

MathWorks News&Notes

The Magazine for the MATLAB® and Simulink® Community

IN THE FRONT SEAT

with a Driver-in-the-Loop Automotive Simulator

ALSO IN THIS ISSUE

Power Converter
Control Software
for J-PARC Particle
Accelerator

Integrating Risk
Analytics and Modeling
in a Production
Enterprise Application

Longitudinal Controls
for a Self-Driving Taxi

Run MATLAB Image
Processing on Raspberry
Pi and NVIDIA

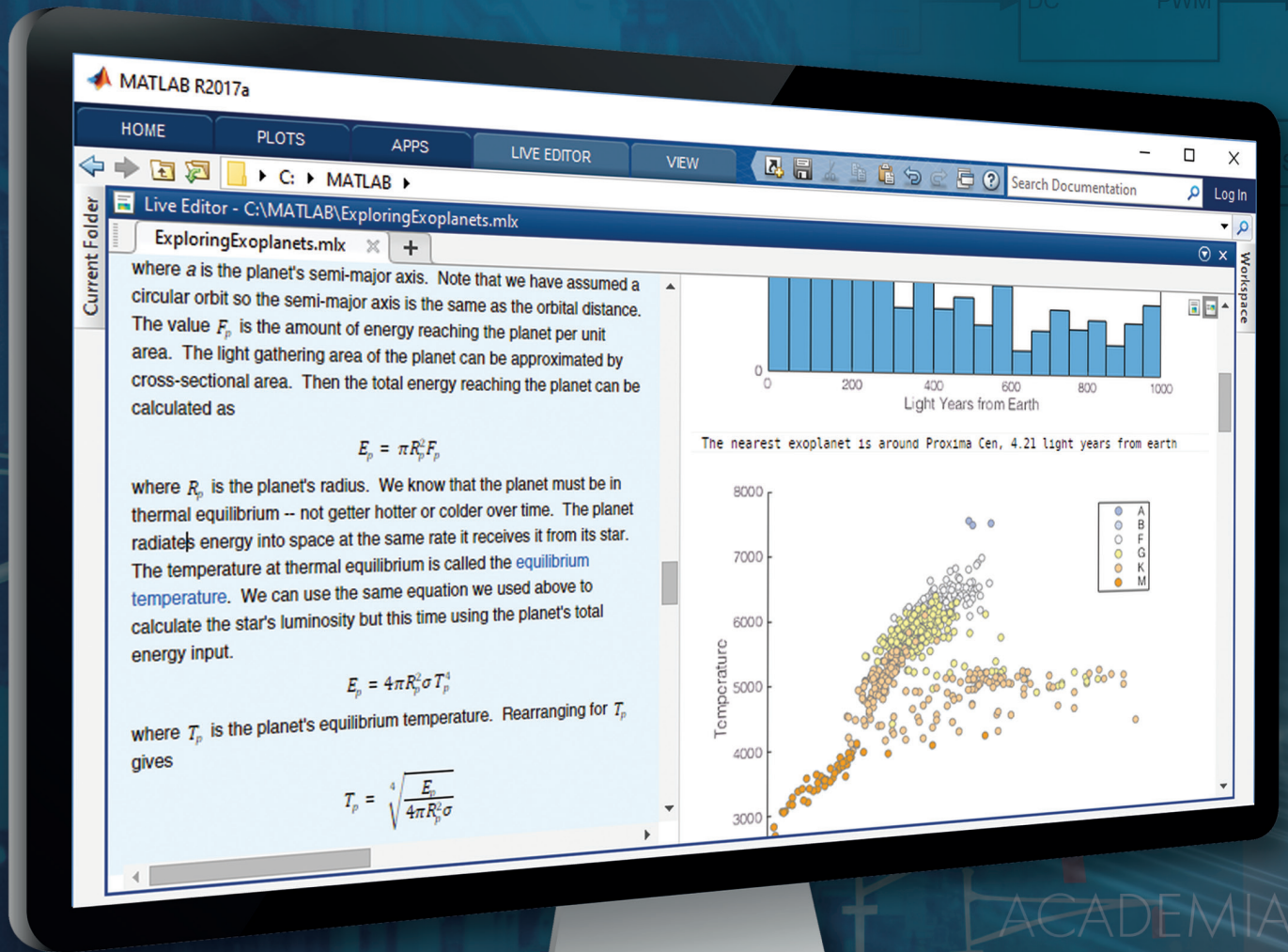
WIRELESS

R6
1.8mOhm

Vin 5V-10V

DEEP LEARNING

DATA



ACADEMIA

INTERNET OF THINGS

DATA ANALYTICS

MATLAB[®] & SIMULINK[®]

Engineers and scientists are using MATLAB and Simulink to advance our knowledge of the environment, make cars safer and smarter, discover new energy sources, develop new medical devices, and educate the next generation.

What will **you** do with MATLAB and Simulink?

MOTOR AND POWER CONTROL



FEATURES

4 Longitudinal Controls for a Self-Driving Taxi

"The very first passenger to ride in one of our self-driving taxis was a blind woman named Bev. Bev said that the ride felt much smoother than some vehicles driven by sighted people."

8 A Smartphone-Based Signals and Systems Laboratory

"The students generate C code from their MATLAB programs for mobile apps that they run on their Android or iPhone devices. This approach gives them first-hand experience with hardware constraints and implementation."

12 Integrating Risk Analytics and Modeling in a Production Enterprise Application

Swiss Re developers made their core risk models available as a production IT system.

16 Power Converter Control Software for the J-PARC Particle Accelerator

Japan's High Energy Accelerator Research Organization (KEK) engineered a power converter capable of delivering more than 100 MW to electromagnets used to control proton beams in J-PARC's main ring particle accelerator.

26 Putting Engineers in the Front Seat with a Driver-in-the-Loop Automotive Simulator

"We were astonished when dozens of students began showing up after school hours to ask us how they could improve their MATLAB algorithms and develop more advanced solutions."

32 Cleve's Corner: A Brief History of MATLAB

Cleve explains how a simple matrix calculator became the sophisticated technical computing language it is today.

HOW TOs

20 Beyond Image Classification: Apply Deep Learning

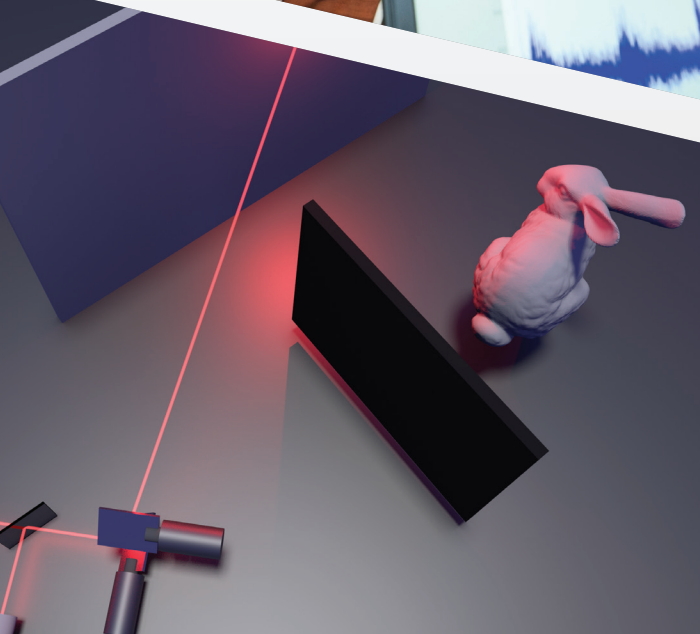
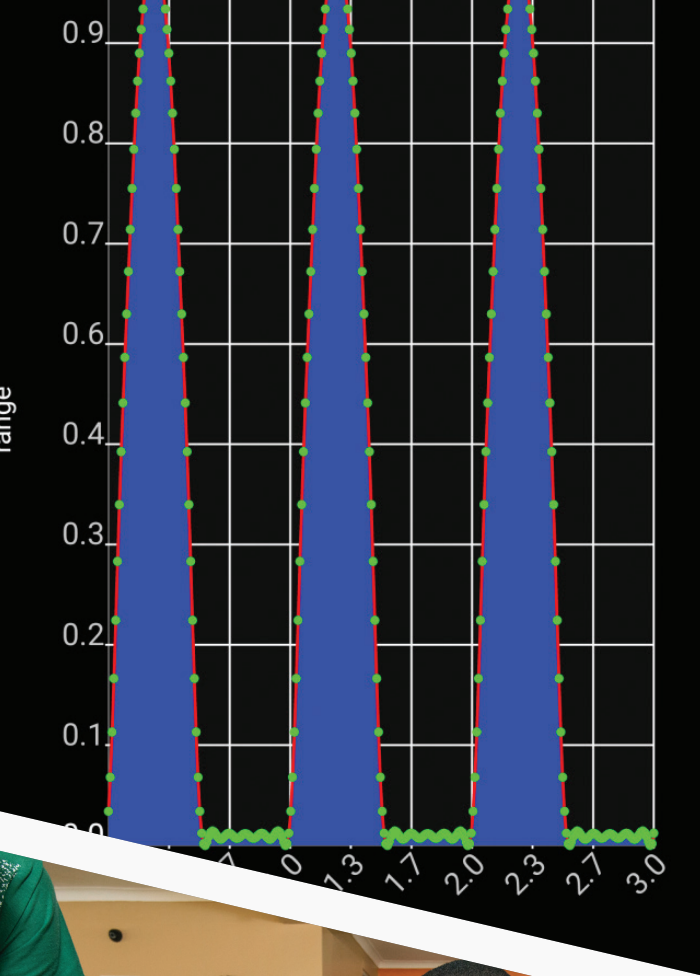
Whether the input is a D-signal, time-series data, or even text, CNNs offer the ability to process data in new ways.

30 Three Ways to Estimate Remaining Useful Life for Predictive Maintenance

Use these Predictive Maintenance Toolbox models for estimating RUL from lifetime data, run-to-failure data, and threshold values.

36 Run MATLAB Image Processing Algorithms on Raspberry Pi and NVIDIA Jetson

Prototype an algorithm on a Raspberry Pi board and deploy it to an NVIDIA Jetson Tx1 platform to achieve real-time performance.



QUICK READS

- 11 A Smart Jacket That Could Save Millions of Children's Lives
- 15 An Imaging Algorithm That Lets You See Around Corners
- 23 Third-Party Products: Extending Simulink for Complex System Simulation and Integration
- 24 MATLAB and Simulink in the World: Transformative Technology
- 41 What Does a Deep Learning Network See?

MANAGING EDITOR

Linda Webb

EDITOR

Rosemary Oxenford

GRAPHIC DESIGNER

Gabrielle Lydon

PRODUCTION EDITOR

Julie Cornell

TECHNICAL WRITER

Jack Wilber

PRINTER

DS Graphics

PRINT LIAISON

Jill Mespelli

EDITORIAL BOARD

Thomas Andrzejek, Michael Carone, Ye Cheng, Stacey Gage, Michelle Hirsch, Maureen Maher, Andy May, Cleve Moler, Sameer Prabhu, Richard Rovner, Loren Shure, John Stewart, Jim Tung

CONTRIBUTORS AND REVIEWERS

R. Agrawal, T. Atkins, A. Balu, P. Barnard, A. Baru, M. Belge, G. Bourdon, J. Brock, G. Campa, R. Cherukuri, B. Chou, M. Corless, K. Deeley, D. Doherty, O. Dufour, S. Eddins, M. Enelund, J. Erickson, J. Ghidella, H. Gorr, C. Grytberg, L. Harvey, L. Heske, A. Hosagrahara, K. Hyman, N. Ide, W. Jin, T. Jones, P. Kapur, N. Kehtarnavaz, D. Koh, S. Kozola, Y. Kurimoto, T. Kush, R. Lang, T. Lennon, R. Lawrence, K. Lorenc, P. Massano, A. Matsumoto, L. McNamara, R. Michaely, A. Mond, A. Nehemiah, D. Oswill, J. Paloschi, B. Patel, R. Pillat, P. Pilotte, J. Pingel, A. Poon, O. Pujado, H. Richardson, J. Rose, D. Sampson, K. Shibata, S. Tandon, B. Tannenbaum, A. Taylor, G. Thomas, T. Tohyama, A. Turevskiy, A. Urbain, R. van de Pol, T. Varga, V. Veerappan, S. Velilla, G. Venkataramani, S. Wilcockson, J. Zhao

SUBSCRIBE

mathworks.com/subscribe

CONTACT US

mathworks.com/contact

FIND US ONLINE



Printed on 30% post-consumer waste materials



SELF-DRIVING TAXI

Developing Longitudinal Control Algorithms to
Maintain Vehicle Stability and Ensure a Smooth Ride

By Alan Mond, Voyage Auto

The very first passenger to ride in one of our self-driving taxis was a blind woman named Bev. When asked about the drive, Bev said she felt safe—that the ride felt much smoother than some vehicles driven by sighted people.



FIGURE 1. A Voyage self-driving taxi on the road at The Villages community in Florida.

At Voyage, we want every passenger to feel that same level of safety and comfort. As a small startup competing against many larger organizations working on autonomous driving technology, we want to iterate as quickly as possible. One of our team's goals is to minimize the time between exploring ideas on the whiteboard and getting those ideas onto the road. To achieve that goal, we focused our efforts, scoping our first taxi service to operations in small communities (Figure 1), and refined our design through multiple iterations. We used Docker containers to manage system dependencies and the Robot Operating System (ROS) as the mid-

dleware for perception, motion planning, and controls. Instead of manually coding the model predictive control (MPC) algorithms for the longitudinal control system, we used Model-Based Design with MATLAB® and Simulink®.

Our team of three engineers completed the initial braking and acceleration control system in just two months.

Bounding the Complexity of the Self-Driving Car

Self-driving cars incorporate multiple complex systems to sense the surrounding

environment, plan a path to a destination, and control steering and speed (Figure 2). Compounding the challenge of designing and implementing these systems are all the objects and hazards in the environment, which include intersections, crosswalks, roundabouts, construction activity, pedestrians, U-turns, one-way streets, animals, and speed limits, not to mention the unpredictable driving patterns of other vehicles.

To simplify the control design challenge, we decided to deploy our first self-driving taxis in strategic partner retirement communities. Not only are these communities

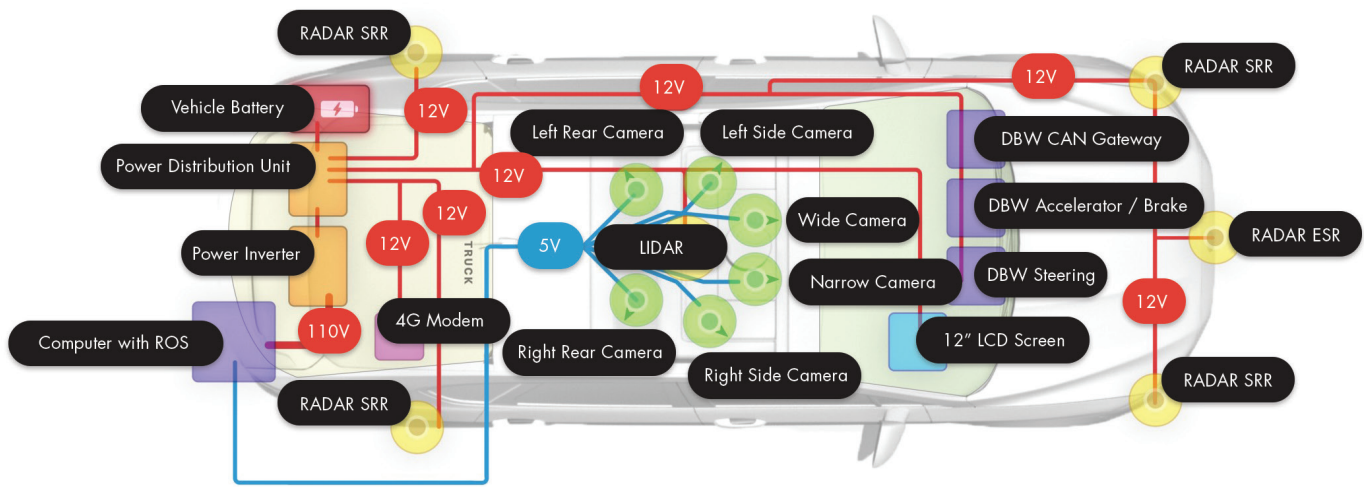


FIGURE 2. System overview of a Voyage self-driving taxi.

well-mapped and clearly defined, they also have set speed limits, typically 25 mph (40 kph).

