

The *Mathematica*[®] Journal

Volume 9, Issue 1

Contents

Departments

Staff and Contributors	2
Calendar	3
News Bulletins	8
New Products	11
New Resources	21
New Publications	22
Article Submissions to <i>The Mathematica Journal</i>	248
Classifieds	252

Columns

Tricks of the Trade <i>edited by Paul Abbott</i>	31
In and Out <i>edited by Paul Abbott</i>	40
Trott's Corner <i>by Michael Trott</i>	49

Feature Articles

What's New in <i>Mathematica 5</i> <i>by The Mathematica 5 Product Team</i>	59
XML and <i>Mathematica</i> <i>by Pavi Sandhu</i>	96
Building User Interfaces Using <i>J/Link</i> <i>by Todd Gayley</i>	189
<i>Mathematica 4.2: A Technical Review</i> <i>by Lars Hobmuth</i>	216

The electronic version of this issue is available at www.mathematica-journal.com/issue/v9i1.

The online edition of *The Mathematica Journal* includes downloadable versions of all articles and packages in this issue.

On the Cover

Dodecahedron that is bent inward at the center of its faces and extended outward at its vertices. See Demos/Notebooks/Cover Image Construction in the Help Browser of your copy of *Mathematica 5*.

Staff and Contributors

Managing Editor: George Beck

Project Manager: Louise Holubek

Design Team: Jeremy Davis, Heidi Kellner

Compositor: Glenn Scholebo

Archives and Web Production: Jean Buck

Web Development: Andy Hunt, Lori Goodman

Editorial Staff: George Beck, Marcia Krause, Krishnan Ranjani

Customer Service: Matt Doman, Roger Sodsod

Columnists: Paul Abbott, Michael Trott

Contributors: Todd Gayley, Lars Hohnmuth, Pavi Sandhu

Editorial Advisory Board: Steve Christensen (MathTensor, Inc.; Christensen and Associates, Inc.); Donald Glaser (University of California, Berkeley); Donald Knuth (Stanford University); Wolfram Koepf (University of Kassel); Silvio Levy (Mathematical Sciences Research Institute); David Lovelock (University of Arizona); Jerry Marsden (Caltech); Robert Moore (University of Washington, Seattle); Alfred van der Poorten (Macquarie University); Bill Press (Harvard University); Sam Savage (University of Chicago); Dana Scott (Carnegie Mellon University); Lawrence M. Seiford (University of Massachusetts, Amherst); Lynn Steen (St. Olaf College); Francis Sullivan (Institute for Defense Analyses); Bill Thurston (University of California, Davis); Hal Varian (University of California, Berkeley); Rob Wolff (Apple Computer, Inc.); and Stephen Wolfram (Wolfram Research, Inc.).

Customer Services

www.mathematica-journal.com

+1-877-294-8184 (U.S. and Canada)

+1-217-398-6000

fax: +1-217-398-1108

subscriptions: subscribe@mathematica-journal.com

Advertising

The Mathematica Journal accepts advertising for all *Mathematica*-related products and services. Classified advertising is provided for *Mathematica* users. Rates are available at www.mathematica-journal.com/advertising. Contact the advertising department through the telephone numbers above, or send email to ads@mathematica-journal.com.

The Mathematica Journal is typeset in *Mathematica 4* using Janson, Futura, Math CMTT and the standard *Mathematica* fonts.

Publishing

The Mathematica Journal (ISSN 1047-5974) is published by Wolfram Media, Inc. Four-issue subscription rates in U.S. dollars: Electronic version: individuals, \$35; institutions, \$70; students, \$20. Electronic and print versions: individuals, \$75; institutions, \$200; students, \$50. Canadian/Mexican print subscriptions, add \$10. Print subscriptions outside North America, add \$32. Single print copy and back issue sales price: \$25. To subscribe or renew, call +1-217-398-6000 or +1-877-294-8184 (U.S. only), or visit our website at www.mathematica-journal.com to enter your subscription information. Postmaster: Send address changes to: *The Mathematica Journal*, P.O. Box 6359, Champaign, IL 61826-6359.

© 2003 Wolfram Media, Inc. All rights reserved. No part of this publication may be reprinted or otherwise reproduced without the express permission of its publisher. No copyright claimed for U.S. government materials.

Mathematica, *MathLink*, and *MathSource* are registered trademarks of Wolfram Research, Inc. *Mathematica* is not associated with Mathematica Policy Research, Inc. or MathTech, Inc. All other product names mentioned are trademarks of their producers. Material in this publication may not be reproduced in any form without permission of the publisher. The opinions expressed in the *Journal* are the authors' own and do not necessarily reflect those of the authors' employers, the editors, or the publishers of the *Journal*.

Calendar

Professional meetings, conferences, trade shows, exhibits,
and *Mathematica* training courses.

Send your event notices to calendar@mathematica-journal.com.

August

August, 2003

M110: Working with Data Using *Mathematica*

Wolfram Education Group

August 4, Washington, D.C.

Info: www.wolfram.com/services/education/calendar.cgi

M220: Programming in *Mathematica*

Wolfram Education Group

August 5–6, Washington, D.C.

Info: www.wolfram.com/services/education/calendar.cgi

M101: A First Course in *Mathematica*

Wolfram Education Group

August 7–8, New York, New York

Info: www.wolfram.com/services/education/calendar.cgi

August 21, 2003

Introduction to *Mathematica*

CANdienst

Amsterdam, Netherlands

Info: www.candienst.nl/english/cursussen/cursussendetail.asp?id=8

September

September, 2003

M101: A First Course in *Mathematica*

Wolfram Education Group

September 8–9, Washington, D.C.

September 15–16, San Francisco, California

September 18–19, Paris, France

September 25–26, London, England

September 29–30, Philadelphia, Pennsylvania

Info: www.wolfram.com/services/education/calendar.cgi

September 23, 2003

Computeralgebrasimulation mit *Mathematica*

ADDITIVE GmbH

Cologne, Germany

Info: www.additive-net.de/software/workshops/agenda_koeln.shtml

September 24, 2003

Computeralgebrasimulation mit *Mathematica*

ADDITIVE GmbH

Friedrichsdorf, Germany

Info: www.additive-net.de/software/workshops/mm_frankfurt.phtml

September 25, 2003

Computeralgebrasimulation mit *Mathematica*

ADDITIVE GmbH

Munich, Germany

Info: www.additive-net.de/software/workshops/agenda_muenchen.phtml

September 25–26, 2003

***Mathematica* Intermediate and Programming Course**

CANDiensten

Amsterdam, Netherlands

Info: www.candiensten.nl/english/cursussen/cursussendetail.asp?id=9

September 25–26, 2003

PrimMath[2003] Conference: *Mathematica* in Science, Technology, and Education

SYSTEMCOM LTD

Zagreb, Croatia

Info: primmath.math.hr

October

October, 2003

M220: Programming in *Mathematica*

Wolfram Education Group

October 2–3, San Francisco, California

Info: www.wolfram.com/services/education/calendar.cgi

M330: Neural Networks

Wolfram Education Group

October 2–3, Copenhagen, Denmark

Info: www.wolfram.com/services/education/calendar.cgi

M101: A First Course in *Mathematica*

Wolfram Education Group

October 9–10, Boston, Massachusetts

October 20–21, New York, New York

October 20–21, Chicago, Illinois

October 21–22, Los Angeles, California

October 23–24, San Francisco, California

Info: www.wolfram.com/services/education/calendar.cgi

M310: Digital Image Processing

Wolfram Education Group

October 13–14, Washington, D.C.

Info: www.wolfram.com/services/education/calendar.cgi**M110: Working with Data Using *Mathematica***

Wolfram Education Group

October 20, Los Angeles, California

Info: www.wolfram.com/services/education/calendar.cgi**M220: Programming in *Mathematica***

Wolfram Education Group

October 23–24, Boston, Massachusetts

Info: www.wolfram.com/services/education/calendar.cgi

October 7, 2003

web*Mathematica* Einführung

ADDITIVE GmbH

Friedrichsdorf, Germany

Info: www.additive-net.de/software/workshops/agenda_webmath.phtml

October 8, 2003

web*Mathematica* Workshop mit Arbeit am Webserver

ADDITIVE GmbH

Friedrichsdorf, Germany

Info: www.additive-net.de/software/workshops/agenda_webmath2.phtml

October 23–24, 2003

M101: A First Course in *Mathematica*

mathemas ordinate

Berlin, Germany

Info: www.wolfram.com/services/education/calendar.cgi

October 15–17, 2003

***Mathematica* Workshop**

QT software GmbH

Munich, Germany

Info: www.qtsoftware.de/mathsem2.htm

November

November 3–4, 2003

M101: *Mathematica*, premier cours

Wolfram Education Group

Paris, France

Info: www.wolfram.com/services/education/calendar.cgi

November, 2003

M101: A First Course in *Mathematica*

Wolfram Education Group

November 17–18, San Francisco, California

November 17–18, Washington, D.C.

Info: www.wolfram.com/services/education/calendar.cgi

M220: Programming in *Mathematica*

Wolfram Education Group

November 3–4, London, England

November 6–7, New York, New York

November 20–21, Washington, D.C.

Info: www.wolfram.com/services/education/calendar.cgi

M110: Working with Data Using *Mathematica*

Wolfram Education Group

November 5, New York, New York

November 19, San Francisco, California

November 19, Washington, D.C.

Info: www.wolfram.com/services/education/calendar.cgi

M330: Neural Networks

Wolfram Education Group

November 13–14, Washington, D.C.

Info: www.wolfram.com/services/education/calendar.cgi

November 4, 2003

Computeralgebrasimulation mit *Mathematica*

ADDITIVE GmbH

Friedrichsdorf, Germany

Info: www.additive-net.de/software/workshops/mm_frankfurt.phtml

November 5, 2003

Computeralgebrasimulation mit *Mathematica*

ADDITIVE GmbH

Stuttgart, Germany

Info: www.additive-net.de/software/workshops/agenda_stuttgart.phtml

November 6, 2003

Computeralgebrasimulation mit *Mathematica*

ADDITIVE GmbH

Munich, Germany

Info: www.additive-net.de/software/workshops/agenda_muenchen.phtml

November 19, 2003

***Mathematica* Grundlagen**

ADDITIVE GmbH

Friedrichsdorf, Germany

Info: www.additive-academy.de/schulungen/software/mathematica/mma.shtml

November 20, 2003

Mathematica Programmierung

ADDITIVE GmbH

Friedrichsdorf, Germany

Info: www.additive-academy.de/schulungen/software/mathematica/mma.shtml

December

December, 2003

M101: A First Course in Mathematica

Wolfram Education Group

December 1–2, Honolulu, Hawaii

December 4–5, Atlanta, Georgia

December 8–9, New York, New York

Info: www.wolfram.com/services/education/calendar.cgi

M101: Mathematica, premier cours

Wolfram Education Group

December 15–16, Geneva, Switzerland

Info: www.wolfram.com/services/education/calendar.cgi

December 10–11, 2003

Maîtrise de Mathematica

Ritme Informatique

Paris, France

Info: www.ritme.com/fr/formations/programm.html

December 12, 2003

Perfectionnement de Mathematica

Ritme Informatique

Paris, France

Info: www.ritme.com/fr/formations/programm.html

News Bulletins

View more *Mathematica* news at www.wolfram.com/news.

***Mathematica* Celebrates 15th Anniversary with New High-Performance Release**

When Wolfram Research first released *Mathematica* 1.0 on June 23, 1988, it was immediately lauded as a dramatic advance. Wolfram Research continues to lead the world of technical computing with the release of *Mathematica* 5, the new high-performance version of this award-winning system. To learn more, visit www.wolfram.com/news/15anniversary.html.

***Macworld* Editors' Choice Award**

Mathematica received *Macworld's* 18th Annual Editors' Choice Award in the Science/Engineering Software category. For more information, visit www.macworld.com/2002/12/features/eddy.

***Mathematica* Developer Conference**

The annual *Mathematica* Developer Conference took place in April 2003 at the Wolfram Research headquarters in Champaign, Illinois. For a listing of presentation notebooks, including discussion of new technologies in *Mathematica* 5, visit www.wolfram.com/news/events/devconf2003/notebooks.html.

***Mathematica* Teacher's Edition for Mac OS X**

Mathematica Teacher's Edition, a product designed especially for high school teachers and community college professors, is now available for Mac OS X and is part of Apple's Math Curriculum Mobile Lab. A fully localized Japanese version of *Mathematica Teacher's Edition* is also available. To learn more, visit www.wolfram.com/news/mteforosx.html.

Download the New *Web Services Package* 1.0 Beta

Wolfram Research is inviting users to try its new *WebServices Package*, which integrates *Mathematica* and web services by letting a user call web services from

Mathematica. Web Services Package gives *Mathematica* users access to additional data and processing services, such as geographical data, financial data, remote calculation, language translation, file conversion, and Wolfram Research online information services, from directly within *Mathematica*. For more information, visit www.wolfram.com/news/webservices.html.

Real-Time Data Feeds for *Mathematica* with HyperFeed

Finance users requiring live real-time data feeds for their trading and risk management applications that are based on *Mathematica* technology now have an extremely fast and simple solution. HyperFeed Technologies, one of the leading providers of real-time and historical market data of various financial instruments, offers subscribers of its data a Java API that can be accessed very easily from the *Mathematica* kernel through *J/Link*. For more information, go to www.wolfram.com/news/hyperfeed.html.

South Africa's National Disaster Atlas Uses web*Mathematica* Technology to Fight Catastrophic Events

In South Africa, disasters such as droughts, floods, and famines are common occurrences. While emergency supplies are often available, they are useful only if government officials can anticipate when and where disaster will strike. Dusan Sakulski, System Integrator and Coordinator of the National Disaster Management Centre of South Africa, is using web*Mathematica* to help predict the time, location, and intensity of natural disasters. For more information, view the full story at www.wolfram.com/news/disasteratlas.html.

Veteran *Mathematica* User Steven Skiena Is Showing the Fun of Discrete Mathematics

Steven Skiena, the original developer of *Combinatorica* is anticipating the release of his most recent book, *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Co-authored with Sriram Pemmaraju, a fellow computer scientist and *Combinatorica* developer, the book is "the definitive guide to *Combinatorica*." Experimenting with *Combinatorica* provides an exciting new way to learn combinatorics and graph theory. Says Skiena, "We want to show the fun of discrete mathematics." For more information, go to www.wolfram.com/news/skiena.html.

***Mathematica* Ported to NEC's Itanium Linux Platform**

Wolfram Research, Inc. and NEC Corporation are pleased to announce the release of *Mathematica* 4.2 for the Itanium Linux platform. Intel's Itanium 64-bit

architecture is designed for demanding next-generation computing applications such as *Mathematica*. For more information, see www.wolfram.com/news/itanium.html.

Fraunhofer-Gesellschaft Formally Increases *Mathematica* Usage

Fraunhofer-Gesellschaft, a collection of over 55 highly specialized applied research institutes, has designated *Mathematica* as its strategic technical computing software. Fraunhofer has been an important force in German science and industry for over 50 years. For more information, visit www.wolfram.com/news/fraunhofer.html.

***Mathematica* Classroom Gets Students Interested in Math**

High school math will never be the same at Torrey Pines High School in San Diego, California. Abby Brown and her students are pushing the limits of traditional learning with *Mathematica*. To learn more, see www.wolfram.com/news/abbybrown.html.

Art Inspires a Lesson in Calculus

Croatian math and computer science teacher Maja Cvitkovic is not content with the same old math lessons. So when she had the opportunity to use *Mathematica* with a group of teenage students at a workshop in Zagreb, she devised a lesson that gave them a chance to play. The assignment: Create a picture using random dots. More information is available at www.wolfram.com/news/croatiacalculus.html.

New Products

Mathematica 5

Newly Released Mathematica 5 Outperforms Dedicated Numerical Systems

Mathematica 5 is the new high-performance version of Wolfram Research's award-winning technical computing software. Key new technologies enable *Mathematica 5* to outperform dedicated numerical systems in raw computational speed, while introducing a host of innovative features.

Until now, technical users had to make a choice between *Mathematica*'s comprehensive, mixed numeric-symbolic environment and the faster numerical routines available in Fortran libraries or specialized packages such as MATLAB, MATRIXx, or O-Matrix. With the release of *Mathematica 5*, this choice is no longer necessary.

"In some cases, *Mathematica 5* is 1,000 times faster than previous versions and surpasses the speed of dedicated numerical systems too," says Tom Wickham-Jones, Director of Strategic Kernel Development for Wolfram Research. "Yet to achieve this, we've compromised none of the accuracy or expert nature of *Mathematica*. To the contrary, we've enhanced them while at the same time adding even more capabilities."

Mathematica 5 introduces extensive new functionality, much of which is based on algorithms that are exclusive to *Mathematica 5*. Other algorithms provide functionality that was up to now available only in custom packages costing tens of thousands of dollars. "The most impressive achievement is the quantity of original research that went into this version—over 100 new algorithms for symbolic and numeric computation have been implemented by in-house developers," says Roger Germundsson, Director of R & D for Wolfram Research.

Mathematica 5 also extends Wolfram Research's position as the leader in providing integration with other software and standards. In addition to updated versions of *J/Link* and *MathLink*, *Mathematica*'s standard connections to Java and C/C++, *Mathematica 5* comes with the new *.NET/Link*, a much requested feature that allows developers to seamlessly integrate *Mathematica* into applications using Microsoft's .NET Framework.

