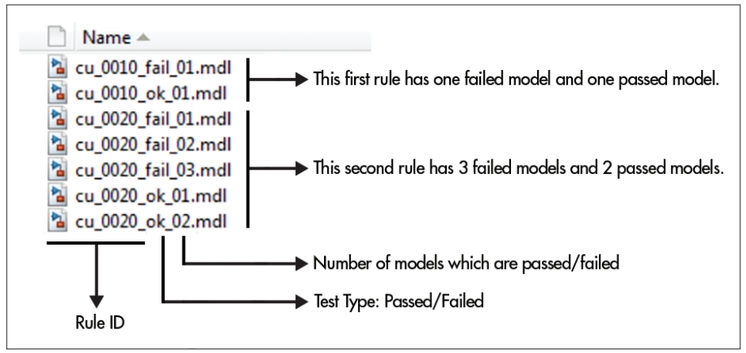


FIGURE 4. *Sample test model hierarchy.*

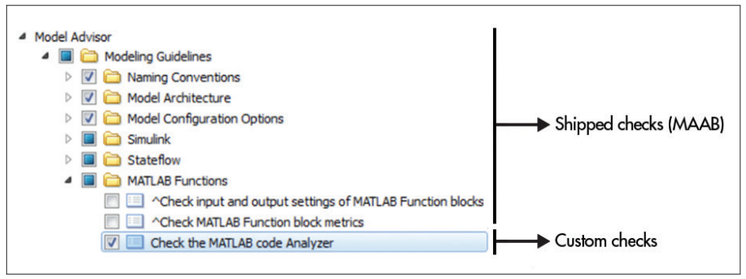


FIGURE 5. *Sample custom configuration.*

LEARN MORE

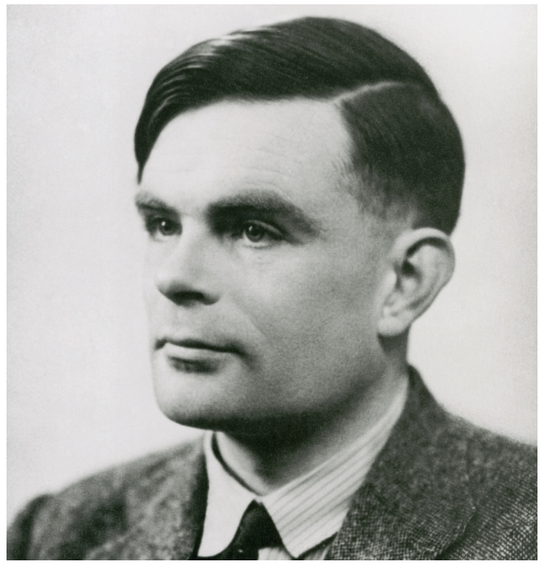
Accelerating Model Verification with Model Advisor
mathworks.com/model-advisor

Organizing Checks and Folders Using the Model Advisor Configuration Editor
mathworks.com/checks-folders

Alan Turing and His Connections to MATLAB

By Cleve Moler, MathWorks

The popularity of the movie "The Imitation Game" has focused public attention on Alan Turing, one of the 20th century's most important mathematicians. The film is about Turing's war-time code breaking and personal life. I am also interested in his unique contributions to science and his influence on MATLAB®.



Alan Turing, 1912–1954. Image courtesy Science & Society Picture Library.

Turing Machines

In 1936, as a 24-year-old student at King's College, Cambridge, Turing delivered a paper that was to become the foundation of modern-day computer science: "On Computable Numbers, with an Application to the *Entscheidungsproblem*." Loosely stated, the *Entscheidungsproblem*, or Halting Problem, asks "Is it possible to begin a proof that never finishes?" The question was first posed in 1928 by the great German mathematician David Hilbert. Hilbert challenged mathematicians to formalize the notion of mathematical proof and determine whether it is possible to state a proposition that can be neither proved nor disproved.

Turing formalized the notion of proof by introducing a theoretical machine. This machine had a finite number of states and an infinite tape containing a finite, but unlimited, number of zeros and ones. A set of rules (what we would today call its program) would cause the machine to read a single zero or one from the tape, then possibly erase it and write a new symbol in its place, move the tape one position to the left or right, and transition to a new state. That was all the machine could do. Turing showed that such a machine was capable of proving anything that could be proved by more complicated formalisms.