Jumpstarting Development with an Adaptive Cruise Control System Example

First, our team researched ways to safely implement longitudinal control as rapidly as possible. We decided to begin with the MATLAB adaptive cruise control (ACC) system example . This example includes a Simulink model that uses MPC to implement an ACC system capable of maintaining a set speed or a set distance from a lead vehicle (Figure 3).

After downloading this model and running some preliminary simulations in Simulink, I generated C++ code from the model for a standalone ROS node with Robotics System Toolbox™ and Simulink Coder™. All the software for our self-driving taxi is modular, and each subsystem—perception, path planning, and longitudinal control, among others—runs as a ROS node. Within three

days we were running the generated code for the ACC in our vehicle.

Creating Our Own Model Predictive Controller from the Ground Up

While the ACC Simulink model had potential, it could not meet all our requirements. For example, the vehicle was too jerky when starting and stopping, and we found that riders are especially sensitive to this type of motion. (A passenger in our taxi will not necessarily feel how well the detection and perception algorithms are working, but they will immediately feel how well the longitudinal control works.)

We went back to the drawing board and designed a system from the ground up, quite literally going to a whiteboard and creating a kinematic model that describes the motion of the taxi based on first principles. We implemented this kinematic model in Simulink, using it as a foundation for the controller design. We then modified the parameters of the MPC model to meet our requirements

and incorporated additional logic to handle edge cases and scenarios that the original MPC model handled suboptimally, such as stop-and-go driving.

In these early stages of development, we imported gigabytes of data from rosbag log files into the MATLAB environment with Robotics System Toolbox, and filtered out all ROS topics not relevant to the longitudinal controller. Once the data was imported, we could access it like any other MATLAB variable, which made it easy to analyze and work with.

We simulated the control model in Simulink to make sure that its output, accelerator pedal position, and braking pedal position looked reasonable and that the model behaved as we expected for our target sets of inputs.

Conducting In-Vehicle Tests

The simulations gave us enough confidence in our control design to try it out in the car, with our team as the first passengers. We generated C++ code from the redesigned control

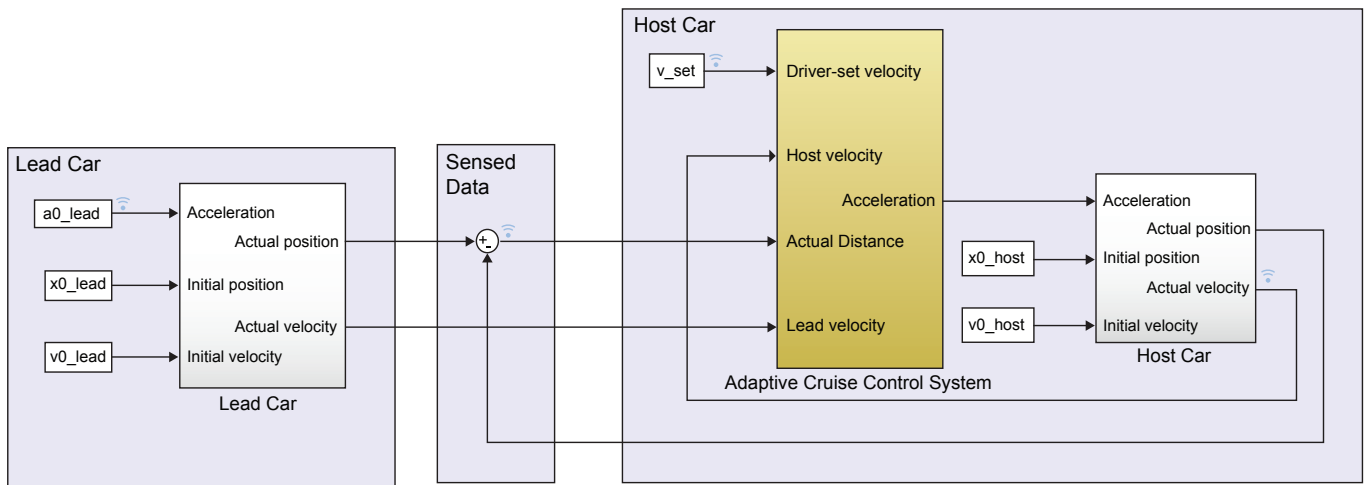


FIGURE 3. Simulink model of an adaptive cruise control system.

model for a ROS node and deployed the node to the vehicle within a Docker container. Docker enabled us to create an image of our production environment with all the necessary dependencies and then maintain and replicate that image consistently throughout development and testing.

During the initial in-vehicle tests, it was immediately apparent that our controller was too aggressive with acceleration and braking. Although the graphs we plotted during simulations showed what looked like smooth changes in velocity, the actual riding experience was anything but smooth. This realization highlighted for us the importance of quickly going from concept to onroad tests with Model-Based Design. We simply could not judge the quality of our design well enough in the lab; we had to experience it as our passengers would, in the car.

We completed several design iterations, tuning parameters and constraints, including limits on acceleration and jerk, as well as time constants and the rate at which outputs from the MPC were updated. We set up ROS

parameters in the Simulink model to make it easier for our colleagues to calibrate parameters directly via ROS. They could quickly update parameter values even if they had no prior experience with Simulink.

Creating Virtual Vehicles to Test Braking Scenarios

Because it would be unsafe to test scenarios in which another vehicle swerves into our vehicle's lane, we created a new type of ROS node to simulate a *ghost barrier*—essentially, a virtual vehicle that we could position at various distances from the taxi. We created this virtual vehicle in Simulink and parameterized it so that we could, for example, have it start at zero velocity and gradually increase speed. We generated code for the ROS node with Simulink Coder and then used the node to test and tune the controller's braking performance. With this node, which took only a few hours to develop, we could generate virtual obstacles in front of our taxi to see how it would respond, and then adjust its performance until it stopped safely and smoothly.

On the Road

The longitudinal controller we developed using Model-Based Design is in operation in self-driving taxis in the retirement communities that Voyage serves. We are seeing increased demand, with usage growing by 10% each week. Our engineering team is learning from data gathered during these rides, and we continue to refine the controller by incorporating what we learn.

LEARN MORE

Voyage
[voyage.auto](#)

Design an Adaptive Cruise Control System Using Model Predictive Control
mathworks.com/adaptive-cruise-control

Creating a Smartphone-Based Signals and Systems Laboratory for Undergraduate Engineering Students

By Nasser Kehtarnavaz, University of Texas at Dallas

All undergraduate electrical engineering students at the University of Texas at Dallas are required to take a course on signals and systems. We supplement this third-year course with a one-credit lab that gives students a chance to implement the concepts they learn in lecture.

For the past several years, I've taught *3102 Signals and Systems Laboratory* with an emphasis on hands-on coding. Students complete assignments on convolution, Fourier series, Fourier transform, and other key signals and systems concepts by writing short MATLAB® programs. I recently added a new dimension to the lab: The students now use MATLAB Coder™ to generate C code from their MATLAB programs to enable the creation of mobile apps that they run on their Android™ or iPhone devices.

This approach enables students to complete lab work anytime and anywhere, and gives them first-hand experience with hardware constraints and practical implementation issues. It also adds interest to the course. In their assessments of the most recent *Signals and Systems Laboratory*, students reported that the use of smartphones increased their engagement with the material.

Taking Lab Work Out of the Lab

Few universities have a lab associated with signals and systems, and those that do often involve running MATLAB code on desktop computer platforms. Because of the limitation on the number of computer platforms in a lab room, lab time is tightly scheduled. Last semes-

ter alone, about 200 third-year electrical and biomedical engineering students enrolled in *3102 Signals and Systems Laboratory*. Even after creating 10 sections, it is challenging to find lab time and space for that many students.

“I wanted to challenge the students without overwhelming them. With students new to programming, it was important to keep the focus on applying signals and systems principles instead of working through complex programming exercises.”

However, the processors in the mobile devices that students bring to class every day are more than powerful enough to run signal processing algorithms. By creating lab assignments that use the students' own devices as the processing hardware, I have enabled students to experiment anywhere and at any time. The school's Total Academic Headcount license, which provides students with campus-wide access to MATLAB on their own laptops, helped make this possible.

Creating a Framework for Signal Processing Mobile Apps

Before I could ask students with little programming experience to create mobile apps, I needed to give them a straightforward way to translate their MATLAB code into C code. I also needed a framework that they could use to run the C code on mobile devices.

I met the first requirement with MATLAB Coder, which enables

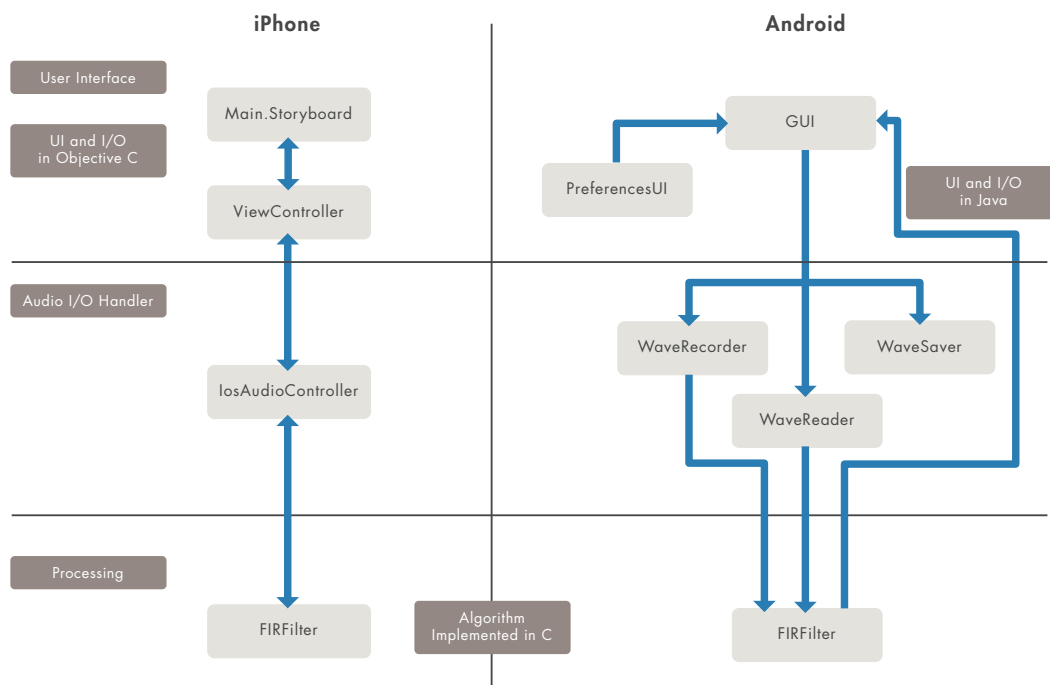


FIGURE 1. Diagram of programming shells used to run C algorithms on iPhone and Android devices.

students to generate efficient C code for their smartphones from MATLAB code that they develop and debug on their laptops. To meet the second requirement, I developed two programming shells that students can install on their mobile devices, one for iOS, written in Objective C, and one for Android, written in Java® (Figure 1).

Introducing MATLAB and Basic Programming

Because few of the students enrolled in *Signals and Systems Laboratory* have experience with MATLAB or with computer programming, I begin the course with an introduction to programming principles in MATLAB. The students first learn basic programming concepts, including arithmetic and vector operations, array indexing, memory allocation, and control flow. I then cover the more advanced techniques that they will need to complete their assignments, including loading and saving data, reading wave files, and generating signals.

Over the following two weeks, the students practice generating C code with MATLAB Coder and compiling apps in the smartphone programming environment for their specific device. All the

materials and assignments come from the book written for the lab course, *Anywhere-Anytime Signals and Systems Laboratory: From MATLAB to Smartphones*.

Developing Apps for Convolution, Fourier Series, and Fourier Transform

The first mobile app that the students develop is for a lab assignment on solving linear time-invariant (LTI) systems via the convolution integral. In this assignment, the LTI systems examined are RLC circuits. The students use MATLAB to perform a numeric approximation of the convolution integral and find the output voltage or current in response to a given input voltage or current. After developing and testing their solution in MATLAB, the students generate C code with MATLAB Coder. They then compile this C code along with the device-specific shells that I provide them to build an app for their mobile devices (Figure 2).

Next, the students explore Fourier series summation and reconstruction of periodic signals. They learn that if they know the response of a linear circuit to one sinusoidal input signal, they can obtain the

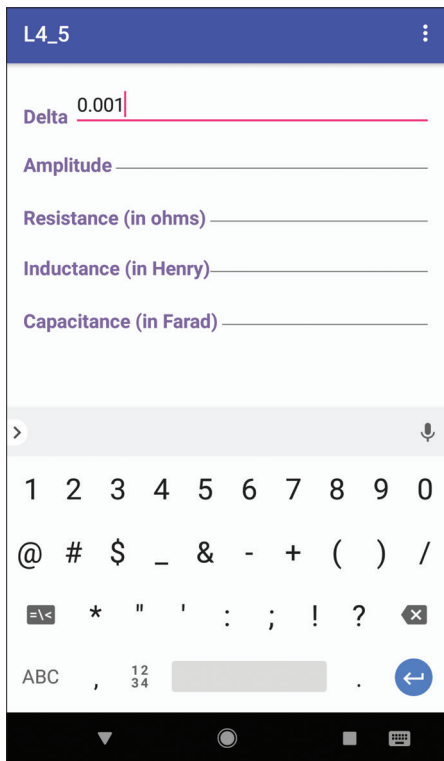


FIGURE 2. Mobile app for lab on linear time-invariant systems and convolution.

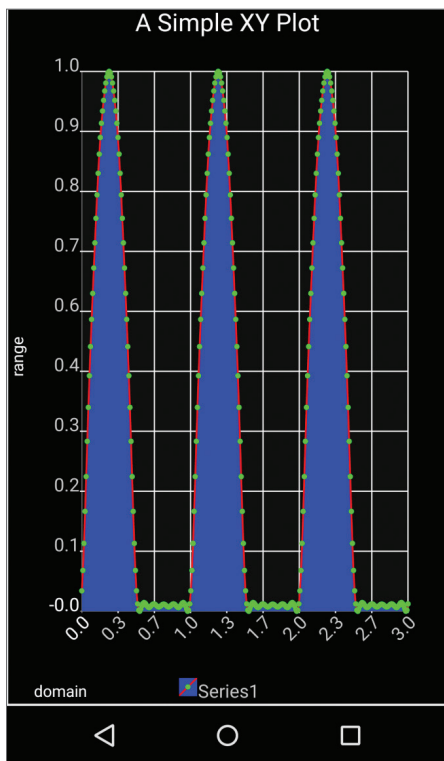


FIGURE 3. Mobile app for the Fourier series lab.

response to any periodic signal by decomposing it into sinusoidal signals and performing a linear superposition of the sinusoidal signals. As with the previous assignment, the students first develop an algorithm that demonstrates this principle in MATLAB and then use MATLAB Coder to implement that algorithm in C code for a mobile app (Figure 3).