Major new features and enhancements in *Mathematica 5* include:

- Record-breaking speed for numerical linear algebra
- Wide-ranging support for fast sparse matrix operations
- New-generation optimized numerical solvers for ordinary and partial differential equations

- Major new algorithms for solving equations and inequalities symbolically over complex numbers, reals, and integers
- Fully integrated solver for differential algebraic equations
- High-performance optimization and linear programming, including interior point method
- Extensive support for vector and array functions in numeric solvers
- State-of-the-art solver for recurrence equations
- Broader support for domain specifications in symbolic computation
- *.NET/Link* for full integration with Microsoft's .NET Framework
- Flexible import and export of DICOM, PNG, SVG, and sparse matrix formats
- Optimized versions available for 64-bit hardware and operating systems
- New quick-start interactive tutorial

The advances in *Mathematica 5* strengthen *Mathematica*'s position of not only having the broadest scope of any technical computing system but also providing the best performance. *Mathematica* accomplishes this by combining outstanding numeric, symbolic, and graphical capabilities with a uniquely productive programming language and interactive document system. In *Mathematica 5*, technologies such as packed arrays, automatic algorithm selection, and symbolic preprocessing—which were developed in phase two of Wolfram Research's gigaNumerics initiative—have helped to deliver a unique combination of both speed and functionality. “The ultimate objective of the gigaNumerics initiative was to equip *Mathematica* to handle giga-sized numerical data sets while maintaining *Mathematica*'s unique breadth of capability. With *Mathematica 5*, we have already produced a tremendous return on our investment,” adds Wickham-Jones.

“Now is the time to switch from dedicated technical systems to *Mathematica*,” says Conrad Wolfram, Director of Strategic Business Development for Wolfram Research. “*Mathematica 5*'s unique combination of speed, scope, and scalability takes you from start to finish, from prototyping to final computations, from research to presentation in the shortest possible time.”

More info

www.wolfram.com/products/mathematica

Mathematica for Students 5

Mathematica for Students 5 Offers All the Power of Mathematica at a Special Student Price

Mathematica for Students 5 is now available, giving students access to the latest *Mathematica* technology at a special low price. *Mathematica for Students 5* contains all of the same innovative features and functionality as *Mathematica 5*, the new

high-performance version of Wolfram Research's award-winning technical computing software that is part of the information infrastructure at top universities, leading corporations, and government organizations worldwide.

Mathematica for Students 5 surpasses other educational computing software with its unique combination of speed, functionality, and ease of use. A new quick-start interactive tutorial will help students familiarize themselves with *Mathematica*, while key new technologies deliver tremendous performance improvements and further extend the scope of *Mathematica*'s already broad capabilities. Numerical computations in Version 5 are vastly faster than in earlier versions, meaning that students will be able to complete their assignments in less time than ever before. *Mathematica for Students 5* can also tackle significantly larger, more challenging and computationally intensive problems, giving students a virtually limitless environment in which to experiment and do their work.

"*Mathematica 5* is a much more powerful program. It represents the state of the art in numerical and symbolic algorithms and, as a result, runs like greased lightning," says Joanne Mason, Director of Math Studies at the prestigious Harker School, located in California's Silicon Valley. "Our students use *Mathematica* extensively in all math classes from precalculus on, as well as in many of our upper-level science and computer science courses." With over 2,500 functions for technical computation, *Mathematica for Students* is suitable for all levels and fields of study from high school through graduate school, including computer science, economics, engineering, mathematics, the physical sciences, and statistics.

With *Mathematica for Students*, students can:

- Perform all of the symbolic and numeric calculations needed for their classes
- Use built-in palettes for instant access to hundreds of functions
- Plot 2D and 3D graphics from data and functions
- Connect to Java, C, .NET, and other languages
- Import, export, and analyze data, graphics, and sound in over 40 standard file formats
- Produce professional-looking reports incorporating formulas and graphics
- Publish work on the web by saving as HTML, XML, and MathML

In addition to adding extensive new functionality, *Mathematica for Students 5* provides enhancements to its uniquely productive programming language and publication-quality technical word processing. Extensive integration with other software and standards also makes *Mathematica for Students* ideal for use in conjunction with other tools, including Wolfram Research's other educational products such as *Mathematica Teacher's Edition*, *Calculus WIZ*, and *The Mathematical Explorer*.

Using *Mathematica for Students* in their studies will help students develop skills that will be needed throughout their educational careers and that are sought by future employers. “*Mathematica* can take the pain out of the ‘brute-force’ calculations that often overshadow the ideas that students need to learn,” says Barry Cherkas, Professor of Mathematics at CUNY’s Hunter College, which operates under a systemwide *Mathematica* license. “It permits us to aim for a higher level of mathematical sophistication than is otherwise possible without this technology.”

More info

www.wolfram.com/products/student/mathforstudents

webMathematica 2.0

Wolfram Research Releases webMathematica 2.0

webMathematica 2.0 from Wolfram Research incorporates features that make it even easier for users to build sophisticated, interactive web applications with the full calculation and visualization power of *Mathematica* technology.

“The original release of webMathematica opened the door to incorporating a whole new generation of interactive functionality into websites, and webMathematica 2.0 builds upon the latest web development standards to further increase this access and make the product more robust,” says Lars Hohmuth, Manager of Strategic Marketing at Wolfram Research. Improvements in Version 2.0 are immediately apparent, starting with a simplified installation process that requires minimal configuration beyond the installation of the webMathematica web application, thereby enabling most customers to set up their websites in less than 15 minutes.

“webMathematica 2.0 blows everything else away.... I am [currently] using PHP, MySQL, and graphics libraries to do things that would be very simple with webMathematica 2.0,” says Ed Luschei, Assistant Professor in Agronomy at the University of Wisconsin.

webMathematica now includes *Mathematica* 4.2, which has many features that are integral to web operations, such as XML, SVG, and MathML capabilities. In addition to the added functionality of the *Mathematica* 4.2 computation engine, the technology behind webMathematica 2.0 uses a new HTML templating mechanism based on JavaServer Pages (JSP) custom tags. This mechanism is fully compatible with the mechanism used in Version 1.0 but is easier to understand. It also allows the use of other JSP custom tag libraries and facilitates the integration of webMathematica into other server applications.

Additional new features in webMathematica 2.0 include:

1. Support for MathML and SVG, which lets users describe equations and graphics in standard XML formats that can be read by browsers
2. Support for catching *Mathematica* **Message** and **Print** output statements, to aid in developing and debugging application content

3. Support for HTTP file upload, a mechanism that enables users to submit data to a web $Mathematica$ web server
4. Support for saving HTTP session variables on the server, which is useful for saving results from one computation to another
5. New HTML formatting functions for formatting results such as tables into HTML

“With web $Mathematica$ 2.0, our focus was on making the product fully compatible with existing web standards. We invested a large amount of design time in implementing a more simplified architecture based on standard JSP and XML technologies. At the same time, we arrived upon a more optimal configuration of components for easier installation and development,” says Tom Wickham-Jones, Director of Strategic Kernel Technology at Wolfram Research and chief developer of web $Mathematica$. “The end result is a more elegant and reliable product that is also more intuitive, enabling faster construction of pages and increased performance.”

More info

www.wolfram.com/products/webmathematica

grid $Mathematica$

gridMathematica Provides Cost-Effective Mathematica Software for Grids

With the release of grid $Mathematica$, Wolfram Research offers a complete parallel computing solution for dedicated grids or clusters. It offers a highly cost-effective way of deploying $Mathematica$ for parallel computations. grid $Mathematica$ can run on any cluster of machines, including Unix, Linux, Windows, and Mac OS X, and requires no more than TCP/IP connectivity.

According to Roman Maeder, creator of *Parallel Computing Toolkit* (a key component of grid $Mathematica$), “One of my main motivations for writing this package was to make parallel computing easily available to a wide range of workgroups. grid $Mathematica$ provides companies and government institutions with affordable $Mathematica$ licensing and simplified maintenance that matches the evolution of declining hardware costs and increasing accessibility of clusters.”

This common-sense approach resonates well with customers and industry leaders. “Sun Microsystems, a long-time platform partner of Wolfram Research, is excited about the scalability that grid $Mathematica$ offers to high-end users. We are seeing increasing demand for vast simulations and computations. grid $Mathematica$ and Sun’s product line—from Linux-based LX50 blade servers to 64-bit SPARC server clusters—provide the scalable solution customers need,” states Joerg Schwarz, Senior Group Manager for Scientific and Engineering Computing at Sun Microsystems.

The benefits of grid $Mathematica$ go beyond pure number crunching. Notes Theodore Gray, Director of User Interfaces at Wolfram Research, “For Stephen

Wolfram's keynote at COMDEX 2002, we were able to create intricate animations of cellular automata in mere hours. After writing and testing code on an iMac, we moved it over to a 16-way Xserve cluster running Mac OS X Server, thereby cutting calculation time from days to less than an hour. Performance gains like this open up new opportunities for Macintosh users in science, engineering, and the arts."

"Apple's new Xserve running Mac OS X Server provides a superior and cost-effective platform for parallel computing solutions such as grid*Mathematica*," says Ron Okamoto, Apple's Vice President of Worldwide Developer Relations. "The Unix-based architecture and ease of use of Mac OS X delivers a powerful platform that scales from single-user solutions to clustered environments."

grid*Mathematica* supports all common parallel programming paradigms, such as virtual shared or distributed memory, automatic or explicit scheduling, and concurrency, including synchronization, locking, and latency hiding. Other features include machine-independent implementation and parallel functional programming as well as failure recovery and automatic reassignment of stranded processes in the event of a system failure.

Says Roger Germundsson, Director of Research and Development at Wolfram Research, "grid*Mathematica* is the result of working closely with some of our key customers and reacting to their needs. By introducing grid*Mathematica* as an integrated product, we have been able to simplify deployment from procurement to installation. We will build on this foundation by providing further additions such as training classes and other services."

More info

www.wolfram.com/products/gridmathematica

Mathematica for the Classroom

Mathematica for the Classroom is a special edition of *Mathematica* for precollege and community college classrooms and computer labs. It gives you the same power and functionality as professional *Mathematica* at an exclusive low price for the precollege level.

Schools around the world are using *Mathematica* to teach algebra, trigonometry, calculus, physics, chemistry, economics, computer science, and many other courses. Students who use *Mathematica* have a head start on getting familiar with tools that will improve their performance in university classes and in their careers. Don't let your students get left behind.

When you provide *Mathematica for the Classroom*, your students will have the tools they need to investigate and explore technical subjects. For example, future engineers can model the real world, future scientists can simulate physical and social systems, future mathematicians can conjecture and prove theorems, and future programmers can create effective and efficient pieces of code. In short, *Mathematica for the Classroom* promotes learning experiences that textbooks cannot.

The *Mathematica* notebook file format is useful for creating platform-independent documents that can be shared among students, teachers, and colleagues. When used in a computer lab or with an overhead projector, notebooks work great for presenting tutorials, lectures, and demonstrations. Since notebooks display traditional mathematical notation, they are perfect for composing hand-outs, tests, and assignments that can be either printed or distributed, completed, and returned electronically.

Mathematica for the Classroom helps you to visually reinforce important concepts with full-color graphics and interactive exercises created with palettes (sets of buttons that correspond to *Mathematica* commands). Palettes are the intuitive way to create documents and tutorials for your students. You can customize as many palettes as you need to organize the features that you use the most.

Mathematica for the Classroom is available for Windows, Macintosh, and Linux platforms. There is also a fully localized Japanese edition. To purchase your copy, or to find out more about volume discounts and special purchase programs for your precollege lab, please contact one of our sales representatives or your local reseller.

More info

www.wolfram.com/products/classroom

***A New Kind of Science* Poster**

This unique poster is a complete representation of the text of *A New Kind of Science*. Overlaid with the pattern of cellular automaton rule 30, each “cell” is a page out of the book and can be read with the aid of a powerful magnifying glass.

More info

store.wolfram.com/view/misc/posters_1.str

New *Mathematica* Applications

***Algebra, Trigonometry, & Mathematica* Download**

Algebra, Trigonometry, & Mathematica (AT&M) is interactive courseware that helps students learn precalculus by doing mathematics instead of reading about it.

More info

store.wolfram.com/view/book/D0711.str

Control System Professional Suite

This product provides powerful, integrated tools for control in an extensible framework of tightly integrated *Mathematica* application packages.

- *Control System Professional 2*
Control System Professional offers an object-oriented environment for solving common problems in control and systems areas.
- *Advanced Numerical Methods*
This application package expands the functionality of *Control System Professional* with an extensive collection of state-of-the-art numerical algorithms.

More info

www.wolfram.com/products/applications/csps

Derivatives Expert III

This updated version forms a high-end financial analysis and engineering tool for banks, asset management companies, and financial exchanges.

More info

www.wolfram.com/products/applications/derivatives

Experimental Data Analyst 1.1

From extensive data fitting capabilities to data visualization and transformation, *Experimental Data Analyst* helps you get the most out of your experimental data.

More info

www.wolfram.com/products/applications/eda

Fuzzy Logic 2

Included in *Fuzzy Logic 2* are a large number of new functions to enhance the already robust set of utilities available with the package as well as other performance improvements to help make it the most flexible environment for exploring fuzzy systems.

More info

www.wolfram.com/products/applications/fuzzylogic

Geometrica02

Geometrica uses the symbolic engine of *Mathematica* to help you create perfect figures and perform analytical geometry and tests.

More info

www.wolfram.com/products/applications/geometrica

machine learning framework 1.1

Offering a complete solution for engineers, professionals, and all experts who want to extract computational models from data, the new version of *machine learning framework* speeds up machine learning kernel functions and improves the numerical scheme for the optimization of fuzzy rules.

More info

www.wolfram.com/products/applications/mlf

Neural Networks

Neural Networks gives professionals and students the tools to train, visualize, and validate neural network models. Examples on paper quality and exchange rate that show the capabilities of *Neural Networks* can be found in the *Mathematica Information Center*.

More info

www.wolfram.com/products/applications/neuralnetworks

A New Kind of Science Explorer: Mathematica Kit

Based on the same programs that Wolfram used to create the striking graphics in his book, *A New Kind of Science Explorer: Mathematica Kit* enables the user to take full advantage of *Mathematica*'s integrated environment.

More info

www.wolframscience.com/nksx/nksxmk.html

Operations Research 3.0

Operations Research 3.0 is a *Mathematica* package for solving problems in linear optimization, quadratic programming, shortest path tasks, and combinatorial optimization and heuristics.

More info

www.wolfram.com/products/applications/operationsresearch

RiskQ 4.2

Using empirical data, *RiskQ* helps build and manipulate population distributions for conducting customized statistical tests.

More info

www.wolfram.com/products/applications/riskq

SchematicSolver

SchematicSolver is the first mouse-driven, interactive drawing tool based entirely on *Mathematica*'s built-in functions, graphics primitives, and palettes.

More info

www.wolfram.com/products/applications/schematicsolver

Tensors in Physics

Tensors in Physics allows the user to compute tensor calculations with hundreds or thousands of components and can also be extended by the addition of user-defined functions.

More info

store.wolfram.com/view/book/D0709.str

UnRisk PRICING ENGINE 1.7

Combine speed with precision for fast, accurate pricing of financial derivatives in *Mathematica* with *UnRisk PRICING ENGINE*. Dozens of exotic options, including Asian, double barrier, chooser, digital, lookback, and supershare options, lead the list of new features in Version 1.7.

More info

www.wolfram.com/products/applications/unrisk

New Resources

Mathematica Information Center

The *Mathematica Information Center* is a treasury of thousands of *Mathematica*-related items, all in one place. This new online resource combines materials from the long-lived and popular *MathSource* website and the former *Mathematica* Resource Library, submissions from *Mathematica* users, and new content from Wolfram Research. Fully indexed and equipped with full-text and advanced search capabilities, the center currently contains nearly five thousand items, and new entries are being added daily.

A What's New page at library.wolfram.com/infocenter/WhatsNew highlights recent additions to the *Mathematica Information Center*, while an updated subject-classification scheme makes finding materials easier than ever. All items are presented in a standardized format that includes summaries, bibliographic information, and in many cases links to the full text.

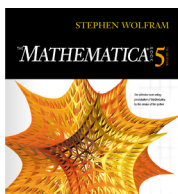
Continuing in the tradition started with *MathSource*, we encourage users to contribute materials and suggestions to the *Mathematica Information Center* by visiting library.wolfram.com/contribute. Sharing programs and notebooks with other users makes *Mathematica* a richer, more valuable tool for everyone.

To learn more, visit the *Mathematica Information Center* at library.wolfram.com.

Graphics Gallery

Take an interactive visual tour of our new Graphics Gallery, an online collection of *Mathematica* graphics created by users around the world. It features examples of 2D graphics, polyhedra, surfaces, and animations that can be used for academic, professional, or recreational purposes. For example, you can download images to enliven your website or to help you visualize a concept. To view the graphics or to contact us about having your work included in the collection, visit gallery.wolfram.com.

New Publications



The *Mathematica* Book, Fifth Edition

Stephen Wolfram, 2003, Wolfram Media, Inc.,
1,488 pp., hardcover

As both a highly readable tutorial and a definitive reference for over a million *Mathematica* users worldwide, this book covers every aspect of *Mathematica*. It is an essential resource for all users of *Mathematica* from beginners to experts. This expanded fifth edition presents *Mathematica* Version 5 for the first time and is important for anyone interested in the progress of advanced computing.

Included in this new edition are the following:

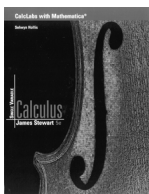
- Visual tour of key features
- Practical tutorial introduction
- Full descriptions of 1,200+ built-in functions
- Thousands of illustrative examples
- Easy-to-follow descriptive tables
- Essays highlighting key concepts
- *Mathematica* language tutorial
- Guide to symbolic programming
- Introduction to document-centered interfaces
- Guide to the *MathLink* API
- Notes on internal implementation
- Index with 10,000+ entries

Chapters

A Tour of *Mathematica* | A Practical Introduction to *Mathematica* | Principles of *Mathematica* | Advanced Mathematics in *Mathematica* | Appendix: *Mathematica* Reference Guide

Available:

store.wolfram.com/view/book/ISBN1579550223.str



Calculus with *Mathematica* for Stewart's Single Variable Calculus

Selwyn Hollis, 2003, Brooks/Cole, 294 pp., paperback

This manual was written to accompany the fifth edition of James Stewart's *Calculus*, which is used widely in calculus courses. The goals of the manual are to show students how *Mathematica* can help them in learning and using calculus, to introduce them to many of *Mathematica*'s capabilities, and to present the central ideas of calculus. *Mathematica* is used as a learning tool to explore calculus concepts and the way they can be used to solve problems. At the end of the manual are 24 projects covering a wide range of topics that reinforce important calculus concepts and are arranged to correspond to the same material in Stewart's *Calculus*.