When he presented his paper, Turing was completely unknown in the field of mathematical logic. His approach was unprecedented. He used his formalism to show that the Hilbert *Entscheidungsproblem* had no solution—in other words, that it is possible to state mathematical propositions whose attempted proofs never terminate.

The implications of what we now know as "Turing machines" go far beyond this esoteric result. They are the basis of modern computer science, and, in fact, the theoretical basis for the actual computers that we use today.

Turing's paper was published in 1936, at about the same time as another paper on the *Entscheidungsproblem* by the American logician Alonzo Church, a professor at Princeton. Turing spent two years working under Church and getting his Ph.D. from Princeton. During this time he also built three of four stages of an electromechanical binary multiplier and began his study of cryptography.

Bletchley Park and Enigma

In 1939, Turing returned to England to join the newly formed, top-secret Government Code and Cypher School at Bletchley Park. This organization's task was to break the codes used by the German military to communicate with troops in the field, especially those on U-boats at sea. One of these codes employed the German cypher machine Enigma.

Early models of Enigma machines had been in use since the 1920s, so their basic principles were not secret. The details of the model used initially by the German military had been reverse-engineered by Marian Rejewski, a Polish mathematician in the Polish Cipher Bureau, in 1931. The machine has a typewriter-like keyboard, a plug board, three rotors, a reflector, and a display board. When a key is depressed, the electrical signal for that letter passes through the plug board, the three rotors, and the reflector. It then passes back through the rotors in the opposite direction and back through the plug board, where it lights the encrypted character on the display. At least one of the rotors then turns so that a different circuit is used for the next character.

Germany's Enigma cipher machine. Image courtesy Science & Society Picture Library.

Each day, three rotors were chosen from a set of five. Their initial positions were set according to code transmitted at the beginning of a message. These settings, together with the settings of the plug board and the reflector, determined the message encryption. The number of possible initial configurations was almost 1.6×10^{17}.

Building on work begun by Polish intelligence, Turing designed an electromechanical machine known as the Bombe. This machine would eliminate configurations that generated decrypted messages failing to contain a specified *crib*, or given plaintext. For example, digits were often spelled out, and so the word "*eins*", for "one", was a frequent crib.

Turing's group was so successful at decrypting German messages that Britain's military had to disguise their responses to hide the fact that they came from intercepted communications.

The Bombe was not yet a full-fledged computer—it had no memory, it was "programmed" by setting switches and inserting patch cords, and its logic units were relays.

An Automatic Computing Engine

Immediately after the war, universities and research laboratories in many countries began to build stored-program electronic computers. In February 1946, while he was working at the National Physical Laboratory (NPL) in Teddington, Turing presented his own design for a stored program machine that he called the ACE (Automatic Computing Engine).

The design for ACE was based on concepts from Turing's theoretical work on computability and on his code-breaking experience at Bletchley Park, where he had been involved with the development of a machine called Colossus. Colossus was built to help with the cryptanalysis of the German Lorenz cipher. It is often regarded as the world's first digital computer, although, since it did not have a memory, it was not a stored-program computer; it was programmed by paper tape, plugs, and switches. Turing hoped the ACE could be built by the engineering team that built Colossus, but the Official Secrets Act prevented him from explaining that his design for ACE could actually be implemented.

Turing's design for the ACE was far more ambitious than machine designs being proposed elsewhere. It included an early form of a programming language called Abbreviated Computer Instructions, and Turing believed that much of the work could be done in subroutines, with different sets of subroutines used for different tasks. His ACE was indeed inspired by a Universal Turing Machine.

Unaware of the computing successes at Bletchley Park, the NPL management was reluctant to approve Turing's elaborate project.

Introducing the MATLAB Enigma Emulator

Although the Enigma machine was a purely electromechanical device, the desire to break its code sparked some of the earliest innovations in software engineering. What would Enigma look like in software instead of hardware?

To find out, we developed a MATLAB app that simulates the Enigma machine (Figure 1). By happy coincidence, nine Enigma machines—one of the world's largest collections—were on display at the Museum of World War II in Natick, Mass., just two miles from the MathWorks headquarters. The museum gave us access to one of them while we were developing the app.