In the remaining lab assignments, the students use this same process to build apps that demonstrate their ability to use Fourier transforms for noise cancellation and amplitude modulation and to perform analog-to-digital and digital-to-analog conversions via signal sampling, quantization, and reconstruction.

In creating the lab assignments, I wanted to challenge the students without overwhelming them. With students new to programming, it was important to keep the focus on applying signals and systems principles instead of working through complex programming exercises. For example, I allowed them to use the `conv()` and `fft()` functions in MATLAB rather than writing their own convolution and Fourier transform algorithms.

When I see the flash of recognition cross my students' faces, I know that the approach has worked. Some students, for instance, question why they need to move to the frequency domain with a Fourier transform if they've already solved a system with convolution. When they see the problems they encounter as the frames get longer and longer and they can no longer use convolution, they see exactly why this is necessary.

With MATLAB Coder, I can provide hands-on activities that enable the students to experience and overcome the challenges that come with implementing actual solutions in the real world. In doing so, I achieve better outcomes and engagement from the students.

LEARN MORE

Anywhere-Anytime Signals and Systems Laboratory:

From MATLAB to Smartphones

mathworks.com/signals-systems-lab

Mobile Device Shells and Sample Code

sites.fastspring.com/bookcodes/product/SignalsSystems-Bookcodes

[Bookcodes](http://sites.fastspring.com/bookcodes/product/SignalsSystems-Bookcodes)

iPhone and iPad Support from MATLAB Coder

mathworks.com/iphone-matlab

A Smart Jacket That Could Save Millions of Children's Lives

According to the World Health Organization, a child dies from pneumonia every 20 seconds. UNICEF reports that in sub-Saharan Africa, more than 490,000 children under 5 die from the disease each year. Diagnostic equipment and trained clinicians are scarce in remote areas, so the disease often goes undiagnosed.

Three engineering graduates from Makerere University, Uganda, developed Mama-Ope ("mother's hope"), a biomedical smart jacket that can diagnose pneumonia faster than a doctor. Designed for children under 5, the jacket works like a wearable stethoscope, using sensors to measure the patient's temperature, breathing rate, and wheezing levels. The results are recorded on a mobile app connected to Mama-Ope via Bluetooth, and sent to a healthcare professional for further analysis.

The team used MATLAB® signal analysis functions to filter and identify abnormal patterns in data collected by the device. The MATLAB analysis helped the team determine crucial parameters, such as the design of the filter and amplifier circuits.

Once the jacket is certified for clinical use in Uganda, the team intends to produce and supply it to countries throughout East Africa.

Read the full story

mathworks.com/fighting-childhood-pneumonia

Image credit: RAEng/Brett Eloff



Integrating Risk Analytics and Modeling in a Production Enterprise Application

By Tamás Varga, Swiss Re

As the world's second largest reinsurer, Swiss Re must take into account a broad set of risk factors from across the globe. For the past 10 years, we have calculated risk measures such as value at risk (VaR) and expected shortfall using the Internal Capital Adequacy Model (ICAM), a core risk model built with MATLAB®. As we continued to expand ICAM's capabilities over the years, however, it became increasingly hard to manage the complexity. Numerous interdependencies made it difficult to fully understand how the model worked.

To make ICAM easier to understand, update, and maintain, we completed a major overhaul. We made the revamped ICAM core available as a production IT system—the Integrated Risk Analytics and Modeling Platform (IRAMP)—and sped up risk modeling calculations by executing them on a computer cluster. MATLAB, MATLAB Production Server™, and MATLAB Distributed Computing Server™ enabled us to achieve both these objectives without having to develop custom IT infrastructure.

Applying Object-Oriented Programming to Improve ICAM Transparency and Maintainability

ICAM is designed to enable risk reporters to understand the aggregate effect of approximately 300,000 risk factors on the company's total economic balance sheet. Categories include interest rates, equity prices, real estate prices, credit spreads, and claims inflation, as well as operational risks, natural disasters, and mortality trends. In rewriting ICAM, we wanted to make it easier for risk reporters to see how

these factors affected risk measures. One of our most effective changes was to apply more object-oriented programming principles in writing the MATLAB code. Today's version of ICAM has more 75,000 lines of MATLAB code—all under version control—comprising 400 data classes and 250 classes for risk factors and loss functions. The graphics and objects classes in our code have enabled us to increase the number of user interfaces in ICAM and to control them in a maintainable way (Figure 1).

Building an Enterprise Application for Risk Analysis

Calculating VaR, expected shortfall, and other risk measures with a one-year horizon from 300,000 risk factors is a computationally intensive process, involving Monte Carlo simulations in which 1,000,000 realizations are generated for each risk factor. We are using Statistics and Machine Learning Toolbox™ for regression, generalized linear models, and data compression and preparation as well as Monte Carlo simulations with random samples drawn from a variety of distributions.

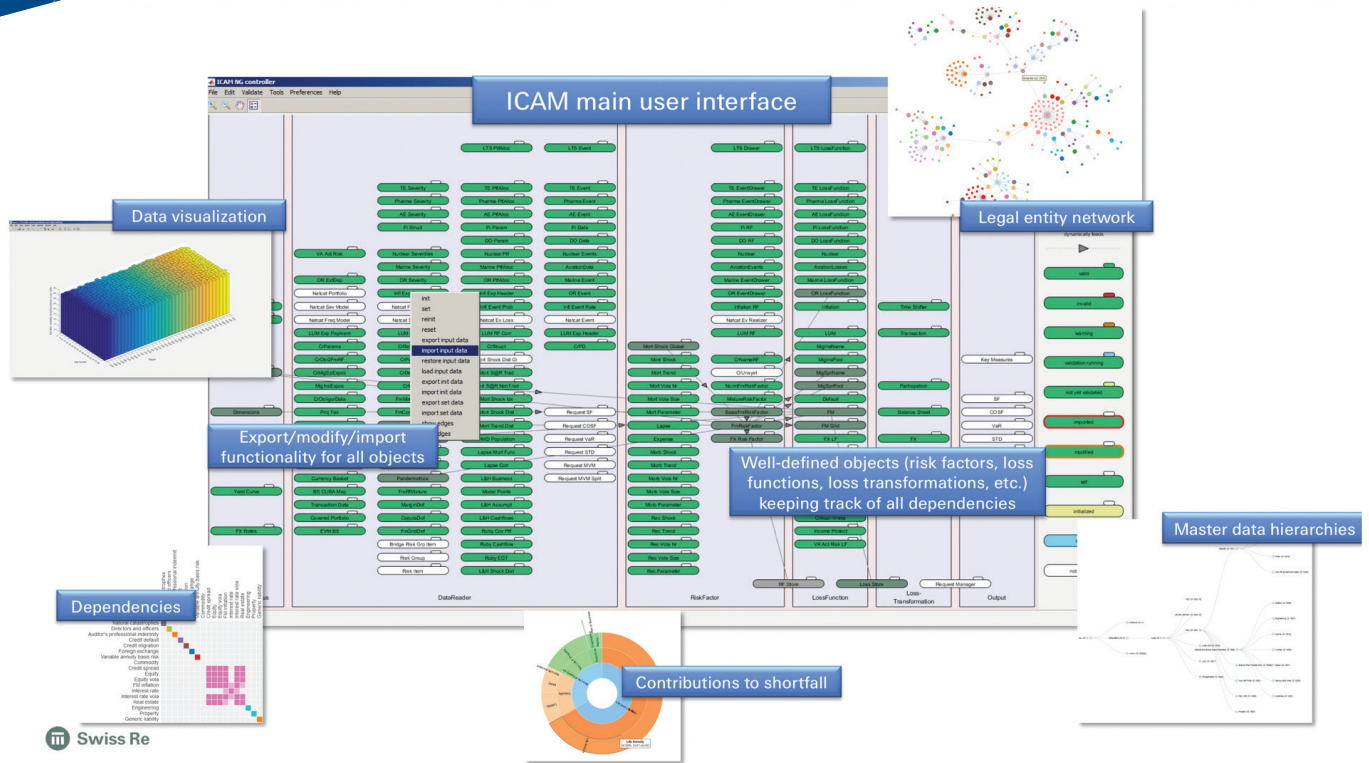


FIGURE 1. ICAM user interface.

We employed a three-part strategy for building an enterprise IT application to manage the lengthy compute times this process requires. First, we set up a computing cluster to support parallel computations with Parallel Computing Toolbox™ and MATLAB Distributed Computing Server. Second, we broke the process down into multiple distinct workflows, including validation, preprocessing, calculation, and evaluation. Third, we used MATLAB Production Server to establish a production IT framework that risk reporters could use to execute multiple workflows on the computing cluster.

We maintain two environments for developing and maintaining ICAM, one for production and one for development and training. The computing cluster in our production environment includes 165 workers. Our development and training environment has a similar computing cluster with 111 workers (Figure 2). After validating our ICAM application in the development and training environment, we prepared it for deployment in the production environment by compiling it into a standalone component using MATLAB Compiler SDK™.

Workers in the cluster are allocated as needed to complete workflows initiated by risk reporters. Each workflow is initiated from the IRAMP web interface and orchestrated by MATLAB Production Server. To begin the process, for example, risk reporters initiate the *validate* workflow, which verifies that the input data is internally consistent. Next, they kick off the *preprocessing* workflow, which transforms the raw input data into a format ready for use by the risk model. In the *calculate* workflow, all the Monte Carlo simulations are performed. This workflow requires the largest number of workers and the most time to complete. The results are stored as a snapshot in a 200 GB file on a shared file system. In the *evaluate* workflow, the risk reporters use a MATLAB application that we created to query results from the image and perform what-if analyses.

From Desktop to Cluster to Cloud

The overhaul of ICAM and development of IRAMP have been well received by risk reporters because the system is more transparent from end to end. While MATLAB provides a powerful and efficient

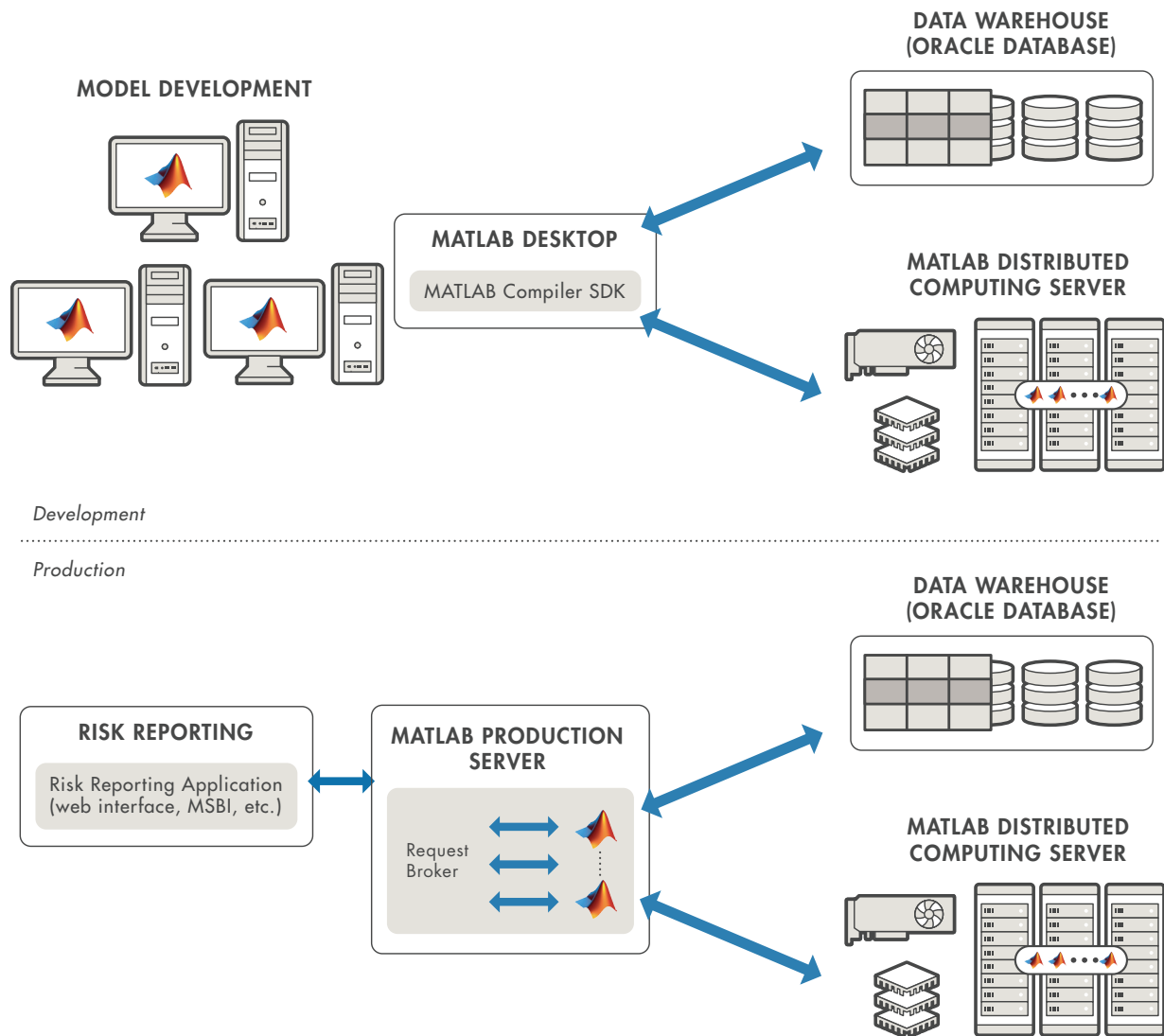


FIGURE 2. IRAMP system architecture for development and production environments.

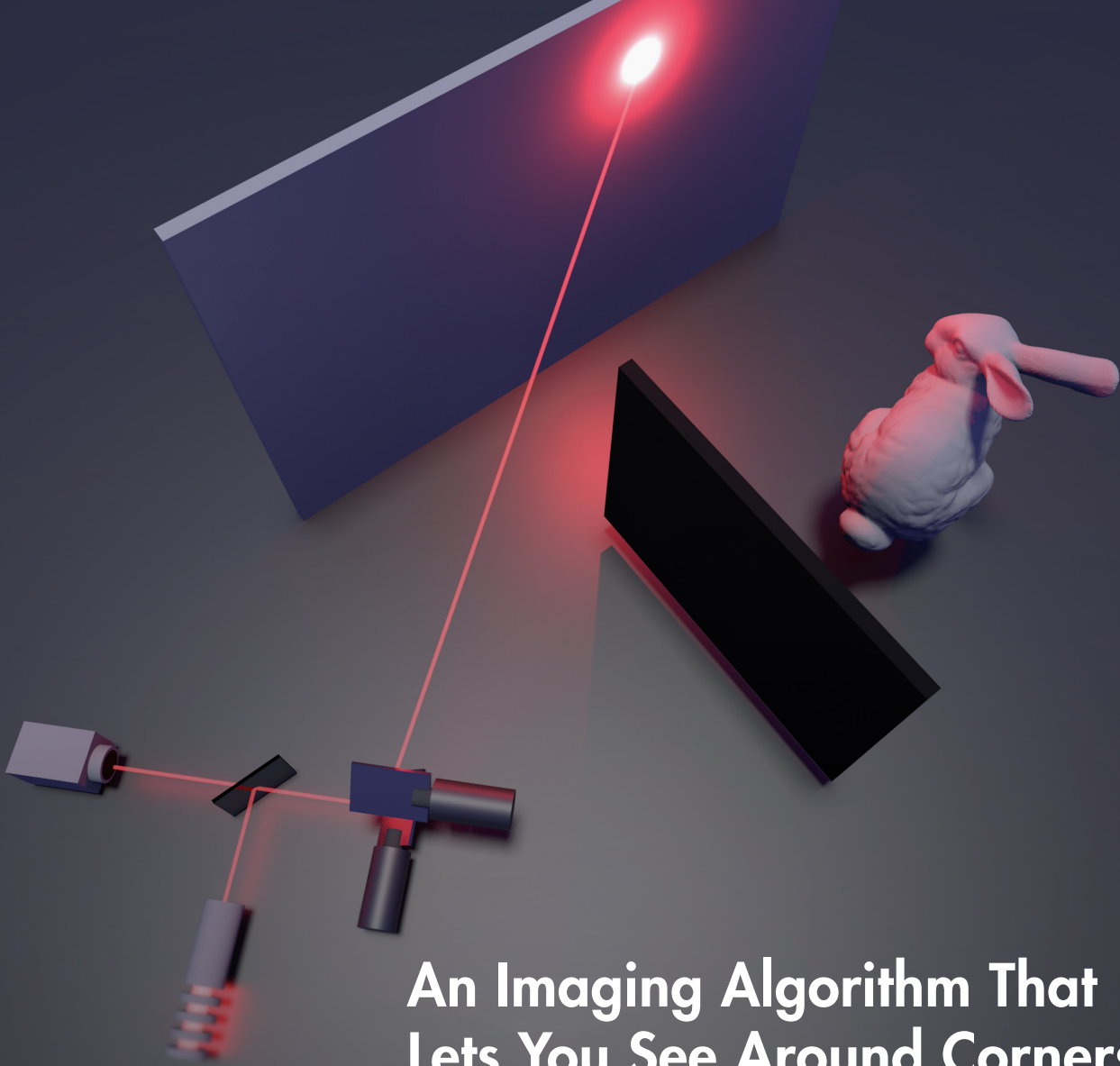
development environment, by using MATLAB Production Server and MATLAB Distributed Computing Server for both the development and production environments, we ensure consistent results and increased stability in production.