Chapters

Mathematica Basics | Limits and the Derivative | Applications of the Derivative | Integration | Applications of the Integral | Differential Equations | Parametric and Polar Curves | Sequences and Series | Projects | Appendices

Available:

store.wolfram.com/view/book/ISBN0534393713.str



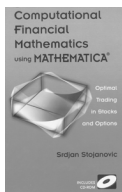
Challenging the Boundaries of Symbolic Computation: Proceedings of the 5th International *Mathematica* Symposium

Peter Mitic, Philip Ramsden, and Janet Carne, editors, 2003, Imperial College Press, 420 pp., hardcover, includes CD-ROM

This book contains the proceedings of the 5th International *Mathematica* Symposium held in July 2003. Included are 53 papers covering a range of applications in fields such as algebraic computation, applied mathematics, education, physics, pure mathematics, statistics and probability, and visualization.

Available:

store.wolfram.com/view/book/ISBN1860943632.str



Computational Financial Mathematics Using *Mathematica*: Optimal Trading in Stocks and Options

Srdjan Stojanovic, 2003, Birkhäuser, 481 pp., hardcover,
includes CD-ROM

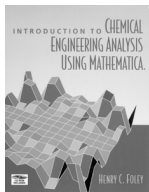
This book provides a comprehensive overview of the mathematics of computational finance for academics in financial mathematics. Enabled by *Mathematica*, the approach is applied, integrating and giving a fresh perspective to many mathematical disciplines including probability, ordinary, stochastic, and partial differential equations; mathematical statistics; and calculus of variations. Three kinds of solution methods are emphasized: symbolic, numerical, and Monte Carlo.

Chapters

Introduction | Cash Account Evolution | Stock Price Evolution | European Style Stock Options | Stock Market Statistics | Implied Volatility for European Options | American Style Stock Options | Optimal Portfolio Rules | Advanced Trading Strategies | Bibliography | Index

Available:

store.wolfram.com/view/book/ISBN0817641971.str



Introduction to Chemical Engineering Analysis Using *Mathematica*

Henry C. Foley, 2002, Academic Press, 509 pp., hardcover,
includes CD-ROM

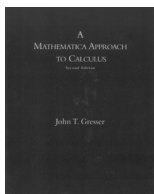
This book provides an introduction to chemical engineering analysis—the processes and designs used to manufacture, use, and dispose of chemical products. Core concepts ranging from the conservation of mass to chemical kinetics are covered. At the same time, the text shows how to use *Mathematica*—from the basics of writing a few lines of code through developing entire analysis programs. Analysis and computation are explained simultaneously, enabling the student to examine a system, determine the key characterizing variables, write down an appropriate statement for the conservation of mass, and then solve the problem using *Mathematica*.

Chapters

A Primer of *Mathematica* | Elementary Single-Component Systems | The Draining Tank and Related Systems | Multiple-Component Systems | Multiple Phases—Mass Transfer | Adsorption and Permeation | Reacting Systems—Kinetics and Batch Reactors | Semi-Continuous Flow Reactors | Continuous Stirred Tank and the Plug Flow Reactors | Worked Problems | Index

Available:

store.wolfram.com/view/book/ISBN0122619129.str



A Mathematica Approach to Calculus, Second Edition

John T. Gresser, 2002, Prentice Hall, 303 pp., paperback

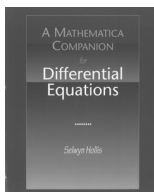
This calculus manual, written for *Mathematica* 4.1 for the Macintosh, is meant to be used as a supplement to a regular calculus text. Topics are covered in a way that parallels the sequence in a typical calculus course. By focusing complete attention on an issue of calculus rather than on the computation, the text shows how *Mathematica* can be used to gain a deeper understanding of calculus.

Chapters

Some Preliminaries | Basic *Mathematica* | Basic Integration | Applications of Integration | The Transcendental Functions | Techniques of Integration | Sequences and Series | Plane Curves and Polar Coordinates | Vectors and Analytic Geometry | Vector-Valued Functions | Multivariate Differential Calculus | Multiple Integration

Available:

store.wolfram.com/view/book/ISBN0130920150.str



A Mathematica Companion for Differential Equations

Selwyn Hollis, 2003, Prentice Hall, 271 pp., paperback

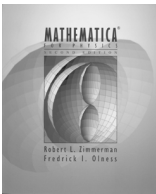
This text is designed to be used for a typical two-semester course in differential equations. It emphasizes numerical solutions and graphics and uses *Mathematica* extensively for animations and visualizations. It covers topics from fundamental concepts through nonlinear systems, Sturm-Liouville eigenvalue problems, and the diffusion and wave equations.

Chapters

Introduction to *Mathematica* | Introduction to Differential Equations | Linear First-Order Equations and Applications | Nonlinear First-Order Equations and Applications | Approximation of Solutions | Coupled Pairs of First-Order Equations | Second-Order Equations and Applications | Linear Differential Systems | Nonlinear Autonomous Systems in the Plane | A Gallery of Nonlinear Systems | Fourier Series and Sturm-Liouville Eigenvalue Problems | Linear Partial Differential Equations | Appendix: The DiffEqs Packages

Available:

store.wolfram.com/view/book/ISBN0130463299.str



Mathematica for Physics, Second Edition

Robert L. Zimmerman and Fredrick I. Olness, 2002, Addison-Wesley, 645 pp., paperback

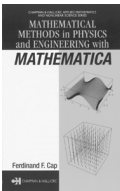
This revised and updated book is intended for undergraduates, graduate students, and practicing physicists who want to learn new *Mathematica* techniques for solving a general class of physics problems. The experimental activities included are designed to deepen and broaden the reader's understanding of physics. The book assumes some familiarity with *Mathematica*, but the focus is on physics rather than on rudimentary *Mathematica* techniques.

Chapters

Getting Started | General Physics | Oscillating Systems | Nonlinear Oscillating Systems | Discrete Dynamical Systems | Langrangians and Hamiltonians | Orbiting Bodies | Electrostatics | Quantum Mechanics | Relativity and Cosmology | Index

Available:

store.wolfram.com/view/book/ISBN0805387005.str



Mathematical Methods in Physics and Engineering with Mathematica

Ferdinand F. Cap, 2003, CRC Press, 340 pp., hardcover

This book provides a practical reference for solving difficult problems involving ordinary and partial differential equations and boundary value problems using *Mathematica*. A variety of applications are explored, including the diffusion of a

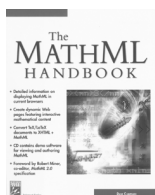
perfume, the spread of infectious diseases, and the critical mass of a nuclear bomb.

Chapters

Introduction | Boundary Problems of Ordinary Differential Equations | Partial Differential Equations | Boundary Problems with One Closed Boundary | Boundary Problems with Two Closed Boundaries | Nonlinear Boundary Problems | Appendix

Available:

store.wolfram.com/view/book/ISBN1584884029.str



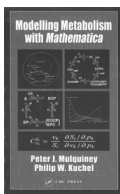
The MathML Handbook

Pavi Sandhu, 2003, Charles River Media, 518 pp., paperback, includes CD-ROM

This resource for the new technology of displaying mathematics on the web provides a detailed tutorial introduction to MathML, the XML-based markup language, and includes many examples to illustrate the syntax. The book begins with an overview of MathML and its practical applications. It then moves on to a clear exposition of the basic concepts of MathML, including XML syntax, presentation markup, content markup, and combined markup. Additional chapters deal with advanced topics such as using CSS and XSLT style sheets to control the display of MathML, converting TeX equations to MathML, and using JavaScript with MathML to create dynamic math websites featuring interactive equations. The book will be useful to anyone interested in working with mathematics on the web, including scientists, engineers, web developers, publishers, teachers, and students.

Chapters

Introduction to MathML | XML Primer | Fundamentals of MathML | Presentation Markup | Content Markup | Combining Presentation and Content Markup | Displaying MathML in Web Browsers | Applying Styles and Transformations | Authoring and Editing MathML | Converting between TeX and MathML | Using MathML for Computations | Creating Dynamic Math Web Pages | Presentation Elements | Content Elements | MathML Characters



Modelling Metabolism with *Mathematica*

Peter J. Mulquiney and Philip W. Kuchel, 2003, CRC Press, 309 pp., hardcover

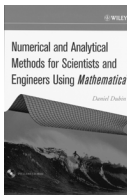
This book presents the approaches, methods, tools, and algorithms for modelling the chemical dynamics of metabolic pathways. Included are an introduction to the basic procedures for representing metabolic systems as arrays of differential equations as well as an explanation of the necessary *Mathematica* procedures for numerically solving these differential equations. *Mathematica* is also used to dynamically simulate simple, intermediate, and complex systems in enzyme kinetics. The book enables its readers to formulate their own models of time-dependent metabolic systems and aids them in the quest for relevant discoveries that remain to be made.

Chapters

Introduction to Chemical Kinetics and Numerical Integration | Elements of Enzyme Kinetics | Basic Procedures for Simulating Metabolic Systems | Advanced Simulation of Metabolic Pathways | Metabolic Control Analysis | Parameter Estimation | Model of Erythrocyte Metabolism | Metabolic Control Analysis of Human Erythrocyte Metabolism | Appendices

Available:

store.wolfram.com/view/book/ISBN0849314682.str



Numerical and Analytical Methods for Scientists and Engineers Using *Mathematica*

Daniel Dubin, 2003, John Wiley & Sons, 1633 pp., hardcover, includes CD-ROM

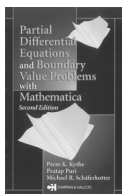
This text is designed for advanced undergraduate and graduate students in the physical and engineering sciences as well as for professionals. It offers a comprehensive *Mathematica*-based guide to the analytical and numerical methods used every day by working scientists and engineers and includes many exercises and worked examples. The entire book is available electronically on the accompanying CD as *Mathematica* notebooks.

Chapters

Ordinary Differential Equations in the Physical Sciences | Fourier Series and Transforms | Introduction to Linear Partial Differential Equations | Eigenmode Analysis | Partial Differential Equations in Infinite Domains | Numerical Solution of Linear Partial Differential Equations | Nonlinear Partial Differential Equations | Introduction to Random Processes | An Introduction to *Mathematica* (electronic version only) | Appendix: Finite-Differenced Derivatives

Available:

store.wolfram.com/view/book/ISBN0471266108.str



Partial Differential Equations and Boundary Value Problems with *Mathematica*, Second Edition

Prem K. Kythe, Pratap Puri, and Michael R. Schäferkötter,
2003, Chapman & Hall/CRC, 418 pp., hardcover

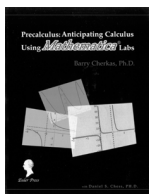
This revised and updated second edition of the book *Partial Differential Equations and Mathematica* emphasizes solution methods and includes additional exercises, problems, and topics. The extensive changes make the text more accessible, thorough, and practical. It is suitable for any mathematics curriculum and is designed to meet the needs of mathematics, science, and engineering students. Readers will encounter partial differential equations and initial and boundary value problems in a variety of applications from fields that include continuum mechanics, potential theory, geophysics, physics, biology, and mathematical economics.

Chapters

Introduction to *Mathematica* | Introduction | Method of Characteristics | Linear Equations with Constant Coefficients | Orthogonal Expansions | Separation of Variables | Integral Transforms | Green's Functions | Initial and Boundary Value Problems | Weighted Residual Methods | Perturbation Methods | Finite Difference Methods | Appendices | Bibliography

Available:

store.wolfram.com/view/book/ISBN1584883146.str



Precalculus: Anticipating Calculus Using *Mathematica* Labs

Barry Cherkas with Daniel S. Chess, 2002, Euler Press,
162 pp., paperback

This laboratory manual/textbook complements the topics found in a traditional precalculus course by introducing topics that are generally bypassed in these courses but are effective in anticipating calculus. Topics—such as finding the turning points for polynomials of any degree, finding equations of tangent lines to polynomials, and solving elementary optimization problems—that are generally reserved for treatment in calculus are explored from a precalculus viewpoint. The textbook presumes access to *Mathematica* and introduces students to a set of *Mathematica* commands. The **Graphing** add-on package used with the textbook includes several commands to create mathematically enhanced graphs for functions with singularities and helps clarify for students the location and behavior of functions at points of singularity.

Chapters

An Introduction to *Mathematica* | Solving Linear Equations and Graphing Inequalities in 1D | Solving Polynomial Equations and Inequalities in 1D | Plotting Straight Lines and Circles and Defining Functions | Lines, Parabolas, and Cubics | Plotting and Graphing in 2D | Further Plotting and Graphing in 2D | Finding Turning Points of Polynomials; Optimization Problems | Tangent and Normal Lines to Polynomial Curves | Graphing Rational Functions | Graphing Exponential and Logarithmic Functions | Graphing Basic Trigonometric Functions | Behind the Scenes of the **Plot** Command | Plotting in the Complex Plane | Index | Appendix

Available:

store.wolfram.com/view/book/ISBN0971634718.str

Tricks of the Trade

Edited by Paul Abbott

This is a column of programming tricks and techniques, most of which, we hope, will be contributed by our readers.

■ TraditionalForm versus StandardForm

The default input and output **FormatType** is **StandardForm**, which provides good readable two-dimensional typeset input and output and is mathematically unambiguous because it uses *Mathematica* syntax. **TraditionalForm** is an enhanced format corresponding to mathematical syntax, as far as this is consistently possible. Nevertheless, most *Mathematica* users cling stubbornly to **StandardForm**.

Since I find **TraditionalForm** to be the most attractive input and output format—and I use it in my columns—I would like to convince readers of the advantages of using **TraditionalForm**. These advantages include:

1. Matrices can be entered and are displayed in two-dimensional form without requiring **MatrixForm**.

$$\text{In}[1]:= \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

$$\text{Out}[1]= \begin{pmatrix} a^2 + b d & a b + e b & a c + b f \\ d^2 + a c & b c + d e & c^2 + d f \end{pmatrix}$$

2. Derivatives can be entered in their familiar two-dimensional form.

$$\text{In}[2]:= \frac{d g(u, v)}{d x}$$

$$\text{Out}[2]= \frac{d v}{d x} g^{(0,1)}(u, v) + \frac{d u}{d x} g^{(1,0)}(u, v)$$

3. Standard notation is available for limits and directional limits.

$$\text{In}[3]:= \lim_{x \rightarrow 0^-} \frac{\sin(x)}{|x|}$$

$$\text{Out}[3]= -1$$

4. Mathematical expressions are easier to read.

$$\text{In}[4]:= \int_0^{\frac{\pi}{2}} \frac{1}{\sqrt{1-m \sin^2(\theta)}} d\theta$$

$$\text{Out}[4]= K(m)$$

5. Where it is not ambiguous, special functions are interpreted automatically. For example, here is the derivative of the Bessel function $\mathcal{J}_n(z)$.

$$\text{In}[5]:= \frac{\partial \mathcal{J}_n(z)}{\partial z}$$

$$\text{Out}[5]= \frac{1}{2} (\mathcal{J}_{n-1}(z) - \mathcal{J}_{n+1}(z))$$

6. Special function names and syntax correspond to those found in standard tables.

$$\text{In}[6]:= {}_2F_1(1, 1; 2; x)$$

$$\text{Out}[6]= -\frac{\log(1-x)}{x}$$

7. Complicated functions are easier to digest.

$$\text{In}[7]= \text{MeijerG}[\{\{a\}, \{b, c\}\}, \{\{d, e\}, \{f, g, b\}\}, z]$$

$$\text{Out}[7]= G_{3,5}^{2,1}\left(z \left| \begin{array}{c} a, b, c \\ d, e, f, g, b \end{array} \right. \right)$$

TraditionalForm can be selected as the default input and output format using Cell ▸ Default Input FormatType and Cell ▸ Default Output FormatType.

There are, of course, issues involved in using **TraditionalForm**.

1. Because a space is (correctly) interpreted as multiplication, entering a space where it is not required leads to the following type of mistake.

$$\text{In}[8]:= \int \sin(x) dx$$

General::spell1 :

Possible spelling error: new symbol name "sin" is similar to existing symbol "Sin".

$$\text{Out}[8]= \frac{\sin x^2}{2}$$

2. It may not always be obvious how to input typeset expressions in **TraditionalForm**. Probably the easiest way is to type the expression using normal syntax, for example,

$$\text{LegendreP}[l, m, x]$$

and then use Convert To ▸ TraditionalForm (under the Cell menu) on the cell to yield

$$P_l^m(x)$$

Note that this expression is not just a P with a subscript l and superscript m . Unformatting the expression (using Show Expression under the Format menu) reveals **LegendreP** in a **Tagbox** at its heart.

```
Cell[BoxData[
  FormBox[
    RowBox[{
      SubsuperscriptBox[
        TagBox["P",
          LegendreP], "l", "m"], "(", "x", ")"]],
    TraditionalForm]], "InputOnly"]
```

Finally, it is possible to include a palette that automatically switches all input and output cells in a notebook from **StandardForm** to **TraditionalForm** and vice versa.

Convert to StandardForm
Convert to TraditionalForm

To produce a palette, select the above cell and then use Generate Palette from Selection (under the File menu).

□ Notation Package

It is also worthwhile to highlight the excellent **Utilities`Notation`** package (the latest version of which is available from library.wolfram.com/packages/notation). This package allows you to define arbitrary input and output syntaxes for functions of your choosing.

Consider defining a notation for mappings. For example, the function $f(x) = x^2$ may be written as the mapping $f : x \mapsto x^2$. The mapping notation corresponds directly to **Function** in *Mathematica*.