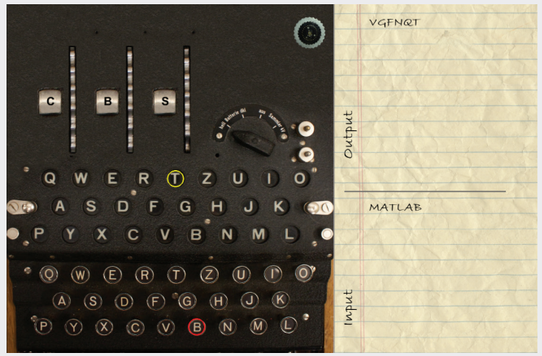


FIGURE 1. *The Enigma M3 Emulator MATLAB app.*

Figure 2 shows key components of the M3 Enigma machine. Pressing a key causes current to flow through a plug board, three rotors, and a reflector, lighting one of the letter lamps above the keys. Each individual plug, rotor, and reflector performs only a simple substitution cipher, but when they are used together there are over 150 quintillion possible combinations—and with at least one rotor turning one position with each key press, creating another combination, decrypting their messages was a colossal challenge.

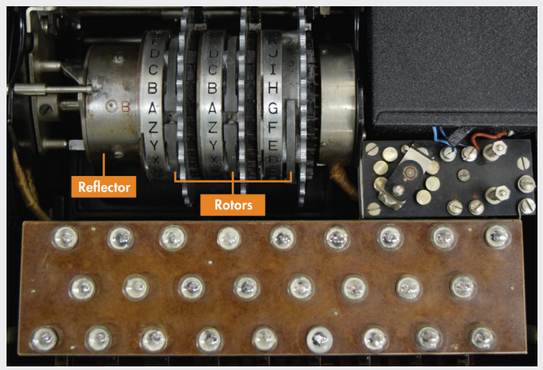


FIGURE 2. *Inside the Enigma machine. The reflector and rotors permute letters of the alphabet to encode or decode the message.*

All the Enigma encrypting components are modeled in MATLAB as permutation matrices. For example, the rotor on the right, which always rotates with each key press, is modeled by

```
I = eye(26);
Right = 'BDFHJLCPRTXVZNYEIWGAKMUSQO';
[~,p] = sort(Right);
R = I(p,:);
```

Additional permutation matrices, `L`, `C`, `F`, and `P`, for the left and center rotors, the reflector, and the plug board, are created with similar code, initialized with other strings.

To model the effect of a rotor advancing, the permutation matrix is modified using the MATLAB `circshift` function.

```
R = circshift(R,[-1,-1]);
```

A series of if-then statements determines which rotors advance during each key press.

A single character `cin` typed at the keyboard is converted into a logical column vector `xin` with

```
xin = logical(zeros(26,1));
k = cin-'A'+1;
xin(k) = 1;
```

Now the entire encryption process, transforming an input `xin` to an output `xout`, is neatly reproduced with code using matrix left division to model the path of the current through the rotors in the outbound direction, and matrix multiplication to model the inbound path.

```
M = L*C*R*P;
xout = M\F*M*xin;
```

The reflector matrix `F` is actually a symmetric permutation matrix, and so the matrix ultimately defining the transformation, `M\F*M`, is also symmetric. This is why the process of decoding an Enigma message uses the same initial settings as encoding one.

To complete the process, the output character is retrieved with

```
k = find(xout)+'A'-1;
cout = char(k);
```

The MATLAB Enigma M3 Emulator is available for download from the File Exchange on MATLAB Central.

LEARN MORE

Enigma M3 Emulator
mathworks.com/enigma-emulator

Turing became frustrated at the bureaucratic delays, and eventually left NPL. In 1948, he accepted a position at the University of Manchester, where computer development was proceeding more quickly.

A Growing Interest in Matrix Computation

In 1947, while he was still at NPL, Turing wrote a paper titled "Rounding-off Errors in Matrix Processing." This was before anyone had actual experience with automatic digital computers. Up to that point all computing had been done by hand. Although the terminology used in the paper is not quite what we would use today, the 13 section headings could serve as the syllabus for a modern course on matrix computation.