We are now working with MathWorks engineers to migrate the IRAMP system to an external cloud-based system such as Microsoft® Azure®. This will provide for a larger scale and more flexible system, allowing us to reduce costs by scaling down during periods of low demand and to reduce wait times by scaling up during periods of high demand.

LEARN MORE

MATLAB Production Server for Financial Applications (38:38)
mathworks.com/video-81937

Aberdeen Asset Management Implements Machine Learning-Based Portfolio Allocation Models in the Cloud
mathworks.com/aberdeen



An Imaging Algorithm That Lets You See Around Corners

Stanford University's new laser-based imaging technology could take blind spot detection in cars to a whole new level. Not only can it see things the driver can't, it "sees" things that are not even in the line of sight.

The Stanford system can detect objects, in 3D, hidden behind walls and around corners. It uses a pulse of laser light and a photon detector to capture light that scatters off a wall and reflects off objects hidden from view. A photon detector sensitive enough to detect a single photon creates a "scan" of the reflected light pulses. A computational reconstruction algorithm, created with MATLAB®, uses information from the scan to infer the 3D shape of the hidden objects.

Read the full story

blogs.mathworks.com/headlines/2018/03/16/imaging-algorithm-lets-you-see-around-corners-with-laser-pulses/

See how the system works

computationalimaging.org/publications/confocal-non-line-of-sight-imaging-based-on-the-light-cone-transform/

Image credit: Stanford Computational Imaging Lab



NEUTRINOS AT J-PARC

Developing Power Converter Control Software
for the J-PARC Particle Accelerator

By Yoshinori Kurimoto, High Energy Accelerator Research Organization (KEK)

Results from the T2K experiment suggest that neutrino oscillations may hold the key to understanding a fundamental question about the universe: why it contains vastly more matter than antimatter when the Big Bang is believed to have produced equal amounts of both.

The T2K experiment is a long baseline neutrino oscillation experiment in which neutrinos and antineutrinos produced at the Japan Proton Accelerator Research Complex (J-PARC) are observed in the Super-Kamiokande detector located 295 km away. Finding a difference in oscillations between neutrinos and antineutrinos would provide an essential clue about how our universe was formed. The largest task of the experiment is the production of numerous neutrinos and antineutrinos. In the T2K experiment, neutrinos are created with the J-PARC proton

accelerator by accelerating protons to near light speed and smashing them into a target material. To expand our investigation of neutrino oscillations, we need to produce more neutrinos by increasing the rate at which we supply protons via the accelerator. Then, once the proton beams enter the main ring, we need more powerful electromagnets to control the beams as they travel around the ring (Figure 1).

None of the manufacturers we usually worked with were able to engineer a power

converter that could deliver the power needed for these stronger electromagnets within our budget. We therefore decided to help the engineering effort by developing the control software ourselves.

Neutrino research is an area of intense competition, and we need to keep pace with labs in the U.S. and Europe that are engaged in similar research. To speed development and keep down costs, we developed the power supply control software using Model-Based Design with Simulink® and deployed it to

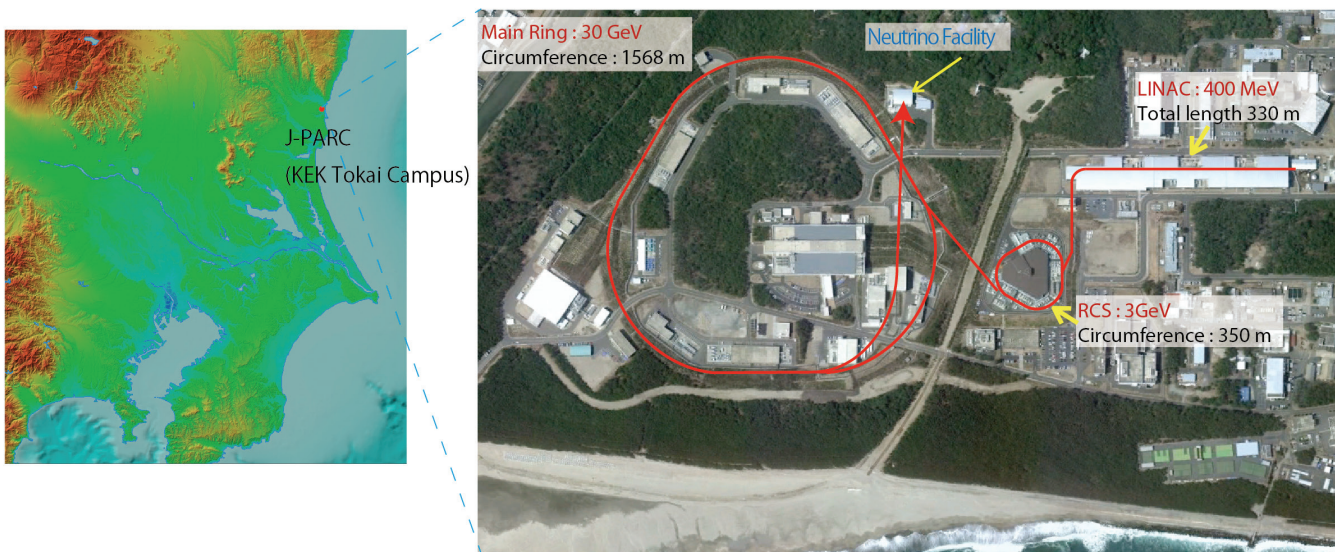


FIGURE 1. Bird's-eye view of J-PARC showing the main ring and path of proton beams in red.

an FPGA using HDL Coder™. Model-Based Design enabled us to develop the control software at a cost 60% less than the estimates provided by major manufacturers and to cut development time by more than 50%.

Our Challenge: Almost Double the Voltage Supplied to J-PARC Electromagnets

To appreciate how important a larger power supply was to our research, it helps to understand the process for generating and detecting neutrinos at J-PARC. First, we use a linear accelerator to accelerate negative hydrogen ions to about 400 million electron volts (MeV). With the J-PARC synchrotron, we convert the ions to protons and accelerate the protons to 1.3 billion electron volts (GeV) in J-PARC's small ring, which is about 350 meters in circumference. The protons are then directed to the main ring (about 1.5km in circumference), where they are accelerated to 30 GeV before being targeted to the neutrino generation facility. In the final stage, the neutrinos are observed at the neutrino observatory located under Mount Ikeno, 295 km away.

In the main ring (Figure 2), bending and quadrupole electromagnets control the proton beams' trajectory by applying precisely synchronized magnetic fields.

For our upcoming experiments, we need to supply more protons, which means reducing the amount of time needed to switch (or cycle) the electromagnet from 2.48 seconds to 1.3 seconds. The time required to switch an electromagnet is inversely proportional to the voltage applied, which means that we have to almost double the voltage, corresponding to the total output power of approximately 100 MW—more than the electrical grid is capable of providing.

Designing and Implementing the Power Converter Controller

The converter has two main components: a three-phase AC-to-DC voltage converter that is used to charge large capacitors, and a

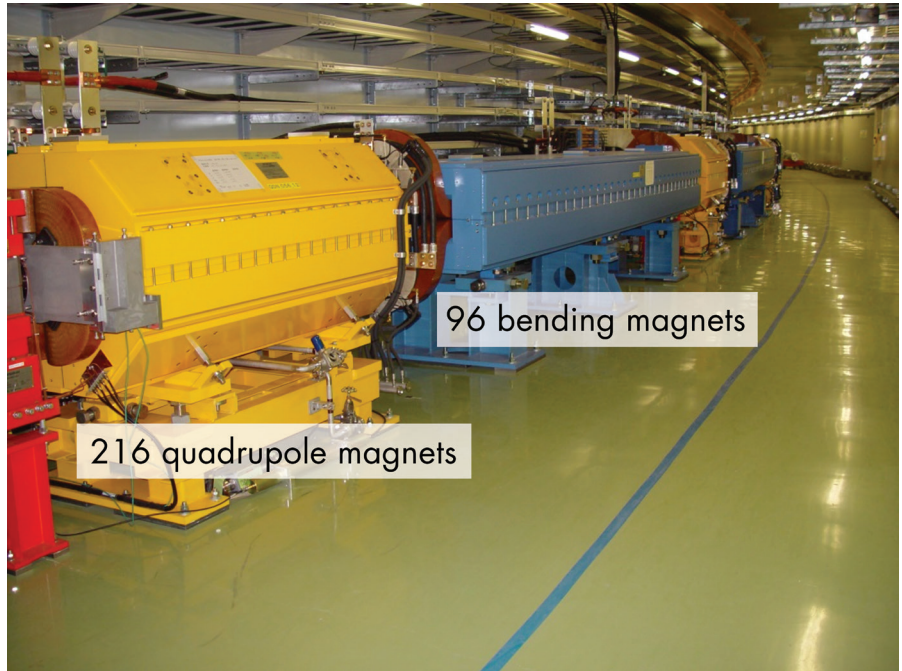


FIGURE 2. The J-PARC main ring, showing the bending and quadrupole electromagnets used to control the proton beam trajectory.

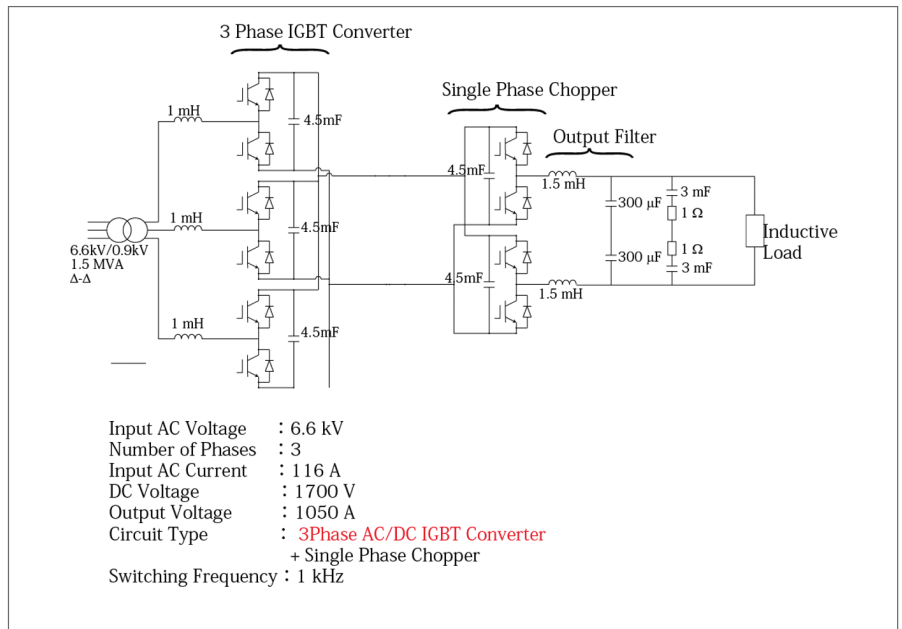


FIGURE 3. Schematic of the new electromagnet power supply unit.

chopper that supplies power from the capacitors to the electromagnet (Figure 3).

One of our goals in designing the power converter controller was to verify our design through simulation before performing tests on

actual hardware. We started by creating a plant model of the power supply's three-phase AC/DC converter and chopper using Simulink, Simscape™, and Simscape Electrical™. We then created a complete system model of the controller and plant (Figure 4).

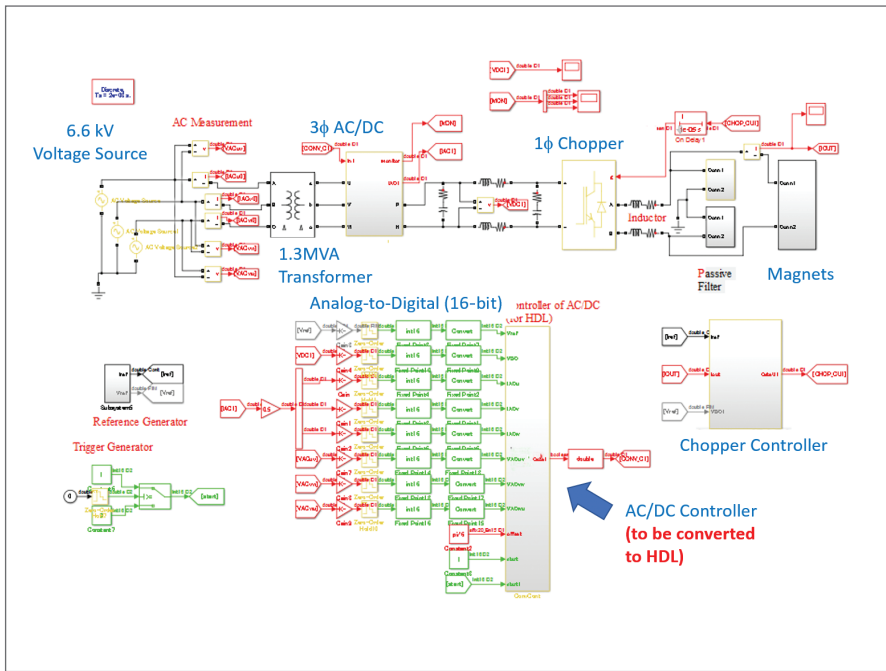


FIGURE 4. Simulink model of the power converter and its controller.