Load the *Notation* package.

```
In[9]:= << Utilities`Notation`;
```

Use the Notation palette to define a *two-way* notation (indicated by $\iff$) for mappings.

```
In[10]:= Notation[f_ : x_ \mapsto y_ \iff f_ := Function[x_, y_]]
```

To make it easier to enter mappings, use the Notation palette to add an input alias for mappings to the current notebook.

```
In[11]:= AddInputAlias[■ : □ \mapsto □, "map"]
```

Now enter “map” templates by simply typing `ESCmapESC`.

```
■ : □ \mapsto □
```

Using the `TAB` key you can cycle through the placeholders in this template. Mappings can now be entered using ordinary mathematical notation. For example, here is the function $f(x) = x^2$.

In[12]:= $f : x \mapsto x^2$

In[13]:= $f(y \cos(y))$

Out[13]= $y^2 \cos^2(y)$

Also, because we have used a *two-way* notation, conversion to and from ordinary mathematical notation and *Mathematica* notation is possible. For example, define the function $g(x) = x \cos(x^2)$ using **Function**.

$g := \text{Function}[x, x \text{Cos}[x^2]]$

Then use Convert To ▸ TraditionalForm (under the Cell menu) on the cell.

$g : x \mapsto x \cos(x^2)$

You can recover the original **Function** form using Convert To ▸ StandardForm.

$g := \text{Function}[x, x \text{Cos}[x^2]]$

■ Using CylindricalAlgebraicDecomposition for Optimization

Frank J. Kampas

fkampas@msn.com

The functions **ConstrainedMin**, **ConstrainedMax**, and **LinearProgramming** can find a minimum or maximum of a linear objective function subject to linear constraints. The function **Experimental`Minimize** can find a minimum of a rational polynomial function subject to rational polynomial constraints, and **NumericalMath`NMinimize** and **NumericalMath`NMaximize** can find a minimum or maximum of more general functions subject to more general constraints. None of these functions can find all the minima or maxima of an optimization problem. However, **Experimental`CylindricalAlgebraicDecomposition`** can find all the minima and maxima of rational polynomial objective functions subject to rational polynomial constraints, although the time required may become prohibitively long for more than simple cases.

First, consider the trivial example of maximizing $x + y$ subject to $x + y \leq 1$, $x \geq 0$, and $y \geq 0$. **ConstrainedMax** gives one solution. (Note that the linear programming functions **ConstrainedMin** and **ConstrainedMax** assume that all variables are greater than or equal to 0. Although **LinearProgramming** assumes nonnegativity by default, it can work with negative variables in Version 4.2.)

```
In[1]:= ConstrainedMax[x + y, {x + y ≤ 1}, {x, y}]
```

```
Out[1]= {1, {x → 1, y → 0}}
```

Reformulating this problem as a set of equalities and inequalities and using **CylindricalAlgebraicDecomposition** gives all the maxima as the line $y = 1 - x$ for $0 \leq x \leq 1$ when the objective function has its maximum value of 1.

```
In[2]:= << Experimental
```

```
In[3]:= CylindricalAlgebraicDecomposition(
      {obj == x + y, x + y ≤ 1, x ≥ 0, y ≥ 0}, {obj, x, y})
```

```
Out[3]= obj == 0 ∧ x == 0 ∧ y == 0 ∨ 0 < obj ≤ 1 ∧ 0 ≤ x ≤ obj ∧ y == obj - x
```

```
In[4]:= % /. obj → 1
```

```
Out[4]= 0 ≤ x ≤ 1 ∧ y == 1 - x
```

Note that putting the objective function first in the list of variables causes the results to be displayed in order of increasing values of the objective function.

Next consider the nonlinear problem of minimizing xy subject to $x^2 + y^2 \leq 1$. **Minimize** gives one minimum whereas it is clear from symmetry ($x \leftrightarrow y$) that there should be two.

```
In[5]:= Timing[Minimize(x y, {x^2 + y^2 ≤ 1}, {x, y})]
```

```
Out[5]= {0.2 Second, {-1/2, {y → -1/√2, x → 1/√2}}}
```

CylindricalAlgebraicDecomposition provides the two minima and the two maxima as well as a lot of information (here suppressed) about the function between the minima and the maxima, at the price of taking considerably longer than **Minimize**.

```
In[6]:= Timing[CAD = CylindricalAlgebraicDecomposition[
      {obj == x y, x^2 + y^2 ≤ 1}, {obj, x, y}];] // First
```

```
Out[6]= 0.883333 Second
```

```
In[7]:= First[CAD]
```

```
Out[7]= obj == -1/2 ∧ (x == -1/√2 ∧ y == 1/√2 ∨ x == 1/√2 ∧ y == -1/√2)
```

This problem can be overcome by first using **Minimize** to find the minimum value of the objective function and then using **CylindricalAlgebraicDecomposition** to find all the solutions for that value of the objective function. The function **multipleMinimize** implements this approach.

```
In[8]:= multipleMinimize[obj_, cons_, vars_] :=
  Block[{soln = Minimize[obj, cons, vars], objfunc},
    CylindricalAlgebraicDecomposition[Join[{objfunc == obj == soln[[1]],
      cons], Join[{objfunc}, vars]] /. objfunc -> obj]
```

```
In[9]:= Timing[multipleMinimize(x y, {x^2 + y^2 ≤ 1}, {x, y})]
```

```
Out[9]= {0.15 Second,
```

$$x y == -\frac{1}{2} \bigwedge \left(x == -\frac{1}{\sqrt{2}} \bigwedge y == \frac{1}{\sqrt{2}} \vee x == \frac{1}{\sqrt{2}} \bigwedge y == -\frac{1}{\sqrt{2}} \right)$$

Note that **ExperimentalInfimum** is likely to be a little faster than **ExperimentalMinimize** when all constraints are strict inequalities.

multipleMinimize also works for linear programming problems.

```
In[10]:= Timing[multipleMinimize(-(x + y), {x + y ≤ 1, x ≥ 0, y ≥ 0}, {x, y})]
```

```
Out[10]= {0.0333333 Second, -x - y == -1 ∧ 0 ≤ x ≤ 1 ∧ y == 1 - x}
```

Finally, here is an example of finding a parametrized minimizing set.

```
In[11]:= Timing[multipleMinimize((x^2 + y^2 - 1)^2, {x + y ≤ 1}, {x, y})]
```

```
Out[11]= {0.133333 Second, (x^2 + y^2 - 1)^2 == 0 ∧
  (x == -1 ∧ y == 0 ∨ -1 < x ≤ 0 ∧ (y == -√(1 - x^2) ∨ y == √(1 - x^2)) ∨
  0 < x ≤ 1 ∧ y == -√(1 - x^2))}
```

■ Anagrams

Roger Germundsson

roger@wolfram.com

It is straightforward to produce all the anagrams of a given string.

```
In[1]:= Anagrams[s_] :=
  Union[StringJoin/@ Permutations[Characters[ToLowerCase[s]]]];
```

```
In[2]:= Anagrams["Brad"]
```

```
Out[2]= {abdr, abrd, adbr, adrb, arbd, ardb, badr, bard, bdar, bdra, brad, brda,
  dabr, darb, dbar, dbra, drab, drba, rabd, radb, rbad, rbda, rdab, rdba}
```

The Version 4 notebook interface includes spell checking and hyphenation (see www.wolfram.com/products/mathematica/history.html). You can call on the extensive dictionary, using **NotebookGetMisspellingsPacket**, to filter human language anagrams.

```
In[3]:= GetMisspelledWords[text_, language_ : "English"] :=
Module[{nb, result}, nb = NotebookPut[Notebook[
  {Cell[text, "Text", LanguageCategory → NaturalLanguage,
    DefaultNaturalLanguage → language]}, Visible → False]];
LinkWrite[$ParentLink, NotebookGetMisspellingsPacket[nb]];
result = LinkRead[$ParentLink];
NotebookClose[nb, Interactive → False]; result]
```

Because `NotebookGetMisspellingsPacket` works with notebooks, the text to be filtered is put into an invisible notebook, which is closed after the spelling errors have been determined. Now we can produce all the human language anagrams of a given string.

```
In[4]:= CorrectWords[wl_List, language_ : "English"] := Complement[wl,
  GetMisspelledWords[ToString[Infix[wl, " "], language]];

In[5]:= HumanAnagrams[s_, l_ : "English"] := CorrectWords[Anagrams[s], l];

In[6]:= HumanAnagrams["Brad"]

Out[6]= {bard, brad, drab}
```

International dictionaries, available at store.wolfram.com/view/app/dictionaries, allow one to produce human language anagrams in a range of languages.

■ Apropos

Lou D'Andria

lou@wolfram.com

Sometimes searching by name is not enough.

```
In[1]:= Names["*divisor*", IgnoreCase → True]

Out[1]= {Divisors, DivisorSigma}
```

Related functions such as `GCD` are missed by such a search. **Apropos** (Oxford English Dictionary: *adj.* having direct reference to the matter in hand) finds all directly relevant functions by searching through usage messages.

```
In[2]:= Apropos[str_String] := Apropos[str, $ContextPath]

In[3]:= Apropos[str_String, {cnt__String}] :=
  Select[Flatten[(Names[#1 <> "*" ] &) /@ {cnt}],
    (StringQ[#1] && StringMatchQ[#1, "*" <> str <> "*",
      IgnoreCase → True] &) [ToExpression[#1 <> "::usage"]]] &];

In[4]:= Apropos["divisor"]

Out[4]= {Divisors, DivisorSigma, ExtendedGCD, GCD, PolynomialGCD}
```

As a second example, we use **Apropos** to find all search functions in the *Combinatorica* package.

```

In[5]:= << "DiscreteMath`Combinatorica`"

In[6]:= Apropos["search", {"DiscreteMath`Combinatorica`"}]

Out[6]:= {Backtrack, BinarySearch, BreadthFirstTraversal, VertexColoring}

```

■ *J/Link* and LiveGraphics3D

Dale Horton

daleh@wolfram.com

With *J/Link* you can invoke Java applets directly from *Mathematica*. We load *J/Link* and install.

```

In[1]:= << JLink` ;
        InstallJava[];

```

A particularly nice applet for interactive visualization is the LiveGraphics3D applet. First, download the applet from wwwvis.informatik.uni-stuttgart.de/~kraus/LiveGraphics3D/download.html. Then, put it on the CLASSPATH for *J/Link*. The directory given by the following evaluation is recommended.

```

In[3]:= ToFileName[{UserAddOnsDirectory, "Applications", "Live3D", "Java"}]

Out[3]:= /Users/paul/Library/Mathematica/Applications/Live3D/Java/

```

This directory is on the Java class path.

```

In[4]:= JavaClassPath[]

Out[4]:= {/Applications/Mathematica 4.2.app/SystemFiles/Java ,
          /Users/paul/Library/Mathematica/Applications/Live3D/Java/,
          /Users/paul/Library/Mathematica/Applications/Live3D/Java/live.jar ,
          /Applications/Mathematica 4.2.app/SystemFiles/Java/Darwin/lib/tools.jar}

```

The following code exports 3D graphics to the LiveGraphics3D applet.

```

In[5]:= Attributes[ShowLive] = {HoldAll};

In[6]:= ShowLive[gr_] :=
        Block[{$DisplayFunction = Identity}, InstallJava[]; showlive[gr]]

In[7]:= showlive[gr_Graphics3D] :=
        AppletViewer["Live", {"INPUT=" <> ToString[InputForm[N[gr]]]]]

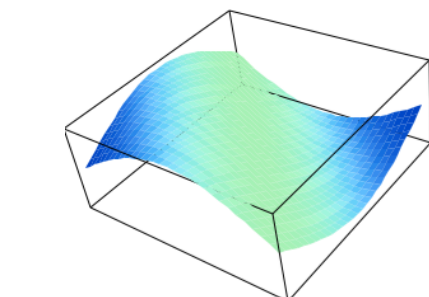
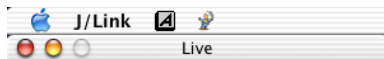
In[8]:= showlive[gr_SurfaceGraphics] := showlive[Graphics3D[gr]]

In[9]:= showlive[_] := $Failed

```

Entering the following command opens an interactive visualization window under *J/Link*.

```
In[10]:= ShowLive[Plot3D[sin(x + y), {x, 0,  $\pi$ },  
{y, 0,  $\pi$ }, PlotPoints  $\rightarrow$  30, Mesh  $\rightarrow$  False, Axes  $\rightarrow$  False]]
```

**Paul Abbott**

Department of Physics
University of Western Australia
35 Stirling Highway
Crawley WA 6009, Australia
tmj@physics.uwa.edu.au
physics.uwa.edu.au/~paul

In and Out

Edited by Paul Abbott

In and Out offers readers an opportunity to ask questions of the experts. *The Mathematica Journal* encourages readers to submit problems in care of the editor.

■ Locus of Intersection

Q: Consider the circle $(x-1)^2 + (y-1)^2 = r^2$ and the parabola $y^2 = x$. First, find the intersection points of the circle and the parabola. Second, compute the chord of the circle through the two intersection points. Third, compute the line segment through $\{1, 1\}$ perpendicular to the chord at point $P = \{x, y\}$. Then, find the locus of the point P . And, finally, animate the process as a function of r . How can I do this?

A: Computing the intersection points of the circle and the parabola as a function of r is immediate. The result is rather complicated so we suppress the output.

```
In[1]:= eqns[r_] = {(x - 1)^2 + (y - 1)^2 == r^2, y^2 == x};
```

```
In[2]:= intersections[r_] = Solve[eqns[r], {x, y}];
```

Next, we compute the chord, which is the line segment joining the two *real* intersection points, using pattern matching.

```
In[3]:= chord[r_] := chord[r] = Line[Cases[N[intersections[r]],  
    {x -> a_ /; a ∈ ℝ, y -> b_ /; b ∈ ℝ} -> {a, b}, ∞]]
```

To compute the line through the point $\{1, 1\}$ perpendicular to the chord at point $P = \{x, y\}$, we use the **ExtendGraphics`Geometry`** package by Tom Wickham-Jones, available from *MathSource* (library.wolfram.com/database/Books/3753). After putting the **ExtendGraphics`** folder into the **AddOns`Applications`** directory (or, alternatively, in the directory returned by evaluating **\$UserAdd`OnsDirectory**), we can load it in the usual fashion.

```
In[4]:= << ExtendGraphics`Geometry`
```

After converting the chord to an **ImplicitLine** object (this implicit parametrization is the loci of points p that satisfy the equation $(p - c) \cdot \vec{n} = 0$ where c is a

point on the line and $\vec{n}$ is the normal to the line), we can determine the point $P = \{x, y\}$ as a function of r using **ClosestPointOnLine**.

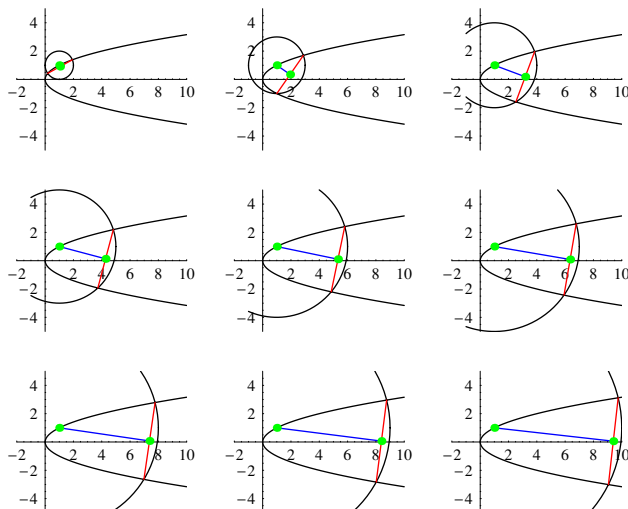
```
In[5]:= P[r_] := P[r] = ClosestPointOnLine[{1, 1}, ToImplicit[chord[r]]]
```

We are now in a position to show the chord, perpendicular, and locus of the point P as a function of r .

```
In[6]:= << Graphics`;
```

```
Table[ImplicitPlot{eqns[r], {x, -1, 10}, PlotRange -> { -2 10, -5 5},
  Epilog -> {Hue[1], chord[r], Hue[2/3], Line[{(1, 1), P[r]}], Hue[1/3],
    AbsolutePointSize[3], Point[{(1, 1)], Point[P[r]]}], {r, 9}];
```

From In[6]:=



Finding the *exact* locus of the point P is more complicated. Because points on the parabola have coordinates $\{y^2, y\}$, computing the locus of the point P as a function of y_1 and y_2 —the two intersection points—is straightforward.

```
In[8]:= locus =
```

$$\text{FullSimplify}\left[\text{ClosestPointOnLine}\left(\{1, 1\}, \text{ToImplicit}\left(\text{Line}\left[\begin{pmatrix} y_1^2 & y_1 \\ y_2^2 & y_2 \end{pmatrix}\right]\right)\right)\right]$$

$$\text{Out[8]} = \left\{ \frac{y_1^2 + (y_2 + 1)y_1 + y_2^2 + y_2}{(y_1 + y_2)^2 + 1}, \frac{y_2 + y_1(y_2(y_1 + y_2) + 1) + 1}{(y_1 + y_2)^2 + 1} \right\}$$

Returning to the original equations, after eliminating x , the *real* values of y at the two intersection points can be determined using **CylindricalAlgebraicDecomposition**.

```
In[9]:= Eliminate[eqns[r], x]
```

```
Out[9]= y4 - y2 - 2 y + 2 == r2
```

```
In[10]:= ExperimentalCylindricalAlgebraicDecomposition[% & r > 0, {r, y}]
```

```
Out[10]= r > 0 ∧ (y == Root[#14 - #12 - 2 #1 - r2 + 2 &, 1] ∨  
y == Root[#14 - #12 - 2 #1 - r2 + 2 &, 2])
```

Hence, we obtain an explicit expression for the locus of P .