For example, Section 3 is "Triangular Resolution of a Matrix." The principal result is

> THEOREM ON TRIANGULAR RESOLUTION.
> *If the principal minors of the matrix* ***A*** *are non-singular, then there is a unique unit lower triangular matrix* ***L****, a unique diagonal matrix* ***D****, with non-zero diagonal elements, and a unique unit triangular matrix* ***U*** *such that* ***A*** *=* ***LDU****.*

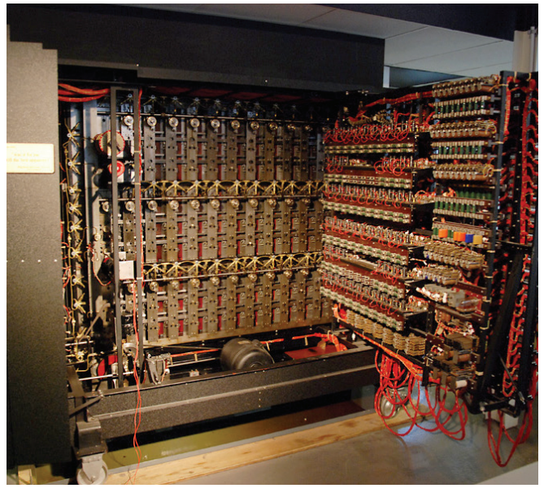
When we absorb **D** into **U**, this becomes the `LU` function in MATLAB. The result was not original with Turing. It also appears in another 1947 paper by von Neumann and Goldstine, which Turing references, and had appeared in other papers before then.

Section 4 is "The Elimination Method." Gaussian elimination is described in words and related to **LDU** "resolution." The fact that solving a set of n equations involves $\frac{1}{3}n^3 + O(n^2)$ multiplications is discussed. But there is no mention of the need for pivoting.

In Section 8, "Ill-Conditioned Matrices and Equations," Turing uses the term "condition number" for the first time in the literature, and presents some of its properties. However, the definition is not expressed in terms of what we now call a compatible matrix norm.

John Todd developed the notions of matrix norms and condition numbers in a series of papers in the 1950s. Several other authors wrote about condition numbers, as well. By 1959, when I studied numerical analysis with Todd at Caltech, the concept was on a sound footing. I don't remember if he mentioned Turing. Now we have `norm`, `cond`, and `condest` in MATLAB.

The last four sections of Turing's paper deal with round-off errors. Turing's entire analysis is based on a quantity denoted by ε, the error made in one step of elimination. This error can be observed and controlled in hand computation but not in automated computation on a modern machine with a fixed word length. Because Turing was not concerned with pivoting and scaling, his analysis does not consider the possibility that ε might grow during the elimination.



A recreation of the Bombe in the Bletchley Park museum.[1] Image courtesy Wikimedia Commons. commons.wikimedia.org/wiki/File:Bomba_turninga2.jpg

The Pilot ACE

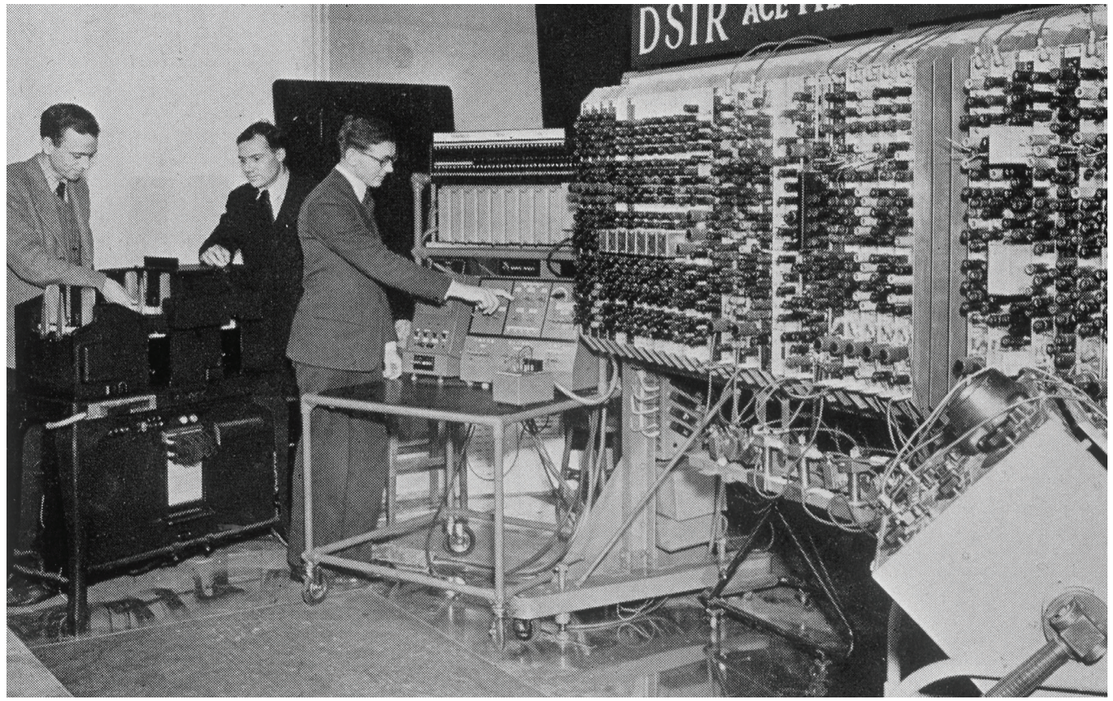
With Turing's departure from NPL, his assistant, J. H. Wilkinson, took over leadership of the group. A simplified version of the ACE, known as the Pilot ACE, was built instead of the full machine. The first program on the Pilot ACE was run in May 1950, and the machine was officially demonstrated in December of that year. Wilkinson designed and built the arithmetic unit. Since Turing's design meant that multiplication was done in a subroutine, Wilkinson was able to introduce floating-point arithmetic as soon as he realized it was desirable.