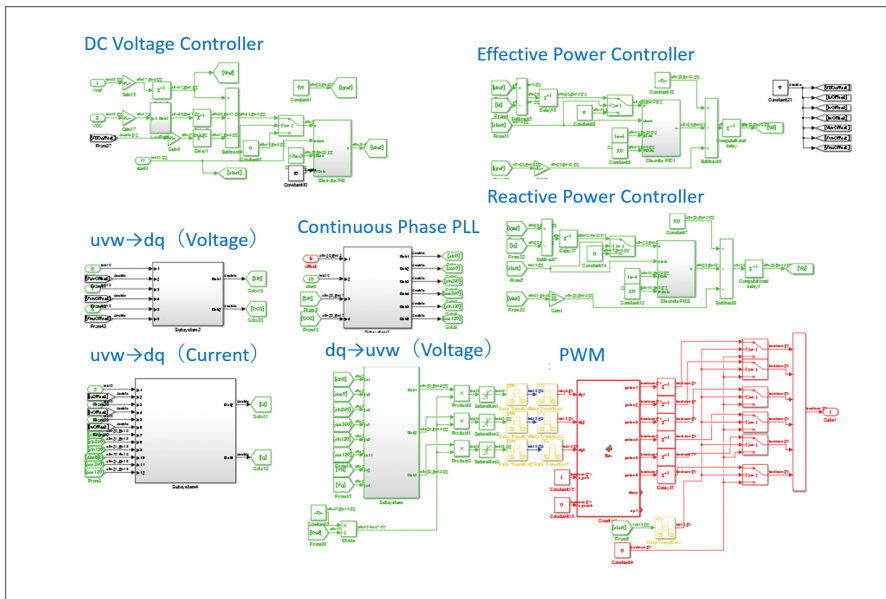


FIGURE 5. Simulink model of controller subsystems.

The controller model includes subsystems for DC voltage control, active power control, reactive power control, and pulse-width modulation, as well as elements for performing the direct-quadrature-zero transformations between three-phase signals and

the direct-quadrature (dq0) reference frame (Figure 5).

We selected an FPGA for the first version of our design because we needed to control multiple modules, and the input/output ca-

pabilities of the FPGA made it preferable to a microcontroller with relatively few inputs and outputs. One advantage of Model-Based Design is that, should we choose to redeploy on a microcontroller in the future, we will be able to generate C code from our existing controller design with Embedded Coder® and be up and running on a new target very quickly.

After running simulations to verify the design and tune control parameters, we generated synthesizable Verilog® code from our controller model using HDL Coder.

We deployed this code to a device from Intel's Cyclone® FPGA family and tested it using a smaller version of the production power supply. We verified that the waveforms from this setup matched the waveforms shown in the simulation results, with only minor deviations.

Finally, we tested and verified the FPGA controller on the actual power converter hardware.

We have completed the implementation of the first power converter unit equipped with our FPGA-based controller. We are currently building the remaining units needed for the entire main ring at J-PARC. We expect to begin neutrino oscillation experiments with this new setup when construction of these units is completed.

LEARN MORE

Power Converters
Modeling Techniques
mathworks.com/power-converter-modeling

Generating HDL Code for
FPGA and ASIC
mathworks.com/verifying-hdl-code

Beyond Image Classification: More Ways to Apply Deep Learning

By Johanna Pingel, MathWorks

Deep learning networks are proving to be versatile tools. Originally intended for image classification, they are increasingly being applied to a wide variety of other tasks, as well. They provide accuracy and processing speed—and they enable you to perform complex analyses of large data sets without being a domain expert. Here are some examples of tasks for which you might want to consider using a deep learning network.

Text Analytics

In this example, we'll analyze twitter data to see whether the sentiment surrounding a specific term or phrase is positive or negative. Sentiment analysis can have many practical applications, such as branding, political campaigning, and advertising.

Machine learning was (and still is) commonly used for sentiment analysis. A machine learning model can analyze individual words, but a deep learning network can be applied to complete sentences, greatly increasing its accuracy.

The training set consists of thousands of sample tweets categorized as either positive or negative. Here is a sample training tweet:

Tweet	Sentiment
"I LOVE @Health4UandPets u guys r the best!!"	Positive
"@nicolerichie: your picture is very sweet"	Positive
"Back to work!"	Negative
"Just had the worst presentation ever!"	Negative

We clean the data by removing "stop words" such as "the" and "and," which do not help the algorithm to learn. We then upload a *long short-term memory (LSTM)* network, a recurrent neural network (RNN) that can learn dependencies over time.

LSTMs are good for classifying sequence and time-series data. When analyzing text, an LSTM will take into account not only individual words but sentence structures and combinations of words, as well.

The MATLAB® code for the network itself is simple:

```
layers = [ sequenceInputLayer(inputSize)
           lstmLayer(outputSize, 'OutputMode', 'last')
           fullyConnectedLayer(numClasses)
           softmaxLayer
           classificationLayer ]
```

When run on a GPU, it trains very quickly, taking just 6 minutes for 30 epochs (complete passes through the data).

Once we've trained the model, it can be used on new data. For example, we could use it to determine whether there is a correlation between sentiment scores and stock prices.

Speech Recognition

In this example, we want to classify speech audio files into their corresponding classes of words. At first glance, this problem looks completely different from image classification, but it's actually very similar. A spectrogram is a 2D visualization of the signals in a 1D audio file (Figure 1). We can use it as input to a convolutional neural network (CNN) just as we would use a "real" image.

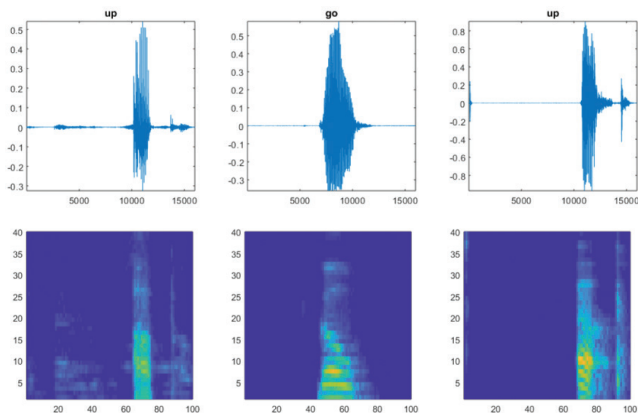


FIGURE 1. Original audio signals (top) with corresponding spectrograms.

The `spectrogram()` function is a simple way of converting an audio file into its corresponding time-localized frequency. However, speech is a specialized form of audio processing, with important features localized in specific frequencies. Because we want the CNN to focus on these locations, we will use Mel-frequency cepstral coefficients, which are designed to target the areas in frequency in which speech is most relevant.

We distribute the training data evenly between the classes of words we want to classify.

To reduce false positives, we include a category for words likely to be confused with the intended categories. For example, if the intended word is "on," then words like "mom," "dawn," and "won" are placed in the "unknown" category. The network does not need to know these words, just that they are *not* the words to recognize.

We then define a CNN. Because we are using the spectrogram as an input, the structure of our CNN can be similar to one we would use for images.

After the model has been trained, it will classify the input image (spectrogram) into the appropriate categories (Figure 2). The accuracy of the validation set is about 96%.

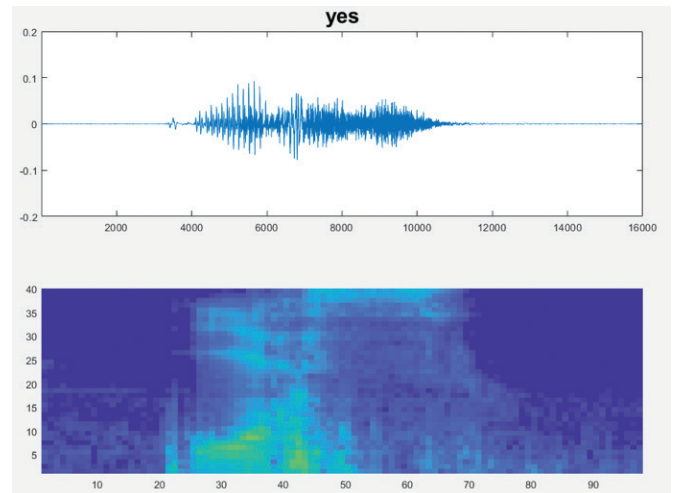


FIGURE 2. Classification result for the word "yes."

Image Denoising

Wavelets and filters were (and still are) common methods of denoising. In this example, we'll see how a pretrained image denoising CNN (DnCNN) can be applied to a set of images containing Gaussian noise (Figure 3).



FIGURE 3. Original image with Gaussian noise added.



FIGURE 4. Left: original (non-noisy) image. Right: denoised image.



FIGURE 5. Zoomed-in view.

We start by downloading an image that has Gaussian noise.

```
imshow(noisyRGB);
```

Since this is a color image, and the network was trained on grayscale images, the only semi-tricky part of this process is to separate the images into three separate channels: red (R), green (G), and blue (B).

```
noisyR = noisyRGB(:,:,1);
noisyG = noisyRGB(:,:,2);
noisyB = noisyRGB(:,:,3);
```

We load the pretrained DnCNN network.

```
net = denoisingNetwork('dncnn');
```

We can now use it to remove noise from each color channel.

```
denoisedR = denoiseImage(noisyR,net);
denoisedG = denoiseImage(noisyG,net);
denoisedB = denoiseImage(noisyB,net);
```

We recombine the denoised color channels to form the denoised RGB image.

```
denoisedRGB = cat(3,denoisedR,denoisedG,...
    denoisedB);
imshow(denoisedRGB)
title('Denoised Image')
```

A quick visual comparison of the original (non-noisy) image and the denoised image suggests that the result is reasonable (Figure 4).

Let's zoom in on a few details:

```
rect = [120 440 130 130];
cropped_orig = imcrop(RGB,rect);
cropped_denoise = imcrop(denoisedRGB,rect);
imshowpair(cropped_orig,cropped_denoise,...
    'montage');
```

The zoomed-in view in Figure 5 shows that the result of denoising has left a few side effects—clearly, there is more definition in the original (non-noisy) image, especially in the roof and the grass. This result might be acceptable, or the image might need further processing, depending on the application that it will be used for.

If you're considering using a DnCNN for image denoising, bear in mind that it can only recognize the type of noise on which it's been trained—in this case, Gaussian noise. For more flexibility, you can use MATLAB and Deep Learning Toolbox™ to train your own network using predefined layers or to train a fully custom denoising neural network.

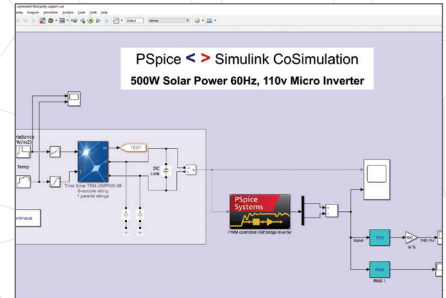
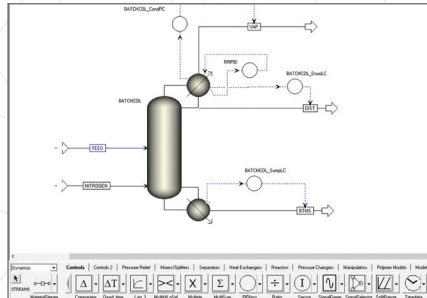
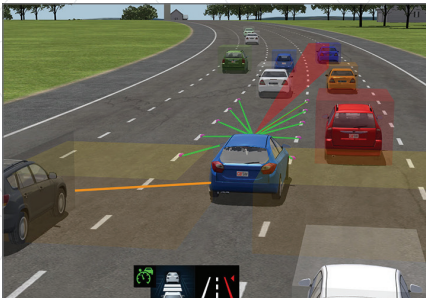
LEARN MORE

Classify Sequence Data Using LSTM Networks
mathworks.com/classify-sequence-data

Deep Learning Speech Recognition
mathworks.com/speech-recognition

Extending Simulink for Complex System Simulation and Integration

Simulink® integrates with third-party modeling tools through its open interfaces, enabling engineers to simulate heterogeneous, multi-domain systems at different fidelity levels. You can connect to over 100 modeling and simulation tools, serving applications such as electronic circuit board and motor design; mechanical and chemical modeling; and specialized vehicle design. Simulink provides the S-function API for efficient model and code integration and simulation and supports standards-based interfaces such as Functional Mock-Up Interface (FMI).



Mechanical Simulation: CarSim, TruckSim, BikeSim

The VehicleSim® products provide methods for simulating vehicle dynamics under a full range of test and driving conditions using SIL, HIL, and driving simulators. The products provide high-fidelity vehicle dynamics models, including braking, handling, ride, stability, and acceleration; portfolios of example vehicles and test maneuvers; and plotting and animation capabilities. Core vehicle models can be extended to work with Simulink models of advanced electronic controllers or with alternative component models. You can connect Simulink models to CarSim vehicle dynamics models through the S-function plug-in and then cosimulate and exchange input and output variables between models.

carsim.com

JSOL: JMAG

JMAG® finite element analysis software is used for developing electromechanical equipment such as motors, power converters, and actuators. JMAG can simulate magnetic flux density and electromagnetic forces in permanent magnet, induction, stepper, and a range of other motors. For motor control development, JMAG-RT extracts motor features as a precise reduced-order model provided as a Simulink block. High-fidelity JMAG-RT models capture device performance, including nonlinear effects, saturation, and space harmonics. By cosimulating control algorithms with accurate motor models, engineers can validate their control systems before hardware prototypes are available.

jmag-international.com

Aspen Technology: Aspen Plus Dynamics

Aspen Plus Dynamics is a dynamic simulation tool for improving plant operations and process design. It enables engineers to complete process control schemes, design verification, safety studies, relief value sizing, and failure analysis. It includes an extensive library of operation and control models with support for polymer processes and batch process optimization. Aspen Plus Dynamics includes a control design interface for extracting linear state-space models of nonlinear processes and importing them into MATLAB® for controller design. Using the Simulink interface you can connect process simulations as a block within a Simulink model. You can verify controller behavior by cosimulating Simulink controller models and nonlinear models of plant processes.

aspentech.com/en/products/engineering/Aspen-Plus-Dynamics

Cadence Design Systems: Cadence PSpice Systems Option

The integration of Cadence® PSpice® with Simulink provides a complete system-level simulation solution for PCB design and implementation. Designers can use PSpice for analog or mixed-signal simulation and perform Simulink based behavioral-level modeling, analysis, and visualization in a single system design and debug environment. The PSpice Systems Option enables cosimulation of SPICE-level electrical systems and Simulink based mechanical systems for application areas including automotive systems, internet of things (IoT), and industrial design.

orcad.com/pspice-and-simulink-integration

LEARN MORE

System Modeling and Simulation
mathworks.com/system-design-simulation

Third-Party Products and Services
mathworks.com/connections

MATLAB AND SIMULINK IN THE WORLD

TRANSFORMATIVE TECHNOLOGY

Restoring sight and voice, connecting cities with a hyperloop transportation system, generating electricity from thin air—engineers at emerging companies are using MATLAB® and Simulink® to develop pioneering solutions and break boundaries.

LEARN MORE

- User Stories mathworks.com/user-stories
- MATLAB and Simulink for Startups mathworks.com/startups

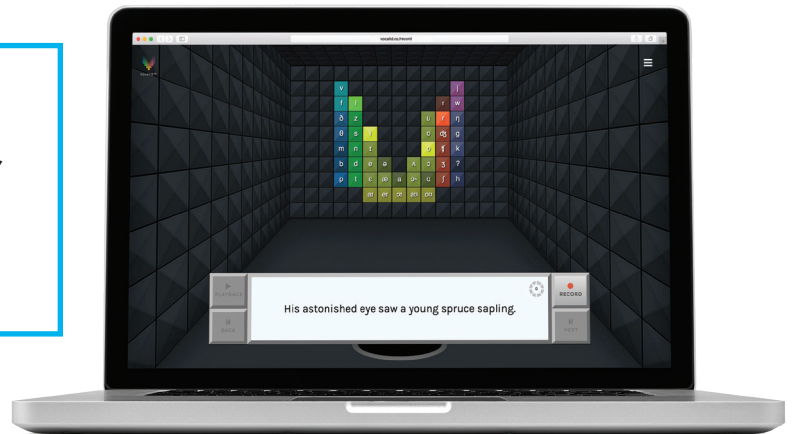


THE WORLD'S FIRST EYE SURGERY ROBOT

Surgery performed inside the eye demands almost superhuman precision and stability. A surgeon at John Radcliffe Hospital, Oxford, removed a retinal membrane one hundredth of a millimeter thick using the PRECEYES Surgical System, an inverted joystick-based device that automatically moves the tool tip in response to the surgeon's movements.