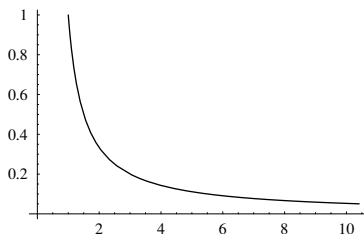
```
In[11]:= locus /. y_i_ :> Root[#14 - #12 - 2 #1 - r2 + 2 &, i] // FullSimplify
```

```
Out[11]= {{(Root[#14 - #12 - 2 #1 - r2 + 2 &, 1]2 +  
(Root[#14 - #12 - 2 #1 - r2 + 2 &, 2] + 1) Root[#14 - #12 - 2 #1 - r2 + 2 &,  
1] + Root[#14 - #12 - 2 #1 - r2 + 2 &, 2]2 +  
Root[#14 - #12 - 2 #1 - r2 + 2 &, 2]) /  
((Root[#14 - #12 - 2 #1 - r2 + 2 &, 1] +  
Root[#14 - #12 - 2 #1 - r2 + 2 &, 2])2 + 1),  
(Root[#14 - #12 - 2 #1 - r2 + 2 &, 2] + Root[#14 - #12 - 2 #1 - r2 + 2 &, 1]  
(Root[#14 - #12 - 2 #1 - r2 + 2 &, 2] (Root[#14 - #12 - 2 #1 - r2 + 2 &,  
1] + Root[#14 - #12 - 2 #1 - r2 + 2 &, 2]) + 1) + 1) /  
((Root[#14 - #12 - 2 #1 - r2 + 2 &, 1] + Root[#14 - #12 - 2 #1 - r2 + 2 &,  
2])2 + 1)}
```

Although this expression can be simplified further (note that **RootReduce** fails here), it is sufficient for our purpose. After plotting the locus, we can display it along with the circle, parabola, chord, and perpendicular.

```
In[12]:= ParametricPlot[%, {r, 0, 10}, Compiled -> False];
```

From In[12]:=

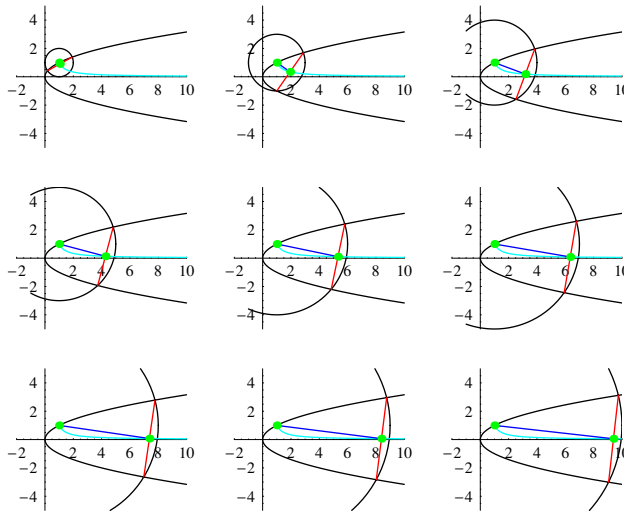


```

In[13]:= Table[ImplicitPlot[eqns[r], {x, -1, 10}, PlotRange -> {{-2, 10}, {-5, 5}}, Epilog ->
  {Hue[1], chord[r], Hue[1/2], First[%], Hue[2/3], Line[{1, 1}, P[r]],
    Hue[1/3], AbsolutePointSize[3], Point[{1, 1}], Point[P[r]]}], {r, 9}];

```

From In[13]:=



■ Pure Recursive Functions

Q: Normally, when creating a recursive function, I use the name of the function to perform the recursive call.

```

In[1]:= fact[n_] := If[n == 1, 1, n fact[n - 1]]

```

However, this has the disadvantage that the symbol **fact** is now global. Is it possible to create a pure recursive function?

A: Andrzej Kozłowski (akoz@mimuw.edu.pl) writes: **#0** is a rarely used “parameter,” which refers to the pure function itself and thus makes it possible to create pure recursive functions.

```

In[2]:= fact = Function[If[#1 == 0, 1, #1 #0[#1 - 1]]];

```

```

In[3]:= fact[5]

```

```

Out[3]= 120

```

■ Binomial Limit

Q: How can I compute the limit of $(-1)^n \binom{-1/2}{n} \sqrt{n\pi}$ as $n \rightarrow \infty$ through integer values?

A: Numerically, it appears that the limit is one.

$$\text{In}[1]:= f(n) = (-1)^n \binom{-1/2}{n} \sqrt{n\pi};$$

$$\text{In}[2]:= \text{Table}[\{10^n, f(10^n) // N\}, \{n, 0, 4\}]$$

$$\text{Out}[2]= \begin{pmatrix} 1 & 0.886227 \\ 10 & 0.987583 \\ 100 & 0.998751 \\ 1000 & 0.999875 \\ 10000 & 0.999988 \end{pmatrix}$$

Analytically, we can use **FunctionExpand** to replace the binomial by gamma functions.

$$\text{In}[3]:= \text{FunctionExpand}[f(n)]$$

$$\text{Out}[3]= \frac{(-1)^n \pi}{\sqrt{n} \Gamma(\frac{1}{2} - n) \Gamma(n)}$$

Next, we use the reflection formula, $\Gamma(z) \Gamma(1 - z) = \pi \csc(\pi z)$, here implemented via pattern matching.

$$\text{In}[4]:= \% /. \Gamma\left(\frac{1}{2} - n\right) \rightarrow \frac{\text{FullSimplify}[\Gamma(\frac{1}{2} - n) \Gamma(n + \frac{1}{2})]}{\Gamma(n + \frac{1}{2})}$$

$$\text{Out}[4]= \frac{(-1)^n \cos(n\pi) \Gamma(n + \frac{1}{2})}{\sqrt{n} \Gamma(n)}$$

This expression simplifies, noting that $n \in \mathbb{Z}$.

$$\text{In}[5]:= \text{Simplify}[\%, n \in \mathbb{Z}]$$

$$\text{Out}[5]= \frac{\Gamma(n + \frac{1}{2})}{\sqrt{n} \Gamma(n)}$$

Since we are interested in the asymptotic limit, $n \rightarrow \infty$, we use **Series** (effectively using Stirling's formula).

In[6]:= % /. $\Gamma(n) \rightarrow \text{Normal}[\text{Series}[\Gamma(n), \{n, \infty, 1\}]]$

$$\text{Out[6]} = \frac{\sqrt{\frac{1}{n}} n^{\frac{1}{2}-n} \sqrt{\frac{1}{n+\frac{1}{2}}} \left(n + \frac{1}{2}\right)^{n+\frac{1}{2}}}{\sqrt{e}}$$

Noting that $n > 0$, followed by computing the limit as $n \rightarrow \infty$, gives us the desired result.

In[7]:= **Simplify**[% , $n > 0$]

$$\text{Out[7]} = \frac{n^{-n} \left(n + \frac{1}{2}\right)^n}{\sqrt{e}}$$

In[8]:= **lim** %
 $n \rightarrow \infty$

Out[8]= 1

■ 3D Text

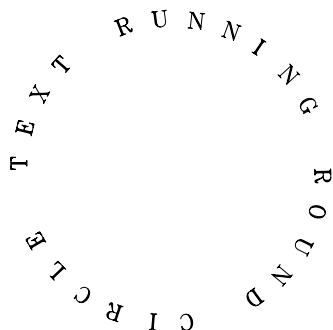
Q: Is there a way to produce 2D text for inclusion within 3D graphics?

A: See “Hershey Text in *Mathematica* Graphics” by Roland Jakschewitz at library.wolfram.com/database/Conferences/196. Using the public domain Hershey font descriptions, it is easy to produce machine-independent, scalable, and rotatable text that can then be projected onto **Graphics3D** surfaces. Here are a few examples of the more than a thousand glyphs included in the Hershey vector fonts.

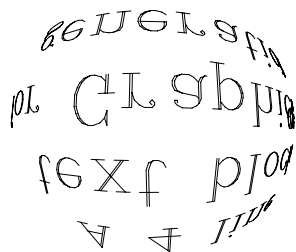


Here are two examples, taken from Jakschewitz's conference notebook.

- Text running around a circle.



- Text on the surface of a sphere.



■ Singular Values

Q: I am computing the singular values of a matrix consisting of physical data. If I have some estimate of the error in each matrix element, is there a simple way to compute the error in the singular values?

A: The singular value decomposition for a numerical matrix A consists of two matrices, U and V , and a list of singular values. Forming the diagonal matrix Σ from the singular values, the matrix decomposition reads

$$U^\dagger . \Sigma . V = A,$$

where $\dagger$ denotes the conjugate transpose. This operator can be implemented as follows.

```
In[1]:= x_† /; MatrixQ[x] := Conjugate[xT]
```

A can be complex and, in general, is not square. U and V are row orthonormal matrices, that is, $U.U^\dagger = I$ and $V.V^\dagger = I$. Hence

$$U^\dagger . \Sigma . V = A \Rightarrow U . A . V^\dagger = U . U^\dagger . \Sigma . V . V^\dagger = \Sigma .$$

In other words, U and V effectively “diagonalize” A .

Here is a random 4×5 matrix.

```
In[2]:= A = Table[Random[], {4}, {5}]
```

$$\text{Out[2]} = \begin{pmatrix} 0.621626 & 0.986186 & 0.0600801 & 0.095822 & 0.199441 \\ 0.826588 & 0.201152 & 0.274642 & 0.271143 & 0.564654 \\ 0.832332 & 0.0404587 & 0.701955 & 0.764831 & 0.592724 \\ 0.941093 & 0.500966 & 0.14063 & 0.87306 & 0.333728 \end{pmatrix}$$

Compute the singular values, σ , and the matrices U , V , and Σ .

```
In[3]:= {U, \sigma, V} = SingularValues[A]; \Sigma = DiagonalMatrix[\sigma]
```

$$\text{Out[3]} = \begin{pmatrix} 2.35829 & 0 & 0 & 0 \\ 0 & 0.954824 & 0 & 0 \\ 0 & 0 & 0.460482 & 0 \\ 0 & 0 & 0 & 0.266865 \end{pmatrix}$$

Since we are dealing with numerical matrices, to check whether $U^\dagger . \Sigma . V = A$, we use the Euclidean norm, $\|M\|$, of a matrix M , defined by $\|M\| = \sqrt{\text{Tr}(M^\dagger . M)}$ (using `DoubleBracketingBar`).

```
In[4]:= ||x_?MatrixQ|| := Sqrt[Tr[x^\dagger . x]]
```

```
In[5]:= ||U^\dagger . \Sigma . V - A||
```

```
Out[5]= 1.53323 \times 10^{-15}
```

Similarly, we confirm that $U . A . V^\dagger = \Sigma$.

```
In[6]:= ||U . A . V^\dagger - \Sigma||
```

```
Out[6]= 1.88369 \times 10^{-15}
```

It is easy to obtain an expression for the sensitivity of the k th singular value, σ_k , to small changes in the element $a_{i,j}$. Since

$$U . A . V^\dagger = \Sigma \Leftrightarrow u_k . A . v_k^* = \sum_{i,j} u_{k,i} a_{i,j} v_{k,j}^* = \sigma_k,$$

we have that

$$\frac{\partial \sigma_k}{\partial a_{i,j}} = u_{k,i} v_{k,j}^* .$$

Hence, the total change in σ_k , due to changes in any of the elements of A , is

$$\Delta\sigma_k = \sum_{i,j} \Delta a_{i,j} \frac{\partial \sigma_k}{\partial a_{i,j}} = \sum_{i,j} u_{k,i} \Delta a_{i,j} v_{k,j}^* = \{U.\Delta A.V^\dagger\}_{k,k}.$$

This result is not surprising. We can obtain it directly as follows:

$$U.(A + \Delta A).V^\dagger = \Sigma + \Delta\Sigma \Rightarrow U.\Delta A.V^\dagger = \Delta\Sigma.$$

For example, form the matrix B by perturbing $a_{2,3}$ by 10^{-3} .

```
In[7]:= B = A; B[[2, 3]] += 10-3
```

```
Out[7]= 0.275642
```

The change in the singular values, $\Delta\sigma_k$, is given by the diagonal entries of $U.\Delta A.V^\dagger$.

```
In[8]:= ΔA = B - A
```

```
Out[8]=  $\begin{pmatrix} 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0.001 & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & 0. \end{pmatrix}$ 
```

```
In[9]:= Δσ = Tr[U.ΔA.V†, List]
```

```
Out[9]= {0.000116798, 0.0000325029, 0.000226689, -0.000510452}
```

These values should be compared with those obtained by computing the difference of the singular values of B and A .

```
In[10]:= SingularValues[B][[2]] - σ
```

```
Out[10]= {0.00011685, 0.0000325704, 0.000226893, -0.000510246}
```

The agreement is very good.

Using this approach, we can also compute the effect of an arbitrary *symbolic* perturbation.

```
In[11]:= B = A; B[[3, 1]] += ε;
```

```
In[12]:= ΔA = B - A;
```

```
In[13]:= Tr[U.ΔA.V†, List] // Chop
```

```
Out[13]= {0.391573 ε, -0.0310559 ε, 0.0196619 ε, -0.264224 ε}
```

Paul Abbott

Department of Physics
University of Western Australia
35 Stirling Highway
Crawley WA 6009, Australia
tmj@physics.uwa.edu.au
physics.uwa.edu.au/~paul

The Functions Website

Michael Trott

In this issue's Corner I want to take a break from making Riemann surface pictures and talk about some of my regular work. Over the last four years I have, together with my coworkers Oleg Marichev, Andy Hunt, and others, built the website functions.wolfram.com.

■ What Is the Website About?

The Functions website is a comprehensive digital library containing a variety of material related to elementary, special, and number theoretical functions, distributions, and other mathematical functions. Typical examples are π , $\sin(z)$, and ${}_pF_q(a_1, a_2, \dots, a_p; b_1, b_2, \dots, b_q; z)$. Currently the website has more than 54,000 identities, inequalities, integrals, sums, differential equations, and so on, all accessible from a web browser and downloadable as PDF files or notebooks. All the formulas are available in *Mathematica* input form, standard form, and MathML.

■ Why Care about Mathematical Functions?

Special functions occur in exact models and approximate solutions of virtually thousands of problems in mathematics and the exact sciences. Typical examples are the eigenfunctions of a quantum mechanical harmonic oscillator that contain Hermite polynomials, or Bessel functions that appear in the solution of Newton's equations of motion for a chain of masses connected by springs. There are so many examples that I do not even attempt to give references at this point. The fact that these functions were given a name indicates their role in the exact sciences better than anything else.

Mathematica users are surely used to a variety of special functions returned by functions such as **Integrate**, **Sum**, and **DSolve**. Here are some typical examples.

$$\begin{aligned} \text{In}[1]:= & \int \frac{\text{Cos}[x + \alpha] x}{x^\beta} dx \\ \text{Out}[1]= & \frac{1}{2} \text{Cos}[\alpha] \\ & (-(-i x)^{-2+\beta} x^{2-\beta} \text{Gamma}[2-\beta, -i x] - (i x)^{-2+\beta} x^{2-\beta} \text{Gamma}[2-\beta, i x]) + \frac{1}{2} i \\ & (-(-i x)^{-2+\beta} x^{2-\beta} \text{Gamma}[2-\beta, -i x] + (i x)^{-2+\beta} x^{2-\beta} \text{Gamma}[2-\beta, i x]) \text{Sin}[\alpha] \end{aligned}$$

$$\text{In}[2]:= \int \text{ArcCosh}[\text{Sec}[\text{ArcSinh}[x]]] \, dx$$

$$\begin{aligned} \text{Out}[2]= & x \text{ArcCosh}[\text{Sec}[\text{ArcSinh}[x]]] - \\ & \left(\frac{1}{2} - \frac{i}{2}\right) e^{(1+i) \text{ArcSinh}[x]} \sqrt{\frac{1 - \text{Cos}[\text{ArcSinh}[x]]}{1 + \text{Cos}[\text{ArcSinh}[x]]}} \text{Cot}\left[\frac{\text{ArcSinh}[x]}{2}\right] \\ & \text{Hypergeometric2F1}\left[\frac{1}{2} - \frac{i}{2}, 1, \frac{3}{2} - \frac{i}{2}, -e^{2i \text{ArcSinh}[x]}\right] - \\ & \left(\frac{1}{2} + \frac{i}{2}\right) e^{(-1+i) \text{ArcSinh}[x]} \sqrt{\frac{1 - \text{Cos}[\text{ArcSinh}[x]]}{1 + \text{Cos}[\text{ArcSinh}[x]]}} \text{Cot}\left[\frac{\text{ArcSinh}[x]}{2}\right] \\ & \text{Hypergeometric2F1}\left[\frac{1}{2} + \frac{i}{2}, 1, \frac{3}{2} + \frac{i}{2}, -e^{2i \text{ArcSinh}[x]}\right] \end{aligned}$$

$$\text{In}[3]:= \sum_{k=1}^n \frac{(-1)^k}{\alpha + \beta k}$$

$$\begin{aligned} \text{Out}[3]= & \frac{\text{PolyGamma}\left[0, \frac{1}{2} + \frac{\alpha}{2\beta}\right]}{2\beta} - \frac{\text{PolyGamma}\left[0, 1 + \frac{\alpha}{2\beta}\right]}{2\beta} - \\ & \frac{(-1)^n \text{PolyGamma}\left[0, \frac{1}{2} + \frac{n}{2} + \frac{\alpha}{2\beta}\right]}{2\beta} + \frac{(-1)^n \text{PolyGamma}\left[0, 1 + \frac{n}{2} + \frac{\alpha}{2\beta}\right]}{2\beta} \end{aligned}$$

$$\text{In}[4]:= \sum_{k=1}^{\infty} \frac{(k - \alpha)^2}{(k + 1)^5}$$

$$\text{Out}[4]= \frac{1}{45} (-\pi^4 - \pi^4 \alpha - 45 \alpha^2 + 45 \text{Zeta}[3] + 45 \text{Zeta}[5] + 90 \alpha \text{Zeta}[5] + 45 \alpha^2 \text{Zeta}[5])$$

$$\text{In}[5]:= \text{DSolve}[\{\mathbf{w}'[z] == e^{\alpha z^2}, \mathbf{w}[0] == 0\}, \mathbf{w}, z]$$

$$\text{Out}[5]= \left\{\left\{w \rightarrow \text{Function}[z, \frac{\sqrt{\pi} \text{Erfi}[z \sqrt{\alpha}]}{2 \sqrt{\alpha}}]\right\}\right\}$$

□ Why Make Such a Website at All?