The Pilot ACE used mercury delay lines for its main memory. At first, there were only 128 32-bit words of memory. That was later expanded to 354 words. A 4096-word drum was added in 1954.

The machine was originally intended to be experimental, but it was so useful that it was kept in operation for several years. One application, publicized by the BBC, involved modeling metal fatigue in a Comet jet airliner. A commercial version of the Pilot ACE known as the DEUCE was produced by British Electric. Altogether, 33 Pilot ACE machines were made between 1955 and 1964.

Wilkinson and the Beginnings of MATLAB

Jim Wilkinson continued to head the Pilot ACE project, as well as work on numerical analysis. He went on to become the world's leading authority on matrix computation. His research on matrix eigenvalue algorithms was published as a series of papers, with various coauthors, in *Numerische Mathematik*. The work was eventually collected in *Handbook for Automatic Computation, Volume II, Linear Algebra*, 1971, with coauthor Christian Reinsch.



J. H. Wilkinson at the console for the launch of the Pilot ACE in 1950. Image courtesy Science & Society Picture Library.

Wilkinson's Algol procedures in the *Handbook* were translated to Fortran by a group at Argonne National Laboratory. This version became EISPACK (Eigensystem Package). I was part of the EISPACK project. I wrote the first version of MATLAB so that my students could get access to the package without writing Fortran.

I never met Alan Turing. I was 15 years old when he died. I wonder what he would have thought of MATLAB. ▪

For Further Reading and Viewing

Andrew Hodges, *Alan Turing: The Enigma*, Vintage, Random House, London, and Princeton University Press, *www.turing.org.uk/book.*

Cleve Moler, "Jim Wilkinson," *Cleve's Corner Blog, mathworks.com/cleve-wilkinson.*

"The Pilot Ace," *youtube.com/watch?v=Sf28IJmm-P4.* This rare video features Wilkinson and Mike Woodger, in 1982, reminiscing about the Pilot ACE computer. I recommend it to anyone interested in the history of computing.

[1]©2015 The MathWorks, Inc. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License."

LEARN MORE

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

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

Speed Up Your Simulations with Rapid Accelerator Mode

If you are doing Monte Carlo simulation or system optimization on your Simulink model, you'll need to run multiple simulations while varying model parameters. How do you get each simulation to run as quickly as possible?

In Simulink®, there are three desktop simulation modes: Normal, Accelerator, and Rapid Accelerator. For most models, Accelerator is faster than Normal, and Rapid Accelerator is faster still. Rapid Accelerator mode speeds up simulation by generating an executable for your model. The exact speedup varies depending on the model, but Rapid Accelerator can be more than 10 times faster than Normal mode.

This article shows how to set up your batch simulation script to get the maximum performance out of Rapid Accelerator.