"We wouldn't dream of fitting a little girl with the prosthetic limb of a grown man—so why, then, the same prosthetic voice?"

— Rupal Patel, VocaliD



CROWDSOURCING UNIQUE DIGITAL VOICES

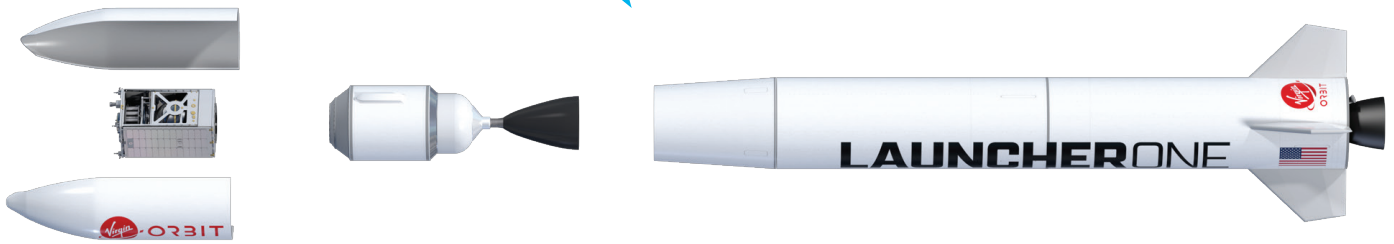
VocaliD is developing the first-ever personalized digital voices, enabling people who rely on synthetic speech for communication to sound like themselves. A personalized voice is a blend of the recipient's vocalization and recordings of a matched speaker from VocaliD's Human Voicebank, a repository of 26,000 contributors worldwide. The resulting BeSpoke™ voice can be downloaded for use on text-to-speech devices and applications across all platforms.

"There are a lot more runways in the world than there are launch pads. Our downrange can be anywhere we point the aircraft."

— Patrick Harvey, Virgin Orbit

LAUNCHING SATELLITES AT 35,000 FEET

LauncherOne is Virgin Orbit's two-stage launch vehicle for delivering small satellites into low earth orbit. To reduce costs and increase launch location flexibility, LauncherOne is designed to be air-dropped from a 747-400 carrier aircraft in flight.



"Imagine a world where distance just doesn't matter anymore—a system that's faster than airplanes, more convenient than trains, and the best alternative for the environment. That's the world we're making into a reality, and it's called the hyperloop."

— Mars Geuze, Hardt

FROM CITY TO CITY AT THE SPEED OF SOUND

Hardt Hyperloop is developing the first high-speed hyperloop test facility in the world in the Dutch province of Flevoland. The hyperloop consists of small, light-weight vehicles travelling through a tube with virtually no air resistance, allowing them to travel over huge distances very fast, with minimal energy consumption.

"Mobile high-altitude wind energy generators can provide cheap renewable energy to everyone who needs it."

— Mario Milanese, Kiteenergy

HARNESSING HIGH-ALTITUDE WIND ENERGY

Kiteenergy converts high-altitude wind energy into electricity by exploiting the flight of automatically controlled kites tethered 200–800 meters above the ground. Electricity is generated at ground level by converting the traction forces acting on the tethers into mechanical and electrical power, using rotating mechanisms and electrical generators.





IN THE FRONT SEAT

Putting Student Engineers in the Front Seat with a Driver-in-the-Loop Automotive Simulator

By Håkan Richardson and Mikael Enelund, Chalmers University of Technology

The Conceive, Design, Implement, Operate (CDIO) approach to engineering education that we use at Chalmers was designed to address the industry demand for graduates who are both well-grounded in engineering principles and equipped to tackle real-world engineering problems.

Hands-on projects with MATLAB® and Simulink® are fundamental to our CDIO-based curriculum because they let students see the effects of their design decisions firsthand.

Two years ago, we saw an opportunity to enable students not just to see the results of their work but to *feel* them, as well. We installed a driver-in-the-loop automotive simulator that moves a student through six degrees-of-freedom as they drive. The simulator is maintained, operated, and continuously improved by Caster, a Chalmers student organization that also helped secure the funds to purchase it. The simulator includes MATLAB and Simulink vehicle models. It has an interface that enables students to incorporate their own models and feel how their designs would perform on a real vehicle.

The mechanical engineering program has integrated the Caster simulator into a first-year undergraduate programming course and a masters-level course in vehicle dynamics. More importantly, the facility that houses the simulator has emerged as a gathering place for students interested in learning more about automotive engineering outside a specific course (Figure 1).



FIGURE 1. The Caster lounge, which houses the simulator as well as workstations and places for students to socialize.



FIGURE 2. A student experiencing the effects of his MATLAB code in a drag race in the Caster simulator.



FIGURE 3. A graduate student running a vehicle dynamics model simulation.

Teaching First-Year Programming With the Simulator

MATLAB is an essential component of the engineering curriculum at Chalmers, and we require all undergraduates to take *Programming with MATLAB* in their first year.

To provide students with a positive introduction to programming, we incorporated the Caster simulator into the first course assignment. The students were asked to write an algorithm in MATLAB that set the appropriate gear based on parameters such as engine speed (in rpm) and vehicle speed. During

simulations, the Caster simulator's MATLAB model transmitted these parameters in real time as a stream of telemetry data.

The students' algorithms were tested in a drag race with a simulated Camaro SS. The students with the fastest times in an initial round of desktop simulations advanced to the finals, in which they sat in the Caster simulator as the simulated Camaro raced down the drag strip (Figure 2).

Developing a shifting algorithm from scratch is daunting for students new to programming, so we provided a rudimentary algorithm that simply shifted from first to fourth gear when the vehicle reached 20 km/h. We also gave them the vehicle's engine torque curve, gear ratios, and tire slip ratio, and delivered a short lecture on how engineers use this data to develop optimal shifting algorithms.

We expected most students to develop a straightforward implementation that simply shifted gears progressively as the car reached certain speeds. We thought maybe a handful would use the additional torque and gear ratio information in a more sophisticated algorithm. We were astonished when dozens of students began showing up at the Caster facility after school hours to ask us how they could improve their MATLAB algorithms and develop more advanced solutions. The enthusiasm generated by this project carried through the remainder of the course.

Teaching Masters-Level Vehicle Dynamics

Use of the Caster simulator in coursework is steadily expanding. At the undergraduate level, we plan to employ the simulator in a second-year machine design course in which mechanical engineering students will design a vehicle braking system in MATLAB and then run simulations to evaluate their systems' performance.

At the graduate level, we have already incorporated the simulator into a course on

MATLAB and Simulink at Chalmers

The CDIO approach to engineering education relies heavily on modeling and simulation, and at Chalmers we use MATLAB and Simulink for these activities. As a result, for our engineering students, MATLAB is as widely used as pen and paper. MATLAB and Simulink are integrated deeply into the mathematics and engineering curriculum at Chalmers. This integration has been enabled by a Total Academic Headcount license, which provides Chalmers students with campus-wide access to the tools.

Our adoption of the CDIO approach with MATLAB and Simulink has led to improved learning outcomes and contributed to a wider recognition of Chalmers as a top-ranked technical university. Our mechanical engineering program was awarded Centre of Excellence status by the National Swedish Agency for Higher Education and Best Engineering Education by the Swedish engineering employers' organization. After a recent evaluation of higher education, the Swedish government awarded Chalmers' Mechanical Engineering program the highest distinction, an honor that came with additional financial support that helped fund the Caster program.

vehicle dynamics. In this course, students develop their own vehicle models. Working in Simulink with a framework model we provide, the students add the necessary equations of motion to accurately capture vehicle dynamics (Figure 3).

After running offline simulations in Simulink, the students plug their models into the Caster simulator and evaluate their performance on a test track with a skid pad and an acceleration straight. We then ask the students to model three different weight distributions between the front and rear axles. They sit in the Caster simulator so that they can feel how the car reacts to each weight distribution.

Next, they run a similar set of simulations while modifying the brake balance between the front and rear axles. In the simulator they experience the effects of front brake and rear brake locking. For an automotive engineer, experiencing the motion in the simulator firsthand—rather than just seeing data plotted in a graph—provides a vivid memory and a much deeper understanding of the effects of design parameters on steering, braking, and overall vehicle dynamics.

Like their undergraduate counterparts in the introductory programming course, the graduate students were enthusiastic about their experience with the Caster simulator. In fact,

after our first use of the simulator in the vehicle dynamics course, student enrollment in the follow-on course, *Advanced Vehicle Dynamics*, doubled.

Putting Engineers in the Front Seat

Caster's motto, "Engineers in the Front Seat," in many ways reflects the culture of continuous improvement at Chalmers. We want to remain at the forefront of engineering education, and are always looking for ways to improve our programs with new courses and assignments.

Caster simulator technology has already played a significant role in enhancing our curriculum, but the Caster student organization has had an even more dramatic effect. We have hosted visitors from about 20 different universities who have expressed interest in setting up a similar program.

Caster has also helped Chalmers' recruitment efforts. When secondary school students visit our campus, we show them the simulator. We've seen interest among secondary school visitors; even if they are not particularly attracted to automotive technology, they still find the coding, virtual reality, and product development aspects of the program appealing.

Lastly, we have seen a very positive response from industry, including leading automotive manufacturers in Sweden. Many company representatives have spent time in the Caster lounge area talking with students about their work with the simulator. We are seeing tremendous interest in the engineers that Chalmers is producing because our graduates have not only the skills companies are looking for but a deeper understanding of engineering principles founded in their experience with CDIO principles and hands-on technology, including the Caster simulator. Some of the students involved with Caster have already gone on to work for automotive companies, and one recently started working as a game developer for a company that produces racing video games.

LEARN MORE

The Caster Simulator (0:30)
youtube.com/watch?v=rYuMF_aZhq8

Modeling a Vehicle Dynamics System
mathworks.com/vehicle-dynamics-example

Chalmers University of Technology Integrates MATLAB Throughout Core Mathematics Curriculum
mathworks.com/chalmers

Three Ways to Estimate Remaining Useful Life for Predictive Maintenance

By Aditya Baru, MathWorks

Remaining useful life (RUL) is the length of time a machine is likely to operate before it requires repair or replacement. By taking RUL into account, engineers can schedule maintenance, optimize operating efficiency, and avoid unplanned downtime. For this reason, estimating RUL is a top priority in predictive maintenance programs.

An RUL estimation model not only predicts RUL but also provides a confidence bound on the prediction. The model inputs are *condition indicators*, features extracted from sensor data or log data whose behavior changes in a predictable way as the system degrades or operates in different modes.

The method used to calculate RUL depends on the kind of data available:

- Lifetime data indicating how long it took for similar machines to reach failure
- Run-to-failure histories of machines similar to the one you want to diagnose
- A known threshold value of a condition indicator that detects failure

Predictive Maintenance Toolbox™ provides models for estimating RUL from each type of data.

Lifetime Data

Proportional hazard models and probability distributions of component failure times are used to estimate RUL from lifetime data. A simple example is estimating the discharge time of a battery based on past discharge times and *covariates*, variables such as the environment in which the battery operated (such as temperature) and the load placed on it.

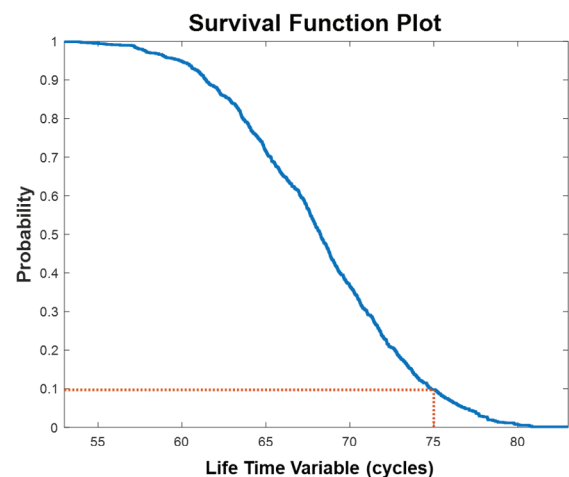


FIGURE 1. Survival function plot. At the end of 75 cycles, the probability of a battery's continuing to operate is 0.1, or 10%.

The survival function plot in Figure 1 shows the probability that a battery will fail based on how long it has been in operation. The plot shows, for example, that if the battery is in operation for 75 cycles, it has a 90% chance of being at the end of its life time.

Run-to-Failure Data

If you have run-to-failure data from similar components or different components showing similar behavior, you can estimate RUL using similarity methods. These methods capture *degradation profiles* and compare them with new data coming in from the machine to determine which profile the data matches most closely.

In Figure 2, the degradation profiles of historical run-to-failure data sets from an engine are shown in blue and the current data from the engine is shown in red. Based on the profile the engine most closely matches, the RUL is estimated to be around 65 cycles.

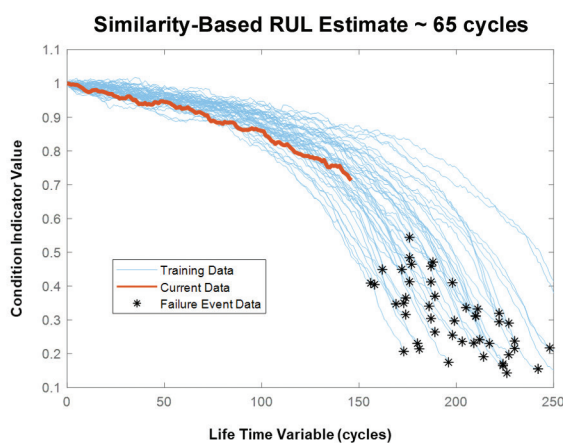


FIGURE 2. Degradation profiles (blue) based on run-to-failure data. The distribution of the stars (or endpoints) of the nearest blue curves gives an RUL of 65 cycles.

Threshold Data

In many cases, run-to-failure data or lifetime data was not recorded but you do have information on prescribed threshold values—for example, the temperature of a liquid in a pump cannot exceed 160°F (71°C) and the pressure must be under 2200 psi (155 bar). With this kind of information, you can fit time series models to condition indicators extracted from sensor data such as temperature and pressure, which rise or fall over time.

These *degradation models* estimate RUL by predicting when the condition indicator will cross the threshold. They can also be used with a *fused condition indicator* that incorporates information from more than one condition indicator using techniques such as principal component analysis.

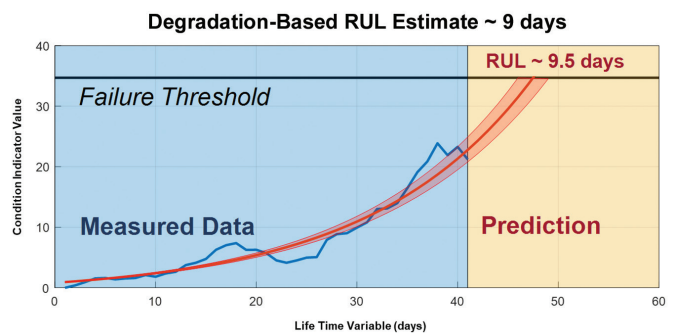


FIGURE 3. Degradation model for a high-speed bearing. The bearing has an estimated RUL of 9.5 days based on its current condition data (blue) and the exponential degradation model (red) fit to this data.

Figure 3 shows an exponential degradation model that tracks failure in a high-speed bearing used in a wind turbine. The condition indicator is shown in blue. The degradation model predicts that the bearing will cross the threshold value in approximately 9.5 days. The region shaded in red represents the confidence bounds for this prediction.

Once you have reliable estimates for RUL, you can integrate them into dashboards used by operators or incorporate them into alarm systems monitored by maintenance teams. Teams can then respond to changes in equipment health as quickly as possible, and without affecting operations.