While *Mathematica* is able to perform a variety of operations on special functions, it does not completely replace special functions textbooks and reference books. Here is a list of the main reasons why such a website is still needed at the beginning of the twenty-first century.

- The classic handbooks about special functions (Abramowitz/Stegun, Bateman/Erdelyi) are slightly out of date. Many new results have been found for many special functions since their publication in 1964 and 1950–54 respectively.
- Not all functions used in today's research are covered in these classic handbooks (e.g., the polylogarithm and Meijer *G*-functions).

- There is no longer any need for tables of numerical values due to the built-in, high-precision numeric capabilities of modern software for all complex values of all special functions.
- The role of special functions in modern science is continually increasing. For instance, the number of mathematics, physics, and engineering articles dealing with special functions increases by about eight percent per year.
- Although the choice of branch cuts is largely arbitrary, in practice, people have adopted a small set of conventions. In computer mathematics all branch cuts of all functions follow uniquely from the branch cut of log or power. To maintain consistency, some classical formulas for special functions must be rewritten.
- Collecting this material is necessary because we still lack a universal set of algorithms to generate it. Encouraging progress has been made, for instance, in algebraic summation and the integration of elementary functions.
- Many important series can only be described by programs, for which no closed form for the n th term exists (e.g., zeros of Airy functions).
- The material is not available in a computer-readable form.
- The classical handbooks are not uniform in style and organization. (Their various chapters were compiled by different authors with different criteria for inclusion of material.) Clearly, the ideal form would be a giant table.

Function	Integral Representations	Sum Representations	Differential Equations	...	Nevanlinna Properties	Uses	Visualizations
$\sin(z)$	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...
Meijer G	...	...	...	...	...	...	...

■ Why Make Such a Website Here at Wolfram Research?

There are a variety of reasons why it was quite natural to start the project of compiling all the known properties of mathematical functions here at Wolfram Research.

- *Mathematica* has implemented more special functions than any other software available. This means a large body of knowledge and experience about many special functions already existed here at Wolfram Research.
- The fact that *Mathematica* notebooks are computer-readable documents allowed for a uniquely suitable environment to collect and verify mathematical data and to transform the resulting collections to a variety of formats, such as web pages and PDF files.

- Since the mid-1990s various efforts to collect identities about special functions have been undertaken. So, for the International Congress of Mathematicians in Berlin, we made prototypes of a large five-panel poster about special functions with more than 2000 formulas and 300 graphics. Unfortunately, with current printing technology, producing the set is prohibitively expensive. But with the advance of SVG and other even more advanced web graphic formats, we will add similar charts describing the hierarchies of various families of mathematical functions to our website.
- With the *MathWorld*TM website (mathworld.wolfram.com), we have a good deal of experience on how to create, update, and present large amounts of mathematical information on the web.

■ What Information Is There?

Currently the website contains information about the approximately 250 mathematical functions of *Mathematica*. They are grouped into the following function categories. For a complete listing of the functions, visit the website.

Elementary • Constants • Bessel, Airy, Struve • Integer Functions • Polynomials • Gamma, Beta, Erf • Hypergeometric • Elliptic Integrals • Elliptic Functions • Zeta • Mathieu • Complex Components • Number Theory • Generalized Functions

As much as applicable, the available information is grouped into the following property categories.

Notations • Primary Definition • Specific Values • General Characteristics • Series Representations • Integral Representations • Analytic Continuations • Product Representations • Limit Representations • Continued Fraction Representations • Generating Functions • Group Representations • Differential Equations • Difference Equations • Transformations • Operations (∂ , f) • Integral Transforms ($\mathcal{F}$, $\mathcal{H}$, $\mathcal{L}$, $\mathcal{M}$) • Identities • Representations through More General Functions • Relations with Other Functions • Zeros • Inequalities • Theorems • Other Information • History and Applications • References

Each formula on this website is given a unique numeric identifier. As the site expands and more information is added, the identifier for a given formula will stay the same even if its position in a group changes. Consider this actual formula with identifier 01.06.27.0010.01:

01.06.27.0010.01

$$\sin(z) = (-1)^{\left[\frac{\operatorname{Re}(z)}{\pi}\right]} \sqrt{\frac{1 - \cos(2z)}{2}} \left(1 - \left(1 + (-1)^{\left[\frac{\operatorname{Re}(z)}{\pi}\right] + \left[-\frac{\operatorname{Re}(z)}{\pi}\right]}\right) \theta(-\operatorname{Im}(z))\right)$$

(This formula demonstrates the previously mentioned branch cut topic. Most classic handbooks state $\sin(z) = \pm \sqrt{(1 - \cos(2z))/2}$ but do not give a detailed description of $\pm(z)$ for complex z .)

Reading the identifier from the left, the groups of digits point to different types of information.

- 01 stands for the function category of Elementary Functions.
- 06 stands for the function **Cos** in the list of Elementary Functions.
- 27 stands for the property category Representations through Equivalent Functions.
- 0010 numbers the formula within this section.
- 01 means the modification number of this formula.

Let me give some more examples of formulas found on the website.

The next formula describes the direction-dependent asymptotics of the Airy function $\text{Bi}(z)$.

03.06.06.0014.01

$$\text{Bi}(z) \propto \frac{(-z^3)^{-5/12}}{2\sqrt{\pi}} \left((-1)^{5/12} e^{-\frac{2i}{3}\sqrt{-z^3}} \left(\frac{1}{\sqrt[3]{-1}} \sqrt[3]{-z^3} + z \right) \left(1 + O\left(\frac{1}{z^{3/2}}\right) \right) - \right. \\ \left. (-1)^{7/12} e^{\frac{2i}{3}\sqrt{-z^3}} \left(\sqrt[3]{-1} \sqrt[3]{-z^3} + z \right) \left(1 + O\left(\frac{1}{z^{3/2}}\right) \right) \right) /; (|z| \rightarrow \infty)$$

The indefinite integral of a Bessel function.

03.01.21.0001.01

$$\int \mathcal{J}_\nu(a z) dz = 2^{-\nu-1} z (a z)^\nu \Gamma\left(\frac{\nu}{2} + \frac{1}{2}\right) {}_1\tilde{F}_2\left(\frac{\nu}{2} + \frac{1}{2}; \nu+1, \frac{\nu}{2} + \frac{3}{2}; -\frac{1}{4} a^2 z^2\right)$$

A closed-form formula for the n th prime number.

13.03.06.0005.01

$$p_n = \left\lfloor 1 - \frac{1}{\log(2)} \log \left(-\frac{1}{2} \sum_{d_j | \prod_{k=1}^{n-1} p_k} \frac{\mu(d_j)}{2^{d_j} - 1} \right) \right\rfloor /; p_k \in \mathbb{P}$$

In addition to formulas and identities for mathematical functions, theorems about them and general mathematical identities are shown and are downloadable from the website.

□ What New Information Is There?

Thousands of the formulas and identities on the website are new and were derived especially for the website. With *Mathematica*'s abilities to return special functions (**Integrate**, **Sum**, **DSolve**), to simplify them (**FullSimplify**, **FunctionExpand**), and to numericize them (**N**), *Mathematica* is an ideal environment to verify identities and to derive new ones. Let me give one example here: the

addition formula for the Weierstrass elliptic function $\wp'(u+v; g_2, g_3)$. In handbooks one finds the formula

$$\begin{aligned} \wp'(u+v; g_2, g_3) = & \\ & -(\wp(u+v; g_2, g_3)(\wp'(u; g_2, g_3) - \wp'(v; g_2, g_3)) + \wp(u; g_2, g_3)\wp'(v; g_2, g_3) - \\ & \wp'(u; g_2, g_3)\wp(v; g_2, g_3))/(\wp(u; g_2, g_3) - \wp(v; g_2, g_3)). \end{aligned}$$

In this form the formula is unsatisfying because the right-hand side contains the function $\wp$ in addition to $\wp'$. To derive a formula only in $\wp'$, I supplement the given formula with the three differential equations for $\wp(u; g_2, g_3)$, $\wp(v; g_2, g_3)$, and $\wp(u+v; g_2, g_3)$ and generate four polynomial equations by clearing the denominators.

```
In[6]:= eqs = (Numerator[Together[#]] &) /@
```

$$\left\{ \wp'[u+v] + \frac{\wp[u+v](\wp'[u] - \wp'[v]) + \wp[u]\wp'[v] - \wp'[u]\wp[v]}{\wp[u] - \wp[v]}, \right. \\ \wp'[u]^2 - 4\wp[u]^3 + g_2\wp[u] + g_3, \\ \wp'[v]^2 - 4\wp[v]^3 + g_2\wp[v] + g_3, \\ \left. \wp'[u+v]^2 - 4\wp[u+v]^3 + g_2\wp[u+v] + g_3 \right\}$$

```
Out[6]= {-wp[v]wp'[u] + wp[u+v]wp'[u] + wp[u]wp'[v] - wp[u+v]wp'[v] +
```

$$\wp[u]\wp'[u+v] - \wp[v]\wp'[u+v], g_3 + g_2\wp[u] - 4\wp[u]^3 + \wp'[u]^2, \\ g_3 + g_2\wp[v] - 4\wp[v]^3 + \wp'[v]^2, g_3 + g_2\wp[u+v] - 4\wp[u+v]^3 + \wp'[u+v]^2}$$

Using **GroebnerBasis** with **EliminationOrder** allows us to eliminate the $\wp$ s after a few minutes on a fast machine.

```
In[7]:= gb = Factor[GroebnerBasis[eqs, {\wp'[u], \wp'[v], \wp'[u+v]}, \\ {\wp[u], \wp[v], \wp[u+v]}, MonomialOrder -> EliminationOrder]];
```

The result is a polynomial of total degree 18 with 670 terms.

```
In[8]:= Length/@List@@gb[[1]]
```

```
Out[8]= {2, 2, 2, 670}
```

Here are the first 25 terms of the polynomial.

```
In[9]:= Take[gb[[1, -1]], 25]
```

```
Out[9]= 4 g2^9 wp'[u]^6 - 324 g2^6 g3^2 wp'[u]^6 + 8748 g2^3 g3^4 wp'[u]^6 - 78732 g3^5 wp'[u]^6 -
```

$$12 g2^9 wp'[u]^5 wp'[v] + 972 g2^6 g3^2 wp'[u]^5 wp'[v] - 26244 g2^3 g3^4 wp'[u]^5 wp'[v] + \\ 236196 g3^5 wp'[u]^5 wp'[v] - 3 g2^9 wp'[u]^4 wp'[v]^2 - 486 g2^6 g3^2 wp'[u]^4 wp'[v]^2 + \\ 32805 g2^3 g3^4 wp'[u]^4 wp'[v]^2 - 472392 g3^5 wp'[u]^4 wp'[v]^2 - 216 g2^6 g3 wp'[u]^6 wp'[v]^2 + \\ 11664 g2^3 g3^3 wp'[u]^6 wp'[v]^2 - 157464 g3^5 wp'[u]^6 wp'[v]^2 + 26 g2^9 wp'[u]^3 wp'[v]^3 - \\ 648 g2^6 g3^2 wp'[u]^3 wp'[v]^3 - 21870 g2^3 g3^4 wp'[u]^3 wp'[v]^3 + 551124 g3^5 wp'[u]^3 wp'[v]^3 + \\ 432 g2^6 g3 wp'[u]^5 wp'[v]^3 - 23328 g2^3 g3^3 wp'[u]^5 wp'[v]^3 + 314928 g3^5 wp'[u]^5 wp'[v]^3 + \\ 12 g2^6 wp'[u]^7 wp'[v]^3 - 648 g2^3 g3^2 wp'[u]^7 wp'[v]^3 + 8748 g3^4 wp'[u]^7 wp'[v]^3$$

The double-argument specification yields another previously unknown formula.

```
In[10]:= Factor[gb[1, -1] /. v -> u] // Simplify
```

$$\begin{aligned} \text{Out[10]} = & 4 (\wp' [u] + \wp' [2u])^6 \left(g_2^9 - (27 g_3^2 + 18 g_3 \wp' [u]^2 - \wp' [u]^3 (\wp' [u] - 8 \wp' [2u]))^3 - \right. \\ & 3 g_2^6 (27 g_3^2 + 18 g_3 \wp' [u]^2 + \wp' [u]^3 (11 \wp' [u] + 8 \wp' [2u])) + \\ & 3 g_3^3 (729 g_3^4 + 972 g_3^3 \wp' [u]^2 + 36 g_3 \wp' [u]^5 (\wp' [u] + 8 \wp' [2u]) + \\ & 54 g_3^2 \wp' [u]^3 (11 \wp' [u] + 8 \wp' [2u]) + \\ & \left. \wp' [u]^6 (-11 \wp' [u]^2 + 80 \wp' [u] \wp' [2u] - 80 \wp' [2u]^2) \right) \end{aligned}$$

Here is another small example: a short program that derives the contiguous relation for Gauss' hypergeometric function. It takes about an hour on a fast machine.

```
In[11]:= contiguousRelation[triple_] :=
Module[{eqs, ser, preSol, rhsVars, rhsRule, sol},
eqs = Sum[C[k, 0] + C[k, 1] z] triple[[k]];
ser = CoefficientList[Series[eqs, {z, 0, 5}], z];
preSol = Solve[ (#1 == 0 &) /@ ser, Cases[eqs, _C, ∞]];
rhsVars = Union[Flatten[(Cases[#1, _C, ∞] &) /@ Last /@ preSol[[1]]]];
rhsRule = (#1 -> 1 &) /@ rhsVars;
sol = (MapAt[Together, #1, 2] &) /@ (preSol[[1]] /. rhsRule); Collect[
Numerator[Together[eqs /. sol /. rhsRule]], triple, Simplify]
```

```
In[12]:= all2F1Triples =
With[{fs = Hypergeometric2F1@@@{{a, b, c, z}, {a-1, b, c, z},
{a+1, b, c, z}, {a, b-1, c, z}, {a, b+1, c, z},
{a, b, c-1, z}, {a, b, c+1, z}}},
Flatten[Table[{fs[[i]], fs[[j]], fs[[k]]}, {i, 7},
{j, i+1, 7}, {k, j+1, 7}], 2]];
```

```
In[13]:= (# == 0 & /@ DeleteCases[ contiguousRelation /@ all2F1Triples, 0]) //
TableForm[#, TableSpacing -> 2] & // TraditionalForm
```

```
Out[13]//TraditionalForm=
```

$$\begin{aligned} & \{(a-c)_2 F_1(a-1, b; c; z) + \\ & (c+a(z-2)-bz)_2 F_1(a, b; c; z) + (a-az)_2 F_1(a+1, b; c; z) == 0, \\ & (c-a)_2 F_1(a-1, b; c; z) + (b-c)_2 F_1(a, b-1; c; z) - \\ & (a-b)(z-1)_2 F_1(a, b; c; z) == 0, (a-c)_2 F_1(a-1, b; c; z) + \\ & (-a-b+c)_2 F_1(a, b; c; z) + (b-bz)_2 F_1(a, b+1; c; z) == 0, \\ & (c-a)_2 F_1(a-1, b; c; z) + (c-1)(z-1)_2 F_1(a, b; c-1; z) + \\ & (a+(b-c+1)z-1)_2 F_1(a, b; c; z) == 0, \\ & -c_2 F_1(a-1, b; c; z) + (c-cz)_2 F_1(a, b; c; z) + (c-b)z_2 F_1(a, b; c+1; z) == 0, \\ & (b-c)_2 F_1(a, b-1; c; z) + (-a-b+c)_2 F_1(a, b; c; z) + \\ & (a-az)_2 F_1(a+1, b; c; z) == 0, -(a-b)(z+1)_2 F_1(a, b; c; z) - \\ & b(z+1)_2 F_1(a, b+1; c; z) + a(z+1)_2 F_1(a+1, b; c; z) == 0, \\ & -(c-1)(z+1)_2 F_1(a, b; c-1; z) - (a-c+1)(z+1)_2 F_1(a, b; c; z) + \\ & a(z+1)_2 F_1(a+1, b; c; z) == 0, \\ & c(-a-bz+c)z_2 F_1(a, b; c; z) + (a-c)(c-b)z_2 F_1(a, b; c+1; z) - \\ & ac(z-1)_2 F_1(a+1, b; c; z) == 0, (b-c)_2 F_1(a, b-1; c; z) + \end{aligned}$$