Setting Up the Model

Each time you run a model or use the `sim` command, Simulink checks for changes in the model. Depending on the size of the model, this initialization phase can be time-consuming. In Rapid Accelerator mode, if you know that your model did not change structurally, you can skip this step by simply turning off the `RapidAcceleratorUpToDateCheck` option. For any model, no matter how large, this reduces initialization time to zero.

For illustration, we'll use a simple model of a bouncing ball. In this model, a ball drops from a certain height onto a hard surface. The position and velocity of the ball are measured at each step of the simulation. The parameters are gravity, the ball's initial height and velocity, and the coefficient of restitution (the bounciness of the ball).

Suppose we want to vary the coefficient of restitution, specified as k in the block dialog. To tune the value of k, we need to define it as a `Simulink.Parameter` object[1]:

```
k = Simulink.Parameter;
k.CoderInfo.StorageClass = 'SimulinkGlobal';
k.Value = -0.9;
```

Once this is done, we can explicitly build the rapid accelerator executable:

```
mdl = 'sldemo_bounce';
rtp = Simulink.BlockDiagram.buildRapidAccelera
torTarget(mdl);
```

The `buildRapidAcceleratorTarget` function returns a structure containing all the information relevant to tunable parameters in the model. Using this structure, we can create an array of runtime parameter structures for all the values we want to run:

```
k_values = [-0.9:0.1:-0.1];
for i = 1:length(k_values)
    paramSet(i) = Simulink.BlockDiagram.
                  modifyTunableParameters(rtp,'k',...
                  k_values(i));
end
```

We are now ready to simulate the model multiple times in Rapid Accelerator mode using the `sim` command:

```
for i = 1:length(k_values)
    simout(i) = sim(mdl,'SimulationMode','rapid',...
       'RapidAcceleratorUpToDateCheck','off',...
       'RapidAcceleratorParameterSets',paramSet(i));
end
```

Using this workflow with parallel computing is straightforward. All you do is replace `for` with `parfor` in the above code. ▪

[1]If you are using Simulink R2015a or earlier, you also need to turn on the "Inline Parameters" option in the Configuration Parameters.

LEARN MORE

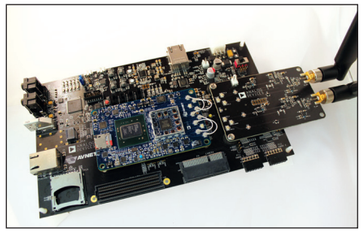
Guy and Seth on Simulink
blogs.mathworks.com/seth

Simulink Performance Improvements
mathworks.com/performance-improvements

Improving Simulation Performance in Simulink
mathworks.com/simulation-performance

Hardware for Wireless Testing with MATLAB and Simulink

Third-party products enable engineers, hobbyists, and students to prototype and test wireless system designs with MATLAB® and Simulink® using a range of hardware platforms. Developers can transmit and receive wireless data under real-world conditions and implement algorithms on programmable software-defined radio (SDR) hardware. High-performance testing equipment helps engineers validate their implementations of wireless standards with MATLAB.



Analog Devices, Avnet, and Xilinx Zynq-7000 SoC-Based SDR Hardware

With Xilinx® Zynq®-7000 All Programmable SoC-based hardware and Analog Devices® RF transceivers, developers can verify communications algorithms and prototype wireless systems. Xilinx Zynq-7000 All-Programmable SoCs combine a dual-core ARM® Cortex®-A9 processing system with Xilinx 7 series programmable logic on a single chip, enabling optimized, hardware- and software-based data processing. The Analog Devices AD9361 RF Agile Transceiver™ adds an RF front end, mixed-signal baseband section, and frequency synthesizers, tunable from 70 MHz to 6.0 GHz. Hardware support packages enable users to test their designs under real-world conditions by transmitting and receiving RF signals with MATLAB and Simulink. Developers can use Zynq-7000 SoC SDR support with HDL Coder™ to prototype and deploy custom radio designs on hardware. Zynq-7000 SoC-based SDR kits and modules are available from Avnet.