LEARN MORE

Estimating RUL Using Run-to-Failure Data from an Engine
mathworks.com/similarity-based-example

Estimating RUL of a Battery Using Physical Modeling and Kalman Filters
mathworks.com/degrading-battery-example

Estimating RUL of High-Speed Bearings Using Exponential Degradation Models
mathworks.com/bearing-example

A Brief History of MATLAB

By Cleve Moler, MathWorks

The first MATLAB® was not a programming language; it was a simple interactive matrix calculator. There were no programs, no toolboxes, no graphics. And no ODEs or FFTs. In this Cleve's Corner, I'll describe some milestones in the evolution of MATLAB from those simple beginnings.

Mathematical Origins

The mathematical basis for the first version of MATLAB was a series of research papers by J. H. Wilkinson and 18 of his colleagues, published between 1965 and 1970 and later collected in *Handbook for Automatic Computation, Volume II, Linear Algebra*, edited by Wilkinson and C. Reinsch. These papers present algorithms, implemented in Algol 60, for solving matrix linear equation and eigenvalue problems.

EISPACK and LINPACK

In 1970, a group of researchers at Argonne National Laboratory proposed to the U.S. National Science Foundation (NSF) to “explore the methodology, costs, and resources required to produce, test, and disseminate high-quality mathematical software and to test, certify, disseminate, and support packages of mathematical software in certain problem areas.” The group developed EISPACK (Matrix Eigensystem Package) by translating the Algol procedures for eigenvalue problems in the handbook into Fortran and working extensively on testing and portability. The first version of EISPACK was released in 1971 and the second in 1976.

In 1975, four of us—Jack Dongarra, Pete Stewart, Jim Bunch, and myself—proposed to the NSF another research project that would investigate methods for the development of mathematical software. A byproduct would be the software itself, dubbed LINPACK, for Linear Equation Package. This project was also centered at Argonne.

LINPACK originated in Fortran; it did not involve translation from Algol. The package contained 44 subroutines in each of four numeric precisions.

In a sense, the LINPACK and EISPACK projects were failures. We had proposed research projects to the NSF to “explore the methodology, costs, and resources required to produce, test, and disseminate high-quality mathematical software.” We never wrote a report or paper addressing those objectives. We only produced software.

Historic MATLAB

In the 1970s and early 1980s, I was teaching Linear Algebra and Numerical Analysis at the University of New Mexico and wanted my students to have easy access to LINPACK and EISPACK without writing Fortran programs. By “easy access,” I meant not going through the remote batch processing and the repeated edit-compile-link-load-execute process that was ordinarily required on the campus central mainframe computer.

So, I studied Niklaus Wirth's book *Algorithms + Data Structures = Programs* and learned how to parse programming languages. I wrote the first MATLAB—an acronym for Matrix Laboratory—in Fortran, with matrix as the only data type. The project was a kind of hobby, a new aspect of programming for me to learn and something for my students to use. There was never any formal outside support, and certainly no business plan.

This first MATLAB was just an interactive matrix calculator. This snapshot of the start-up screen shows all the reserved words and functions. There are only 71. To add another function, you had to get the source code from me, write a Fortran subroutine, add your function name to the parse table, and recompile MATLAB.

```

< M A T L A B >
Version of 05/12/1981
<>

The functions and commands are...
ABS  ATAN  BASE  CHAR  CHOL  CHOP  COND  CONJ
COS  DET   DIAG  DIAR  DISP  EIG   EPS   EXEC
EXP  EYE   FLOP  HESS  HILB  IMAG  INV   KRON
LINE LOAD  LOG   LU    MAGI  NORM  ONES  ORTH
PINV PLOT  POLY  PRIN  PROD  QR    RAND  RANK
RAT  RCON  REAL  ROOT  ROUN  RREF  SAVE  SCHU
SIN  SIZE  SQRT  SUM   SVD   TRIL  TRIU  USER
CLEA ELSE  END   EXIT  FOR   HELP  IF    LONG
RETN SEMI  SHOR  WHAT  WHIL  WHO   WHY
    
```


1984

PC-MATLAB
FFT
Graphics

1985

Pro-MATLAB
Control System Toolbox™

1987

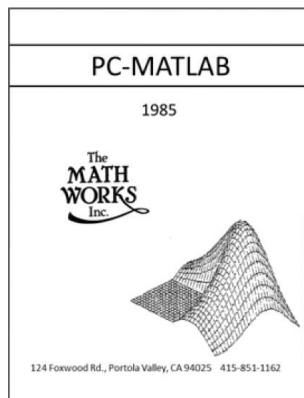
Signal Processing Toolbox™
ODEs

1992

Sparse matrices
Simulink™

Commercial MATLAB

I spent the 1979–80 academic year at Stanford, where I taught the graduate course in Numerical Analysis and introduced the class to this matrix calculator. Some of the students were studying subjects like control theory and signal processing, which I knew nothing about. Matrices were central to the mathematics in these subjects, though, and MATLAB was immediately useful to the students.



Jack Little had been in the graduate engineering program at Stanford. A friend of his who took my course showed him MATLAB, and he adopted it for his own work.

In 1983, Little suggested the creation of a commercial product based on MATLAB. The IBM® PC had been introduced only two years earlier. It was barely powerful enough to run a program like MATLAB, but Little anticipated its evolution. He left his

job, bought a Compaq® PC clone at Sears, moved into the hills behind Stanford, and with my encouragement, wrote a new and extended version of MATLAB in C. A friend, Steve Bangert, worked on the new MATLAB in his spare time.

PC-MATLAB made its debut in December 1984 at the IEEE Conference on Decision and Control in Las Vegas. Pro-MATLAB, for Unix workstations, followed a year later.

Little and Bangert made many important modifications and improvements to Historic MATLAB when they created the new and extended version. The most significant were functions, toolboxes, and graphics.

Modern MATLAB

While preserving its roots in matrix mathematics, MATLAB has continued to evolve to meet the changing needs of engineers and scientists. The key developments are shown in the timeline. Here, I'll elaborate on some of them.

ODEs

The numerical solution of ordinary differential equations has been a vital part of MATLAB since its commercial beginning. ODEs are also the core of Simulink®, the MATLAB companion product for simulation and Model-Based Design.

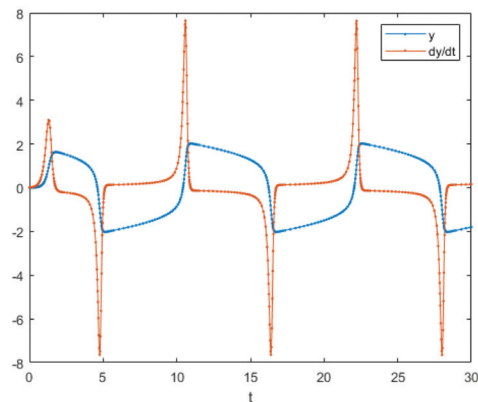
The Van der Pol oscillator is a classical ODE example.

$$\ddot{y} = \mu(1 - y^2)\dot{y} - y$$

The parameter μ is the strength of the nonlinear damping term. When $\mu = 0$, we have the basic harmonic oscillator.

The MATLAB code expresses the oscillator as a pair of first-order equations.

```
mu = 5;
vdp = @(t,y) [y(2); mu*(1-y(1)^2)*y(2)-y(1)];
tspan = [0 30];
y0 = [0 0.01]';
[t,y] = ode23s(vdp,tspan,y0);
plot(t,y,'.-')
legend({'y','dy/dt'})
xlabel('t')
```



1993

Image Processing Toolbox™
Symbolic Math Toolbox™

1996

Single precision
Cell arrays
Structures

1999

Objects

2000

Desktop
LAPACK

2004

Integer data
Function handles
Parallel computing

The Van der Pol oscillator, with the parameter μ set to 5, is a mildly *stiff* differential equation. In anticipation, I used the `ode23s` solver; the 's' in the name indicates that it is for stiff equations. In the plot you can see some clustering of steps where the solution is varying rapidly. A nonstiff solver would have taken many more steps. A stiff ode solver uses an *implicit* method requiring the solution of a set of simultaneous linear equations at each step. The iconic MATLAB backslash operator is quietly at work here.

Data Types

For many years, MATLAB had only one numeric data type: IEEE standard 754 double-precision floating point, stored in the 64-bit format. As people began to use MATLAB for more applications and larger data sets, we provided more ways to represent data.

Single Precision and Integer

Support for single-precision arithmetic began in the early 2000s and was complete by MATLAB 7 in 2004. Requiring only 32 bits of storage, single precision cuts memory requirements for large arrays in half. MATLAB does not have declarations, so single-precision variables are obtained by executable conversion functions.

MATLAB 7 also introduced three unsigned integer data types, `uint8`, `uint16`, and `uint32`; three signed integer data types, `int8`, `int16`, and `int32`; and one logical data type, `logical`.

Sparse Matrices

Sparse matrices were introduced with MATLAB 4 in 1992. They are a memory-efficient way to represent very large arrays that have few nonzero values. Only the nonzero elements are stored, along with row indices and pointers to the starts of columns. The only change to the outward appearance of MATLAB is a pair of functions, `sparse` and `full`. Nearly all the operations apply equally to full and sparse matrices. The sparse storage scheme represents a matrix in space proportional to the number of nonzero entries, and most of the operations compute sparse results in time proportional to the number of arithmetic operations on nonzeros.

Cell Arrays

Cell arrays were introduced with MATLAB 5 in 1996. A cell array is an indexed, possibly heterogeneous collection of MATLAB objects, including other cell arrays. Cell arrays are created by curly braces, `{ }`.

Cell arrays can be indexed by both curly braces and smooth parentheses. With braces, `c{k}` is the contents of the *k*-th cell. With parentheses, `c(k)` is another cell array containing the specified cells. Think of a

collection of mailboxes. `box(k)` is the *k*-th mailbox. `box{k}` is the mail in the *k*-th box.

Structures

Structures and associated "dot notation" were introduced in 1996. This script for creating a grade book for a small class shows structures and dot notation at work.

```
Math101.name = ["Alice Jones"; ...
               "Bob Smith"; "Charlie Brown"];
Math101.grade = ["A"; "B+"; "C"];
Math101.year = [4; 2; 3];
```

To call the roll, we need the list of names.

```
disp(Math101.name)
    "Alice Jones"
    "Bob Smith"
    "Charlie Brown"
```

Changing Charlie's grade involves both structure and array notation.

```
Math101.grade(3) = "W";
disp(Math101.grade)
    "A"
    "B+"
    "W"
```

Objects

Major enhancements to MATLAB object-oriented programming capabilities were made in 2008. Creating classes can simplify programming tasks that involve specialized data structures or large numbers of functions that interact with particular kinds of data. MATLAB classes support function and operator overloading, controlled access to properties and methods, reference and value semantics, and events and listeners.

The MATLAB graphics system is one large, complex example of the object-oriented approach to MATLAB programming.

Making MATLAB More Accessible: Desktop and Live Editor

The first versions of MATLAB were simple terminal applications. Over time we added separate windows for graphics, editing, and other tools. These gradually made MATLAB easier to use, especially for users without prior programming experience. Two specific features that have had the biggest impact are the desktop and the Live Editor.

2008

Objects improved

2010

GPUs

2014

New graphics system

2016

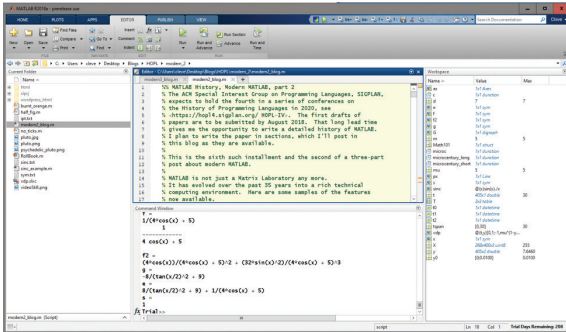
Live Editor

2017

Tall arrays and categorical arrays

Desktop

The MATLAB desktop was introduced in 2000. Here is a screenshot showing how it looks today.



Four panels are visible: the current folder viewer (left), the workspace viewer (right), the editor/debugger (top center), and the traditional command window (bottom center). A file viewer and a command history window can also be included in personalized layouts.

Any panel can be closed or undocked into a standalone window.

Live Editor

The Live Editor was introduced in 2016 and is still evolving rapidly. Descriptive text and MATLAB input, output, and graphics are combined in a single interactive document that can be exported to HTML, PDF or LaTeX.

Symbolic math is typeset.

```
x = sym('x');
f = 1/(4*cos(x)+5)
```

$$f = \frac{1}{4 \cos(x) + 5}$$

Differentiate twice, then integrate twice.

```
f2 = diff(f,2)
```

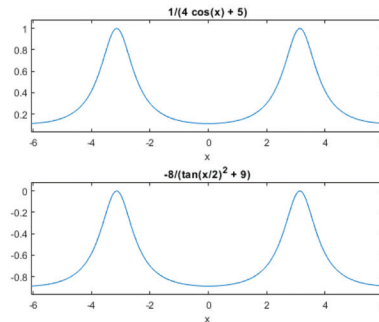
$$f2 = \frac{4 \cos(x)}{(4 \cos(x) + 5)^2} + \frac{32 \sin(x)^2}{(4 \cos(x) + 5)^3}$$

```
g = int(int(f2))
```

$$g = -\frac{8}{\tan\left(\frac{x}{2}\right) + 9}$$

Graphics merge with code and output.

```
subplot(2,1,1)
ezplot(f)
subplot(2,1,2)
ezplot(g)
```



Parallel Computing

Parallel Computing Toolbox™ was introduced at the SuperComputing conference in 2004. The following year, at SC05, Bill Gates gave the keynote talk, using MATLAB to demonstrate Microsoft's entry into high-performance computing.

The toolbox supports coarse-grained, distributed memory parallelism by running many MATLAB workers on several machines in a cluster or on many cores in a single machine. MPI is used for the underlying message passing. By far the most popular feature of the toolbox is the parallel **for** loop command, **parfor**.

The toolbox also supports fine-grained, shared memory parallelism in attached graphics processing units (GPUs). Here, the **gpuArray** array gets things started.

Toolboxes

Much of the power of modern MATLAB derives from the toolboxes available for specialized applications. As of release 2018a, there are 63 of them.

What's Next?

MATLAB has come a long way since the simple calculator that started it all. It is a living ecosystem supporting all aspects of technical computing. We will continue to strengthen existing features as we carefully add new ones. Our goals are always ease of use, power, and speed.

LEARN MORE

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

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

Run MATLAB Image Processing Algorithms on Raspberry Pi and NVIDIA Jetson

By Jim Brock and Murat Belge, MathWorks

Thanks to low-cost hardware platforms such as Raspberry Pi™, it is now easier than ever to prototype image processing algorithms on hardware. Most image processing algorithms are computationally intensive, and it can be challenging to run them on an embedded platform with acceptable frame rates. While Raspberry Pi is sufficient for running simple image processing algorithms, large images and complex algorithms are best run on more powerful hardware such as NVIDIA® Jetson.

Using a chroma key effect as an example, this article describes a simple workflow for deploying a MATLAB® image processing algorithm to embedded hardware. We'll generate C code from the algorithm with MATLAB Coder™, and then use the Run on Hardware utility to prototype the algorithm on a Raspberry Pi board. Finally, we'll move the algorithm to an NVIDIA Jetson Tx1 platform to achieve real-time performance.

The Chroma Keying Algorithm

Widely used in TV weather reports, movie production, and photo editing applications, chroma keying is a video processing technique in which a foreground subject is shot against a solid color background, such as a green screen, that is later replaced by a different scene (Figure 1).