$$\begin{aligned}
& (c + b(z - 2) - az) {}_2F_1(a, b; c; z) + (b - bz) {}_2F_1(a, b + 1; c; z) = 0, \\
& (c - b) {}_2F_1(a, b - 1; c; z) + (c - 1)(z - 1) {}_2F_1(a, b; c - 1; z) + \\
& \quad (b + (a - c + 1)z - 1) {}_2F_1(a, b; c; z) = 0, \\
& -c {}_2F_1(a, b - 1; c; z) + (c - cz) {}_2F_1(a, b; c; z) + (c - a)z {}_2F_1(a, b; c + 1; z) = 0, \\
& -(c - 1)(z + 1) {}_2F_1(a, b; c - 1; z) - \\
& \quad (b - c + 1)(z + 1) {}_2F_1(a, b; c; z) + b(z + 1) {}_2F_1(a, b + 1; c; z) = 0, \\
& c(-b - az + cz) {}_2F_1(a, b; c; z) + (a - c)(c - b)z {}_2F_1(a, b; c + 1; z) - \\
& \quad bc(z - 1) {}_2F_1(a, b + 1; c; z) = 0, \\
& (c - 1)c(z - 1) {}_2F_1(a, b; c - 1; z) + c(-2zc + c + (a + b + 1)z - 1) {}_2F_1(a, b; c; z) + \\
& \quad (a - c)(b - c)z {}_2F_1(a, b; c + 1; z) = 0, \\
& -(a - b)(a - c) {}_2F_1(a - 1, b; c; z) + b(c + a(z - 2) - bz) {}_2F_1(a, b + 1; c; z) + \\
& \quad a(a + b - c) {}_2F_1(a + 1, b; c; z) = 0, \\
& (-a^2 + 2ca - a - c^2 + c) {}_2F_1(a - 1, b; c; z) + (c - 1)(c + a(z - 2) - bz) \\
& \quad {}_2F_1(a, b; c - 1; z) + a(a + (b - c + 1)z - 1) {}_2F_1(a + 1, b; c; z) = 0, \\
& -c {}_2F_1(a - 1, b; c; z) + c {}_2F_1(a, b - 1; c; z) + (a - b)z {}_2F_1(a, b; c + 1; z) = 0, \\
& (a - c)(-b + c - 1) {}_2F_1(a - 1, b; c; z) - (a + b - c)(c - 1) {}_2F_1(a, b; c - 1; z) + \\
& \quad b(a + (b - c + 1)z - 1) {}_2F_1(a, b + 1; c; z) = 0, \\
& -(a - b)(b - c) {}_2F_1(a, b - 1; c; z) - b(a + b - c) {}_2F_1(a, b + 1; c; z) + \\
& \quad a(-c - b(z - 2) + az) {}_2F_1(a + 1, b; c; z) = 0, \\
& (a - c + 1)(c - b) {}_2F_1(a, b - 1; c; z) - (a + b - c)(c - 1) {}_2F_1(a, b; c - 1; z) + \\
& \quad a(b + (a - c + 1)z - 1) {}_2F_1(a + 1, b; c; z) = 0, \\
& (a - b)(c - 1)(z + 1) {}_2F_1(a, b; c - 1; z) - b(a - c + 1)(z + 1) {}_2F_1(a, b + 1; c; z) + \\
& \quad a(b - c + 1)(z + 1) {}_2F_1(a + 1, b; c; z) = 0, \\
& (a - b)(a - c)(b - c)z {}_2F_1(a, b; c + 1; z) - bc(a + (b - c)z) {}_2F_1(a, b + 1; c; z) + \\
& \quad ac(b + (a - c)z) {}_2F_1(a + 1, b; c; z) = 0, \\
& -(c - 1)c(a + (b - c)z) {}_2F_1(a, b; c - 1; z) + \\
& \quad (b - c)(a^2 - 2ca + a + (c - 1)c)z {}_2F_1(a, b; c + 1; z) + \\
& \quad ac(-2zc + c + (a + b + 1)z - 1) {}_2F_1(a + 1, b; c; z) = 0, \\
& (-b^2 + 2cb - b - c^2 + c) {}_2F_1(a, b - 1; c; z) + (c - 1)(c + b(z - 2) - az) \\
& \quad {}_2F_1(a, b; c - 1; z) + b(b + (a - c + 1)z - 1) {}_2F_1(a, b + 1; c; z) = 0, \\
& -(c - 1)c(b + (a - c)z) {}_2F_1(a, b; c - 1; z) + \\
& \quad (a - c)(b^2 - 2cb + b + (c - 1)c)z {}_2F_1(a, b; c + 1; z) + \\
& \quad bc(-2zc + c + (a + b + 1)z - 1) {}_2F_1(a, b + 1; c; z) = 0\}
\end{aligned}$$

While these identities are all well known, generalizing this approach allowed us to derive the contiguous relations for various ${}_pF_q$ functions that were previously not known.

■ What Next?

With an NSF grant for digital libraries (see www.dlib.org/dlib/november02/zia/11zia.html), the Functions website will see two big enhancements in the next two years.

- With more than 54,000 formulas, finding the right one will not always be easy. So a natural extension is a mathematical search engine.
- “A graph is worth a thousand formulas.” A natural addition will be a graphics gallery showing a variety of visualizations of the mathematical functions of the website.

Here are some previews of some of the graphics to expect on the Functions website.

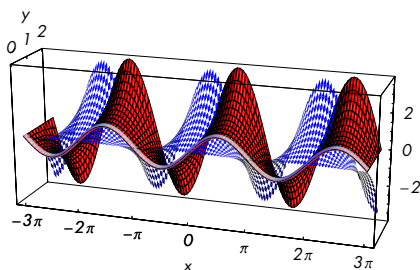


Figure 1. The function $\sin(z)$ in the complex z -plane near the real axis.

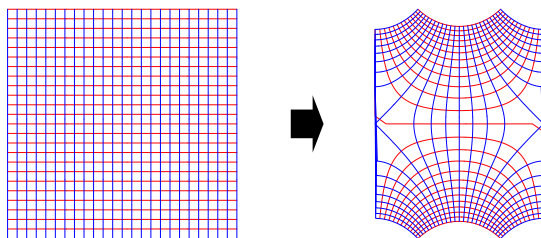


Figure 2. The image of a square grid of size $[-3, 3] \times [-3, 3]$ under the map $z \mapsto \sin^{-1}(z)$.

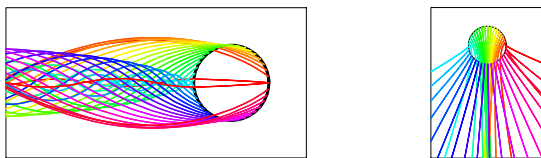


Figure 3. Integral curves of Newton's equations in the potentials $V(x, y) = \operatorname{Re}(\sin^{-1}(x + iy))$ and $V(x, y) = \operatorname{Im}(\sin^{-1}(x + iy))$. The black circle is the unit circle in the x, y -plane.



Figure 4. Contour plot of the real part of $q^{-1}(z)$ over the unit disk.

■ Use It!

I encourage the reader to visit the Functions website and to make frequent use of it. Comments and suggestions are always welcome.

Michael Trott

Special Functions Developer
Wolfram Research, Inc.
mtrott@wolfram.com

What's New in Mathematica 5

The *Mathematica* 5 Product Team

Key new technologies enable *Mathematica* 5 to outperform dedicated numerical systems in raw computational speed, while introducing a host of innovative features. Until now, technical users had to make a choice between *Mathematica*'s comprehensive, mixed numeric-symbolic environment and the faster numerical routines available in Fortran libraries or specialized numerical systems. With the release of *Mathematica* 5, this choice is no longer necessary. In addition to updated versions of *J/Link*[™] and *MathLink*[®], *Mathematica*'s standard connections to Java and C/C++, *Mathematica* 5 comes with the new *.NET/Link*[™], a much requested feature that allows developers to seamlessly integrate *Mathematica* into applications using Microsoft's .NET Framework.

■ **Overview**

With *Mathematica* 5, Wolfram Research has initiated a marked transformation in the world of technical computing, providing all the capabilities of both a dedicated numerical package and a programming language in one complete numeric and symbolic environment. *Mathematica* 5 can now do production-scale work in terms of both speed and scalability, without sacrificing any of its broad scope of features. This speed, scope, and scalability make *Mathematica* 5 a uniquely compelling all-around technical system for institution-wide adoption.

Mathematica 5 is primarily an advanced algorithm release with a large number of newly developed and refined key technologies, including those that make *Mathematica*'s performance so outstanding. New and improved numerical algorithms and routines allow *Mathematica* 5 to run as fast as or faster than many dedicated numerical software packages and libraries. *Mathematica* 5 also includes a completely new numerical differential equation solver and many other algorithmic improvements.

Mathematica 5 further extends Wolfram Research's position as the leader in providing integration with other software and standards. In addition to an updated version of *J/Link* and *MathLink*, *Mathematica*'s standard connections to Java and C/C++, *Mathematica* 5 comes with a fully functional technology preview

of *.NET/Link*, a much requested feature that allows developers to seamlessly integrate *Mathematica* into applications using Microsoft's .NET Framework.

□ **List of Major Improvements**

- Record-breaking speed through processor-optimized numerical linear algebra
- Full support for fast sparse linear algebra
- New-generation, optimized numerical solvers for ordinary and partial differential equations
- Major new algorithms for solving equations and inequalities symbolically over different domains
- Fully integrated solver for differential algebraic equations
- High-performance optimization and linear programming including interior-point method
- More generalized numeric solvers that now accept vector and general array variables
- Industry-leading solver for recurrence equations
- Broader support for assumptions in symbolic computation
- Included *.NET/Link*, providing full integration with Microsoft's .NET Framework
- Flexible import and export of DICOM, PNG, SVG, and sparse matrix formats
- Optimized versions for 64-bit hardware and operating systems
- New quick-start interactive tutorial

■ **Performance Enhancements**

□ **Fast Dense Numerical Linear Algebra**

Dense numerical linear algebra is an important building block for most of *Mathematica*'s numerical analysis functionality, ranging from data analysis over matrix operations to numerical differential equation solvers to graphics. The performance increases in *Mathematica 5* affect basically every aspect of *Mathematica*. On modern microprocessors, *Mathematica 5* now offers class-leading performance for these kinds of operations, operating on par with Fortran or MATLAB code.

Here are some examples of speedups between *Mathematica 4.2* and *Mathematica 5*.

Operation	<i>Mathematica</i> 4.2	<i>Mathematica</i> 5	Speedup
Dot product (1000 by 1000)	3.953 s	0.625 s	6.37
Solving a dense system of equations (1000 by 1000)	3.235 s	0.375 s	8.62
Inverse (1000 by 1000)	14.218 s	1.093 s	13.01
Singular values (500 by 500)	3.859 s	0.594 s	6.49

Mathematica 5 also takes advantage of multiple processors for some numerical operations (e.g., dot product) on operating systems such as Linux, Windows, and HP-UX that support this multithreaded capability.

□ High-Speed Sparse Linear Algebra

A large number of real-world problems deal with sparse matrices, that is, matrices in which most of the elements are zero. Examples of operations that involve sparse matrices include solving ordinary and partial differential equations, optimization problems, and large-scale simulations. *Mathematica* 5's implementation of sparse linear algebra is unique in that it allows for arrays of any dimension (or rank) and is fully integrated with the rest of the *Mathematica* system. The performance of basic sparse linear algebra operations is on par with, or better than, those in dedicated numerical systems.

Creating Sparse Arrays and Converting between Sparse and Dense Arrays

Adding the head **SparseArray** forces the creation of a sparse array object.

```
In[1]:= SparseArray[{{i_, j_} /; Abs[i - j] ≤ 1 → Random[]}, {5, 5}]
```

```
Out[1]= SparseArray[<13>, {5, 5}]
```

Applying **Normal** to a sparse array object gives the corresponding dense array.

```
In[2]:= Normal[%] // MatrixForm
```

```
Out[2]//MatrixForm=
```

$$\begin{pmatrix} 0.677541 & 0.957458 & 0 & 0 & 0 \\ 0.100017 & 0.476679 & 0.919703 & 0 & 0 \\ 0 & 0.0640709 & 0.289716 & 0.838933 & 0 \\ 0 & 0 & 0.678863 & 0.287626 & 0.502739 \\ 0 & 0 & 0 & 0.0169479 & 0.0544147 \end{pmatrix}$$

Operations on Sparse Matrices

The sparse array data structure and the specialized sparse algorithms allow *Mathematica* users to do operations on very large matrices. The following matrix has 10^{12} elements.

```
In[3]:= n = 106;
```

```
In[4]:= A = SparseArray[{{i_, j_} /; Abs[i - j] ≤ 1 → Random[]}, {n, n}]
Out[4]= SparseArray[<2999998>, {1000000, 1000000}]
In[5]:= b = Table[Random[], {n}];
```

Notice the big difference in size between sparse and dense representation. The dense representation requires eight terabytes of memory for storage, while the sparse array object needs only a bit over 40 megabytes.

```
In[6]:= sparseMemory = N[ByteCount[A]] Byte
Out[6]= 4.00004 × 107 Byte
In[7]:= denseMemory = N[106 106 8] Byte
Out[7]= 8. × 1012 Byte
```

Operations on sparse arrays are extremely fast. Notice that **LinearSolve** correctly detects that it has been given a sparse array object.

```
In[8]:= Timing[LinearSolve[A, b];]
Out[8]= {9.29 Second, Null}
```

We just solved an equation system with 1,000,000 equations and 1,000,000 variables. Inverting a large sparse matrix is also very fast.

```
In[9]:= Timing[A.A;]
Out[9]= {1.125 Second, Null}
In[10]:= Clear[n, A, b]
```

□ Large-Scale Linear Programming

Mathematica is now optimized for solving large-scale linear programming problems. It uses an efficient interior-point method. Similar functionality has until now been only available in expensive special-purpose packages.

Example: Solving a Standard Test Problem

The following **Import** command loads a standard linear programming test problem that comes with *Mathematica* 5.

```
In[11]:= << "Optimization`MPSData`"
In[12]:= p = Import["80bau3b.mps", "MPSData"];
In[13]:= {c, A, b, d} = ToLinearProgrammingData[p];
```

The **80bau3b** problem has about 2,000 constraints and about 10,000 variables.

```
In[14]:= Dimensions[A]
Out[14]= {2262, 9799}
```

This solves the linear programming problem and returns the optimizing argument.

```
In[15]:= xo = LinearProgramming[c, A, b, d, Method → "InteriorPoint"]; //
Timing
```

```
Out[15]:= {5.188 Second, Null}
```

This is the optimal value.

```
In[16]:= c.xo
```

```
Out[16]:= 987224.
```

```
In[17]:= Clear[c, A, b, d, p]
```

This problem could not have been solved in *Mathematica* 4.2 because a dense form of the data would have been used. The memory requirements and processing time for such an operation would have made the problem infeasible to solve.

Example: Solving Another Standard Test Problem

This is an example of solving a linear programming problem with 232,000 variables and 10,000 inequalities.

```
In[18]:= {c, A, b, d} =
ToLinearProgrammingData[Import["osa-60.mps", "MPSData"]];
```

```
In[19]:= Dimensions[A]
```

```
Out[19]:= {10280, 232966}
```

```
In[20]:= AbsoluteTiming[Timing[
xo = LinearProgramming[c, A, b, d, Method → "InteriorPoint"];]]
```

```
Out[20]:= {113.9120820 Second, {102.75 Second, Null}}
```

As in the previous example, here is the optimal value.

```
In[21]:= c.xo
```

```
Out[21]:= 4.04407 × 106
```

```
In[22]:= Clear[c, A, b, d]
```

□ Big-Number Arithmetic

Mathematica has been able to handle numbers with millions of digits for quite some time, but more efficient implementations and better algorithms have improved the performance by a factor of up to three for numbers with less than one thousand digits. For numbers with millions of digits, performance improvements are even higher when an asymptotically more efficient algorithm is used.

The big-number performance in *Mathematica* 5 is on par with, or faster than, the performance of special-purpose libraries and is unchallenged for general computation systems.

Operation	<i>Mathematica</i> 4.2	<i>Mathematica</i> 5	Speedup
Multiplying two 1000-digit real numbers 10000 times	0.797 s	0.406 s	1.96
Multiplying two integers with 1000 digits 10000 times	1.000 s	0.407 s	2.46
Computing the greatest common divisor of two integers with 100000 and 100 digits	0.407 s	0.047 s	8.66
Calculating the square roots of 1000 1000-digit real numbers	0.359 s	0.062 s	5.79

□ **64-Bit Platform Support**

Users running increasingly larger computations and applications can now access nearly a million terabytes (1 terabyte = 1,024 gigabytes), overcoming the 4-gigabyte address ceiling in 32-bit systems like the current Intel IA-32-bit architecture.

The combination of a 64-bit address space and fast numerics, part of Wolfram Research's gigaNumerics initiative, lets *Mathematica* users solve very large problems.

Mathematica 5 is optimized for a large number of 64-bit CPUs and operating systems, including Sun Solaris for UltraSPARC, HP-UX for PA-RISC, IBM AIX for the Power architecture, HP Tru64 Unix on Alpha, and Linux on Alpha. The two main benefits are the ability to solve vastly larger problems than on 32-bit platforms and speed increases for big-number arithmetic due to the 64-bit word length.

This 64-bit optimization will also let *Mathematica* users take full advantage of planned performance increases in future versions of these 64-bit processors.

□ **Faster MathLink**

MathLink uses TCP/IP devices for communications between parts of *Mathematica*, such as the front end and the kernel, as well as for the primary means of communication between multiple *Mathematica* kernels, such as in grid*Mathematica* clusters.

A new TCP/IP protocol allows *Mathematica* 5 to communicate at the speed of the underlying network. On a standard 100Base-T network, bandwidth improves by a factor of 10 and latency by a factor of 200. On faster networks, such as Gigabit networks and crossbars found in leading-edge computing centers, the gains are even higher. Additional improvements—for example, a new **SharedMemory** protocol for Windows platforms—give additional 10-fold improvement for communication between parts of *Mathematica* on the same machine.

Because all *MathLink* bindings (*J/Link*, *.NET/Link*, and almost all **Import** / **Export** formats) rely on *MathLink*, these will also improve as a result of improvements in *MathLink*.

■ Numeric Computations

□ **NDSolve**

The function **NDSolve** has been completely rewritten, resulting in better performance and much greater flexibility. While many of the enhancements are transparent and will benefit all users, the new **NDSolve** gives advanced users many more options for selecting methods, setting evaluation options, and monitoring the progress of the solver—and even ways for implementing their own custom solvers into **NDSolve**.

Some of the most significant improvements include:

- New additional solving methods including explicit Runge-Kutta methods, implicit Runge-Kutta methods of arbitrary order, Deuffhard arbitrary-order extrapolation methods, BDF-based methods, and double step
- More efficient implementations leading to large speed increases for many types of differential equations
- **NDSolve** now supporting vector and array variables
- **NDSolve** now solving some differential-algebraic equations
- New options **EvaluationMonitor** and **StepMonitor** that allow monitoring of the solution progress and more fine-tuning of the solving procedure
- **NDSolve** providing a framework for inclusion of methods defined in the *Mathematica* language.