xilinx.com/zynq
avnet.com
analog.com/sdr
picozed.org/sdr

NooElec: RTL-SDR Hardware

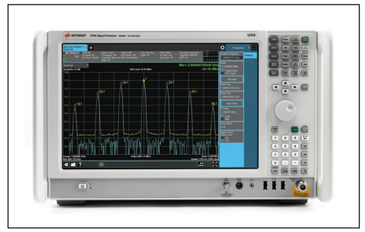
NooElec offers a series of low-cost USB dongles for students and hobbyists to develop broadcast radio, digital audio/video broadcast, GPS receivers, and other wireless applications. They can connect Communications System Toolbox™ to any RTL-SDR USB radio to learn communications concepts and validate systems developed in MATLAB and Simulink. The RTL-SDR radio support package enables users to design wireless receivers using actual signals at frequencies from 25 MHz to 2300 MHz.

nooelec.com

Ettus Research: USRP Hardware

The USRP™ product line includes the USRP Bus Series for low-cost experimentation, the USRP Networked and X Series for more sophisticated prototyping, and the USRP Embedded Series for production deployment, with support for frequencies from DC to 6 GHz. With Communications System Toolbox and the USRP hardware support package, engineers can use USRP radios as peripherals for importing live RF data I/O into MATLAB. The products support generating code with HDL Coder for deployment on USRP hardware.

ettus.com



Keysight Technologies: X-Series Signal Analyzers

Keysight X-Series Signal Analyzers provide calibrated instruments for capturing and analyzing test data, with calibration traceable to NIST and other physical science laboratories. The instruments are used by engineers engaged in cellular, wireless connectivity, MILCOM, SATCOM, and general-purpose testing. Test engineers can configure, control, and acquire data from the instruments directly from MATLAB using Instrument Control Toolbox™ and perform custom measurements using Communications System Toolbox. The instruments can also be used with LTE System Toolbox™ to analyze LTE standard-compliant wireless signals.

keysight.com

LEARN MORE

Software-Defined Radio
mathworks.com/sdr

Hardware Support
mathworks.com/hardware

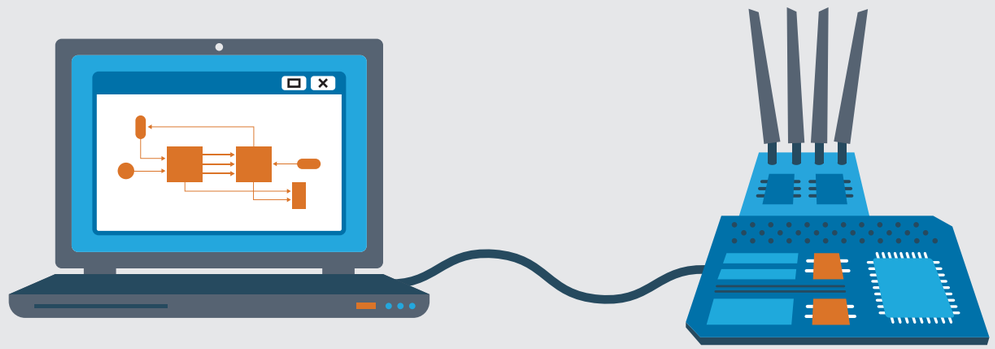
Third-Party Products and Services
mathworks.com/connections

Algorithm-to-Antenna Wireless Design

With MATLAB, you can take your wireless communication algorithms all the way to full system simulation, test, and hardware implementation.

1 SIMULATE SYSTEM
- Baseband algorithms
- RF front end
- Antenna arrays
- LTE and WLAN waveforms

2 TEST OVER THE AIR
- Software-defined radio
- RF instruments



3 PROTOTYPE AND IMPLEMENT
- FPGA, SoC, and SDR hardware
- HDL and C code

Explore the latest MATLAB® and Simulink® capabilities for wireless system design.
mathworks.com/wireless-systems

MathWorks®

Why does `fft` show harmonics at different amplitude?

How do I use `randperm` to produce a completely new sequence?

Can I pass arrays to and from a GUI?

How do I verify that a Support Package is installed?

Got questions? Get answers.

How do I calculate a loglikelihood value?

How do I pre-allocate memory when using MATLAB®?

Is it possible to write several statements into an anonymous function?

How do I use multiple colormaps in a single figure?

How do I reverse the order of a vector?

Can I plot the spectrum diagram of a signal?

Does `dec2hex` or `hex2dec` work on negative numbers (twos complement)?

MATLAB Answers

Tap into the knowledge and experience of over 100,000 MATLAB Central members.

mathworks.com/matlab-answers

92919v00 11/15