The chroma keying algorithm compares each pixel in the image with a reference color representing the solid background color. If the color of the pixel is close enough to the reference color, the pixel is replaced with the corresponding pixel from a pre-selected scene image. Mathematically, the chroma keying algorithm can be formulated as:

$$P_{final}(j, k) = m(j, k) * P_{original}(j, k) + (1 - m(j, k)) * P_{scene}(j, k)$$

Where $P_{final}(j, k)$ represents the final pixel value at location (j, k) after chroma keying, $P_{original}(j, k)$ is the pixel value corresponding to the original image, $P_{scene}(j, k)$ is the pixel value representing the scene that replaces the solid background color, and $m(j, k) \in [0, 1]$ is a mask value. The mask value $m(j, k)$ should be 1 for foreground pixels and 0 for background pixels. A mask value between 0 and 1 provides a smooth transition from background to foreground.

The mask value at each pixel is usually computed in the YCbCr color space instead of the usual RGB color space. The Y component of the YCbCr image represents the luminance component and determines how light or dark the image is. Cb and Cr components represent the chroma components that can be used to measure similarity to a reference color. Measuring color similarity using only the Cb and Cr components of the image makes the algorithm robust to variations in luminance values in light and dark areas of the solid background color.

To measure the similarity of a pixel color to a reference color, we use the squared Euclidian distance in chroma space:

$$d^2(j, k) = (Cb(j, k) - Cb_{ref}(j, k))^2 + (Cr(j, k) - Cr_{ref}(j, k))^2$$

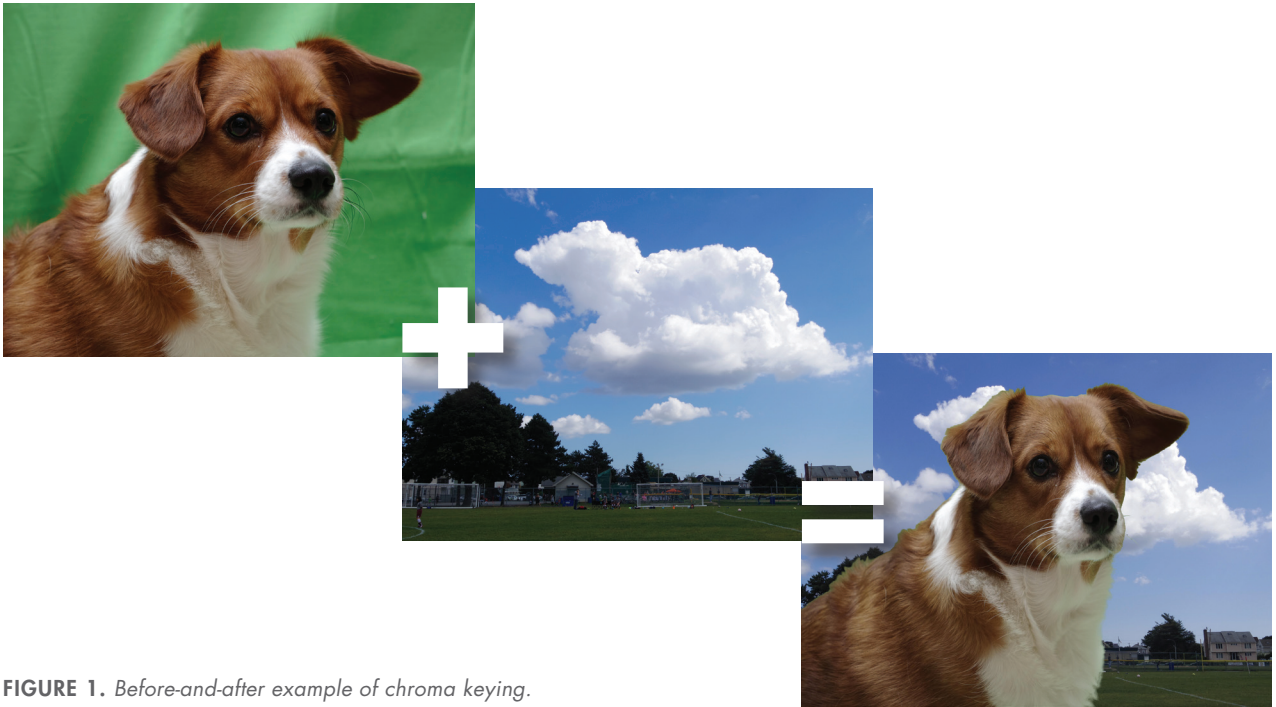


FIGURE 1. Before-and-after example of chroma keying.

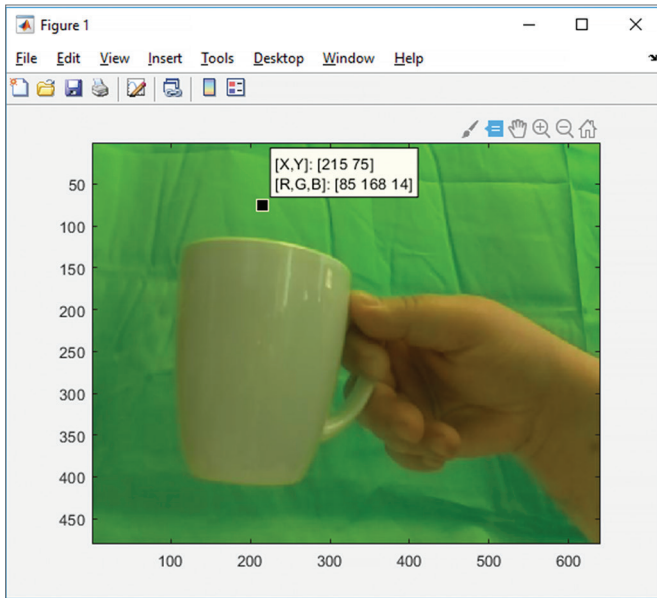


FIGURE 2. The MATLAB Data Cursor tool, used to determine background color values.

Finally, we compute the mask value at location (j,k) in the image using the following formula:

$$m(j, k) = \begin{cases} 1 & \text{if } d(j, k) > t_2 \\ 0 & \text{if } d(j, k) < t_1 \\ \frac{d^2(j,k) - t_1^2}{t_2^2 - t_1^2} & \text{if } t_1 < d(j, k) < t_2 \end{cases}$$

Where t_1 and t_2 with $t_2 > t_1$ represent threshold values to be determined.

MATLAB Implementation

Code excerpts begin on page 39.

Code Excerpt 1 shows the MATLAB implementation of the chroma keying algorithm.

In MATLAB, images are represented as $[N, M, 3]$ arrays of type `uint8`. This means that we'll need to convert the image data type to 'double' before performing mathematical operations. To avoid abrupt transitions from background to foreground, we apply a Gaussian filter to the computed mask.

Determining Reference Color and Thresholds

A chroma keying algorithm requires a reference color and thresholds. Using the camera interface in MATLAB Support Package for Raspberry Pi, we capture images of the actual scene. We can then empirically determine the approximate reference color for the background and the approximate threshold values (Code Excerpt 2).

The `img = snapshot(cam)` command plots the image captured from Raspberry Pi camera in MATLAB. We use the Data Cursor tool in the MATLAB plot to specify the background color (Figure 2).

To determine the thresholds, we run the algorithm in a loop and adjust the threshold values (Code Excerpt 3).

When we run the code we get an image shown against the background we selected (Figure 3).



FIGURE 3. Top: original image. Bottom: image obtained after running the chroma keying algorithm.

Deploying the Chroma Keying Algorithm to Raspberry Pi

Before deploying the code, we need to write a loop around the chroma keying algorithm to capture images from a camera and display them on a monitor attached to Raspberry Pi (Code Excerpt 4).

`matlab.raspi.webcam` and `matlab.raspi.SDLVideoDisplay` are System objects™ in the Run on Hardware utility that facilitate use of camera and Raspberry Pi display in a deployment workflow. To compile and run the code, we execute the command shown in Code Excerpt 5.

The function `runOnHardware` creates a MATLAB Coder configuration for Raspberry Pi hardware, generates code for the `chromaKeyApp.m` script, and deploys it. In order to run the algorithm at a reasonable frame rate, the image size can be reduced to 640x480 or 320x240.

Generating GPU Code

The algorithm is working on the Raspberry Pi, but it is not achieving the real-time performance we're looking for. To accelerate the algorithm, we will use GPU Coder™ to deploy it to the NVIDIA Jetson platform. We need to generate GPU code to take advantage of

the inherent parallelism in the algorithm. First, we write a wrapper `main` function that uses OpenCV to access a USB camera connected to the NVIDIA Jetson. This function will marshal video frames from the camera to our `chromaKey` algorithm and then display the output on the screen.

When generating GPU code, we first create a GPU Coder configuration object, set the GPU parameters to target the NVIDIA Jetson platform, and include our custom `main` function. We will not compile the code on the MATLAB host computer, because we are generating code specifically for the NVIDIA Jetson board. We will create a script to set up the GPU Coder configuration, input example data, and generate source code for our application (Code Excerpt 6).

We then run the script in MATLAB to generate CUDA code for the `chromaKey` algorithm.

Deploying a Green Screen Algorithm to NVIDIA Jetson

To deploy the generated code to the NVIDIA Jetson, we need to package all the required files into the `codegen` directory, with the MATLAB commands shown in Code Excerpt 7.

The next step is to copy the entire generated `codegen` folder from the host machine to the NVIDIA Jetson board. After the files have been transferred, we log in to the NVIDIA Jetson directly to build and run the application.

Once logged in to the NVIDIA Jetson, we run the `jetson_clocks.sh` script provided by NVIDIA to maximize the performance of the board, change to the `codegen` directory containing the generated source code we just transferred, and execute the compile command shown in Code Excerpt 8.

Once the executable (`chromaKey`) has been built, the application is run with a USB-connected webcam on the NVIDIA Jetson board with the command shown in Code Excerpt 8. The frames-per-second rate will be displayed on the output.

Figure 4 shows the output from the NVIDIA Jetson board's USB camera before and after the green screen effect.

Comparing Raspberry Pi and NVIDIA Jetson Performance

The greater parallel processing power of the GPU on the NVIDIA Jetson significantly improves the algorithm's performance. The Raspberry Pi achieved approximately 1 frame per second, while the NVIDIA Jetson achieved more than 20 frames per second for an image size of 1280x720—we gained a more than 20-fold speedup without making any modifications or optimizations to our algorithm. We could improve performance even more by optimizing the MATLAB algorithm for more efficient GPU code generation.



FIGURE 4. Before-and-after example of applying the green screen effect.

Summary

In this example we saw how to rapidly generate code for a MATLAB algorithm and deploy it to embedded hardware like the Raspberry Pi. We quickly determined that our algorithm was working correctly and needed to be parallelized. Using MATLAB and GPU Coder, we generated a highly parallel implementation of the algorithm and deployed it to an NVIDIA Jetson board, achieving a significant performance improvement.

LEARN MORE

Raspberry Pi Programming with MATLAB and Simulink
mathworks.com/raspberry-pi-programming

NVIDIA GPU Support from GPU Coder
mathworks.com/nvidia-gpu

CODE EXCERPT 1

```
function Pfinal = chromaKey(P, Pscene, refColorYCbCr, t1, t2)
Cbref = double(refColorYCbCr(1,1,2));
Crref = double(refColorYCbCr(1,1,3));
PYCbCr = rgb2ycbcr(P);
Cb = double(PYCbCr(:, :, 2));
Cr = double(PYCbCr(:, :, 3));
d = (Cb - Cbref).^2 + (Cr - Crref).^2;
t1 = t1^2;
t2 = t2^2;
m = zeros([size(d,1) size(d,2)]);
for j = 1:size(m,1)
    for k = 1:size(m,2)
        if d(j,k) > t2
            m(j,k) = 1;
        elseif d(j,k) > t1
            m(j,k) = (d(j,k) - t1) / (t2 - t1);
        end
    end
end
m = repmat(imgaussfilt(m,0.8), [1 1 3]);
Pfinal = uint8(double(P).*m + double(Pscene).*(1-m));
end
```

CODE EXCERPT 2

```
r = raspi;
cam = cameraboard;
for k = 1:10
    img = snapshot(cam);
end
```

CODE EXCERPT 3

```
refColorRGB = zeros([1,1,3],'uint8');
refColorRGB(1,1,:) = uint8([93 177 21]);
refColorYCbCr = rgb2ycbcr(refColorRGB);
t1 = 28;
t2 = 29;
data = coder.load('background.mat','bg');
scene = data.bg;
% Main loop
for k = 1:1000
    img = snapshot(cam);
    imgFinal = chromaKey(img, scene, ...
        refColorYCbCr, t1, t2);
    figure(1),image(img);
    figure(2),image(imgFinal);
    drawnow;
end
```

CODE EXCERPT 4

```
function chromaKeyApp()
    %Chroma keying example for Raspberry Pi hardware.
    w = matlab.raspi.webcam(0,[1280,720]);
    d = matlab.raspi.SDLVideoDisplay;

    refColorYCbCr = zeros([1,1,3],'uint8');
    refColorYCbCr(1,1,:) = uint8([0 76 98]);
    data = coder.load('background.mat','bg');
    scene = imrotate(data.bg,90);
    % Main loop
    for k = 1:60
        img = snapshot(w);
        img = chromaKey(img, scene, refColorYCbCr, 28, 29);
        displayImage(d,img);
    end
    release(w);
    release(d);
end
```

CODE EXCERPT 5

```
runOnHardware(r,'chromaKeyApp')
```

CODE EXCERPT 6

```
% Create GPU Coder configuration for Jetson TX2
cfg = coder.gpuConfig('exe');
cfg.GpuConfig.MallocMode = 'Unified';
cfg.GpuConfig.ComputeCapability = '6.2';
cfg.GenCodeOnly = 1;
cfg.CustomSource = 'main_webcam.cu';

% Create sample inputs
fg = imread('greenScreenFrame.jpg');
bg = imread('Scenery.jpg');
refColorRGB = [70 130 85]; % RGB light Green
tmpColor = zeros([1,1,3],'uint8');
tmpColor(1,1,:) = uint8(refColorRGB);
refColor = rgb2ycbcr(tmpColor);
threshold1 = 14;
threshold2 = 20;

% Generate CUDA code for chromaKey
codegen -config cfg -args ...
    {fg,bg,refColor,threshold1,threshold2} chromaKey
```

CODE EXCERPT 7

```
% Prepare files for transfer to NVIDIA Jetson TX2
copyfile('Scenery.jpg','codegen/exe/chromaKey/');
copyfile('main_webcam.cu','codegen/exe/chromaKey/');
copyfile(fullfile(matlabroot,'extern','include',...
    'tmwtypes.h'),'codegen/exe/chromaKey/');
copyfile('buildAndRun.sh','codegen/exe/chromaKey/');
```

CODE EXCERPT 8

```
>> sudo ./jetson_clocks.sh
>> cd codegen/exe/chromaKey
>> nvcc -o chromaKey *.cu -rdc=true -arch ...
    sm_62 -O3 `pkg-config --cflags ...
    --libs opencv` -lcudart
>> ./chromaKey 1
```


What Does a Deep Learning Network "See"?

Let's say you've trained a deep learning network to recognize common objects. You run the network on some test data and, hopefully, get the result you're looking for: correct classification of the input image.

Have you ever wondered what your deep learning network actually looked at to produce that result? For example, if a network classifies this image as "French horn," what part of the image mattered most for the classification?

In a paper about visualization techniques for convolutional neural networks, MathWorks developer Birju Patel came across the idea of occlusion sensitivity. If you block out, or occlude, a portion of the image, how does that affect the probability score of the network? And how does the result vary depending on which portion you occlude?

See what Birju discovered

blogs.mathworks.com/deep-learning/2017/12/15/network-visualization-based-on-occlusion-sensitivity/



MATLAB SPEAKS DEEP LEARNING

With just a few lines of MATLAB® code, you can use CNNs and training datasets to create models, visualize layers, train on GPUs, and deploy to production systems.

mathworks.com/deeplearning