Example: Solving a System of Differential Equations

The following example demonstrates the use of vector variables to specify a system of differential equations of dimension 12.

To assist in the visualization of the results of the following numerical differential equation, it is useful to view the solution as a three-dimensional graphic.

```

In[1]:= IntrinsicCurve3D[{κ_, τ_}, {s_, smin_, smax_}, {s0_?NumericQ,
      x0_?VectorQ, t0_?VectorQ, n0_?VectorQ}, opts__?OptionQ] :=
Module[{x, t, n, b, s, c},
  c = x /. First@
    NDSolve[
      {
        x'[s] == t[s],           x[s0] == x0,
        t'[s] == κ n[s],         t[s0] == t0,
        n'[s] == τ b[s] - κ t[s], n[s0] == n0,
        b'[s] == -τ n[s],        b[s0] == Cross[t0, n0]
      },
      {x, t, n, b},
      {s, smin, smax}
    ];
  ParametricPlot3D[c[s], {s, smin, smax}, opts]
];

In[2]:= IntrinsicCurve3D[{κ_, τ_},
      {s_, smin_, smax_}, opts__?OptionQ] :=
IntrinsicCurve3D[{κ, τ}, {s, smin, smax},
      {0, {0, 0, 0}}, {1, 0, 0}, {0, 1, 0}}, opts];

```

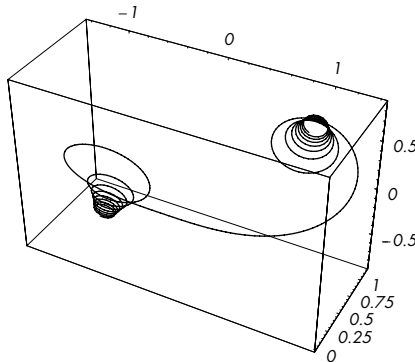
The following example is of a space curve defined by curvature and torsion.

```

In[3]:= IntrinsicCurve3D[{Abs[s], 0.3}, {s, -10, 10}, PlotPoints -> 500];

```

From In[3]:=



Example: Solving a Matrix Differential Equation

In this example we solve the matrix differential equation $X'(t) = A.X(t)$ with

$$A = -\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \text{ and the initial condition } X(0) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

```

In[4]:= A = -{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

```

```

In[5]:= matrixExpA = X /. First[NDSolve[
      X'[t] == A.X[t] ^ X[0] == IdentityMatrix[3], X, {t, 0, 5}]]

```

```

Out[5]:= InterpolatingFunction[{{0., 5.}}, <>]

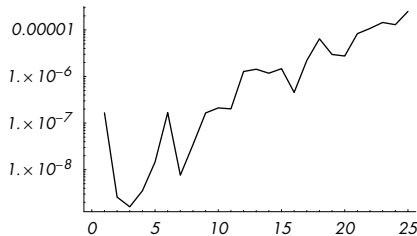
```

We can define a particular matrix norm and look at the differences.

```
In[6]:= << "Graphics`Graphics`"
```

```
In[7]:= LogListPlot[Table[Norm[MatrixExp[A, t] - matrixExpA[t]],
    {t, .2, 5, .2}], PlotJoined -> True];
```

From In[7]:=



Example: Solving a Differential-Algebraic Equation

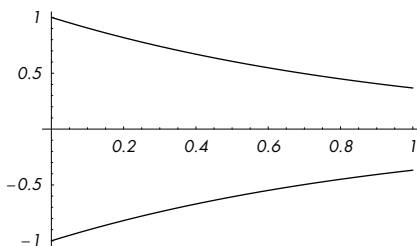
In this case we have one differential equation $x'[t] = x[t] + 2y[t]$ and one explicit algebraic equation $0 = x[t] + y[t]$. Notice how (in this case only) we give one initial value equation $x[0] = 1$. The initial value equation for $y[0]$ is implicitly determined by the algebraic equation.

```
In[8]:= NDSolve[x'[t] == x[t] + 2y[t] & 0 == x[t] + y[t] & x[0] == 1,
    {x, y}, {t, 0, 1}]
```

```
Out[8]:= {{x -> InterpolatingFunction[{{0., 1.}}, <>],
    y -> InterpolatingFunction[{{0., 1.}}, <>]}}
```

```
In[9]:= Plot[Evaluate[{x[t], y[t]} /. First[%]], {t, 0, 1}];
```

From In[9]:=



Example: Modeling an Exothermic Chemical Reactor

The following is an exothermic chemical reactor model. In this simple first-order isomerization process, the heat is removed through an external cooling circuit. In this case we have c product concentration, T product temperature, and R reaction rate. Furthermore, we have c_0 reactant concentration, T_0 reactant temperature, T_c cooling circuit temperature, and $k_1, \dots, k_4$ constants.

```
In[10]:= << "Graphics`Colors`"
```

```

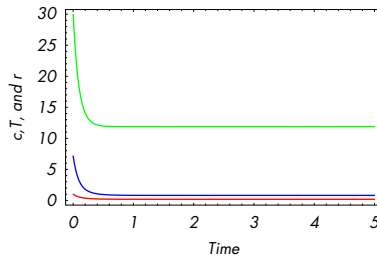
In[11]:= results =
  Block[{k1 = 1, k2 = 1, k3 = 10, k4 = 10, c0 = 1, T0 = 30, Tc = 10},
    NDSolve[{
      c'[t] == k1 (c0 - c[t]) - r[t],
      T'[t] == k1 (T0 - T[t]) + k2 r[t] - k3 (T[t] - Tc),
      0 == r[t] - k3 Exp[-k4/T[t]] c[t],
      c[0] == c0,
      T[0] == T0},
      {c, T, r},
      {t, 0, 5}
    ]
  ]

Out[11]:= {c -> InterpolatingFunction[{{0., 5.}}, <>],
  T -> InterpolatingFunction[{{0., 5.}}, <>],
  r -> InterpolatingFunction[{{0., 5.}}, <>]}

In[12]:= Plot[Evaluate[{c[t], T[t], r[t]} /. First[results]],
  {t, 0, 5}, PlotStyle -> {Red, Green, Blue},
  Frame -> True, FrameLabel -> {"Time", "c, T, and r"}];

```

From In[12]:=



Example: Solving a Partial Differential Equation

A soliton is a solitary wave that survives collisions with other solitary waves. For example, ocean waves are solitons. The Korteweg-de Vries equation was the first differential equation describing a soliton. Technical applications of solitons include long-haul optical fiber transmission systems.

This is the Korteweg-de Vries equation in one spatial dimension.

```

In[13]:= KdV[{α_, t_, x_] :=
  D[ψ[t, x], t] + α ψ[t, x] D[ψ[t, x], x] + D[ψ[t, x], {x, 3}] == 0;

```

This is a solitary wave solution to the KdV equation above.

```

In[14]:= S[{α_, c0_, x0_}, t_, x_] := (3 c0 / α) (Sech[ (sqrt(c0) / 2) (x - x0 - c0 t) ])^2;

```

Because we are going to use periodic boundary conditions, we will add up all the period contributions to get the following initial condition.

```

In[15]:= Φ = S[{1, 3, 5}, 0, x] + S[{1, 3/2, 15}, 0, x] +
  S[{1, 3, 5+40}, 0, x] + S[{1, 3/2, 15+40}, 0, x] +
  S[{1, 3, 5-40}, 0, x] + S[{1, 3/2, 15-40}, 0, x];

```

```
In[16]:= (ϕ /. x → 0.) == (ϕ /. x → 40.)
```

```
Out[16]:= True
```

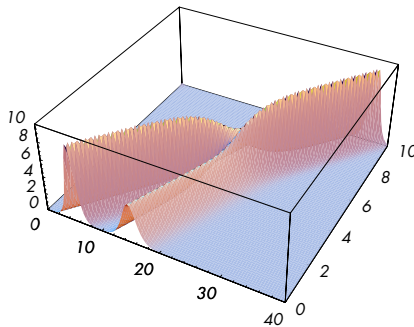
Here we can see how these solitary waves interact.

```
In[17]:= s = NDSolve[KdV[{1}, t, x] && ψ[0, x] == ϕ && ψ[t, 0] == ψ[t, 40], ψ,
             {t, 0, 10}, {x, 0, 40}, Method → StiffnessSwitching]; // Timing
```

```
Out[17]:= {12.484 Second, Null}
```

```
In[18]:= Plot3D[Evaluate[ψ[t, x] /. First[s]], {x, 0, 40}, {t, 0, 10},
             PlotRange → {0, 10}, PlotPoints → 100, Mesh → False];
```

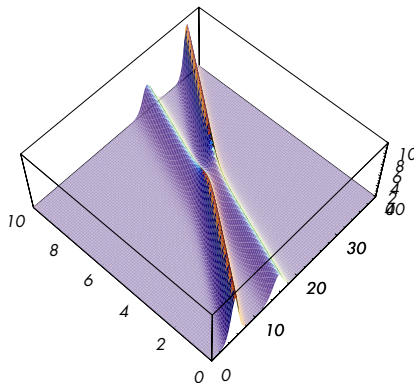
From In[18]:=



Notice the nonlinear interaction when they pass each other.

```
In[19]:= Show[%, ViewPoint → {-1.855, -1.725, 4.000}];
```

From In[19]:=



□ FindRoot

The function **FindRoot** now supports general array variables and includes new and improved algorithms that lead to speed increases and better handling of large-scale problems on computers with limited memory.

Example: Solving a Matrix Riccati Equation

Here is an example of solving a matrix equation—in this case a continuous-time algebraic Riccati equation $Q + A^T P + P A - P B R^{-1} B^T P = 0$.

Suppose we have a system (e.g., a double integrator, or some type of mechanical system) and constraints.

```
In[20]:= A = {{0, 1}, {0, 0}}; B = {{0}, {1}};
```

```
In[21]:= Q = {{1, 0}, {0, 1}}; R = {{1}};
```

This finds a matrix root to the equation $Q + A^T P + P A - P B R^{-1} B^T P = 0$.

```
In[22]:= FindRoot[
  Transpose[A] . P + P . A - P . B . Inverse[R] . Transpose[B] . P == -Q,
  {P, {{10, 2}, {2, 10}}}
Out[22]:= {P -> {{1.73205, 1.}, {1., 1.73205}}}
```

FindFit

The new function **FindFit** supersedes and extends the functionality of the functions **Fit** and **NonlinearFit**. **FindFit** supports array variables, provides more methods, and can use the functions **StepMonitor** and **EvaluationMonitor** for keeping track of the process.

Example: Fitting Data

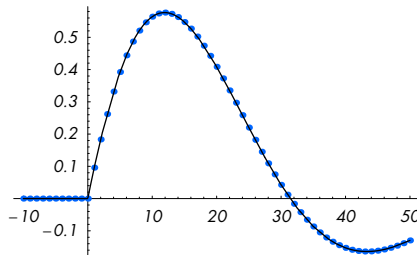
FindFit can even fit data using a piecewise continuous function.

```
In[23]:= data = Table[{t, Which[t < 0, 0, True, e-0.04 t Sin[0.1 t]]}, {t, -10, 50}];
model = Which[t < 0, 0, True, e-a t Sin[b t]];
sol = FindFit[data, model, {{a, 0.02}, {b, 0.3}}, {t}]
Out[23]:= {a -> 0.04, b -> 0.1}
```

The data can be shown with the following fitting function.

```
In[26]:= Show[ListPlot[data, PlotJoined -> False, PlotStyle ->
  {PointSize[0.015], Hue[0.6]}, DisplayFunction -> Identity],
  Plot[model /. sol, {t, -10, 50}, DisplayFunction -> Identity],
  DisplayFunction -> $DisplayFunction];
```

From In[26]:=



Example: Fitting a Circle

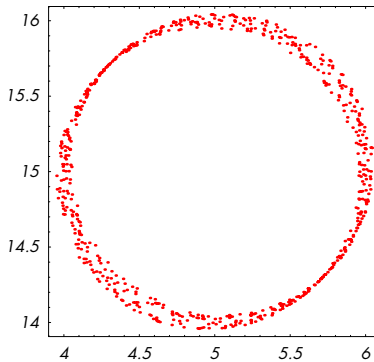
Suppose we are taking measurements along a circle.

```
In[27]:= data = Table[{Cos[t] + 5, Sin[t] + 15, 0.} +
    Random[Real, {-0.05, 0.05}], {t, 0., 2 Pi, 0.01}];

In[28]:= << "Graphics`Colors`"

In[29]:= dg = ListPlot[Take[data, All, 2],
    Frame → True, PlotStyle → Red, AspectRatio → Automatic];
```

From In[29]:=



This fits a circle with radius R and center $\{x0, y0\}$ to the data.

```
In[30]:= Clear[R]

In[31]:= s = FindFit[data, (x - x0)^2 + (y - y0)^2 - R^2, {x0, y0, R}, {x, y}]

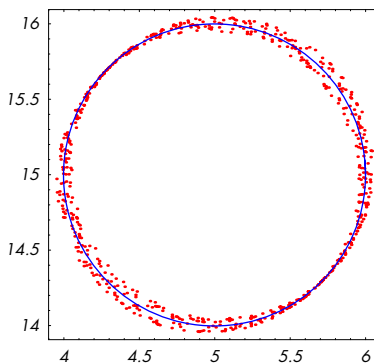
Out[31]:= {x0 → 4.99835, y0 → 14.9984, R → 1.00091}

In[32]:= fg =
    Show[Graphics[{RGBColor[0, 0, 1], Circle[{x0 /. s, y0 /. s}, R /. s]}],
    DisplayFunction → Identity];
```

This shows the data and the fitted circle together.

```
In[33]:= Show[{dg, fg}, AspectRatio → Automatic];
```

From In[33]:=



□ FindMinimum and FindMaximum

The new function **FindMaximum** has been added for convenience. The functions **FindMinimum** and **FindMaximum** now support general array variables, offer new and improved algorithms, and work better in limited-memory situations.

Example: Computing a Matrix Norm

In this case we are using a vector variable.

```
In[34]:= A = {{1, 2}, {3, 4}};
In[35]:= FindMaximum[Norm[A.x, 4] / Norm[x, 4], {x, {1, 1}}]
Out[35]= {5.95734, {x -> {0.949799, 1.04995}}}
```

■ Symbolic Computations

□ DSolve

The function **DSolve** can now find all rational solutions to systems of linear equations with rational coefficients. It can also solve linear systems of differential equations with constant coefficients.

Example: Linear System with Rational Function Coefficients

Here is a linear system with rational function coefficients that has a rational solution.

```
In[1]:= DSolve[
  {
    u'[t] ==  $\frac{(6 + 2t - 3t^3 - t^5) u[t]}{t(-3 - 2t + t^2)(-1 + t^3)} + \frac{(5 + t) v[t]}{(-3 - 2t + t^2)(-1 + t^3)}$ ,
    v'[t] ==  $-\frac{4t(-3 - 2t + t^2) u[t]}{(5 + t)(-1 + t^3)} + \frac{(1 + 20t^2 + 3t^3) v[t]}{(5 + t)(-1 + t^3)}$ ,
    {u, v},
    t
  ]
Out[1]= {{u -> Function[{t},  $\frac{t C[1]}{-3 - 2t + t^2} + \frac{t^2 C[2]}{-3 - 2t + t^2}$ ],
  v -> Function[{t},  $\frac{C[1]}{5 + t} + \frac{t^4 C[2]}{5 + t}$ ]]}}
```

In addition, **DSolve** can solve some singular or differential-algebraic equations—in particular, systems of linear equations of the form, where the matrix **A** is singular:

$$A.x'[t] + B.x[t] == g[t]$$

Notice how the general solution to this differential-algebraic equation has only one general parameter even though it is a system of order two. Again, one degree of freedom is removed by having an algebraic equation between variables.

```
In[2]:= DSolve[x'[t] == x[t] + 2 y[t] ∧ 0 == x[t] + y[t], {x, y}, t]
```

```
Out[2]= {{x → Function[{t},  $\frac{1}{4} e^{-t} C[1]$ ], y → Function[{t},  $-\frac{1}{4} e^{-t} C[1]$ ]}}
```

In this case you have the freedom to choose only one initial value freely.

```
In[3]:= DSolve[x'[t] == x[t] + 2 y[t] ∧ 0 == x[t] + y[t] ∧ x[0] == 1, {x, y}, t]
```

```
Out[3]= {{x → Function[{t},  $e^{-t}$ ], y → Function[{t},  $-e^{-t}$ ]}}
```

This is the implied initial value of the last variable.

```
In[4]:= y[0] /. First[%]
```

```
Out[4]= -1
```

□ RSolve

The function **RSolve** can solve systems of linear and nonlinear difference equations, difference-algebraic equations, and partial difference equations. It can also solve q -difference or divide-and-conquer equations.

Just as with differential equations, fairly innocent-looking difference equations can generate very complicated solutions, and ultimately there are not general-enough functions to express all solutions in closed form. In many cases, however, closed-form solutions can be found as in the following examples.

Example: Legendre Polynomials

RSolve recognizes the difference equation for Legendre polynomials.

```
In[5]:= RSolve[u[x] - ((3 + 2*x)*(d + c*z)*u[1 + x])/(1 + x) +
  ((2 + x)*u[2 + x])/(1 + x) == 0, u, x]
```

```
Out[5]= {{u →
  Function[{x}, C[1] LegendreP[x, d + c z] + C[2] LegendreQ[x, d + c z]]}}
```

Example: General and Particular Solutions to $y_{k+1} = (k+1)y_k$

This gives the general solution to $y_{k+1} = (k+1)y_k$ involving one arbitrary constant $C[1]$.

```
In[6]:= RSolve[y[k + 1] == (k + 1) y[k], y, k]
```

```
Out[6]= {{y → Function[{k}, C[1] Gamma[1 + k]]}}
```

This solves the difference equation $y_{k+1} = (k+1)y_k$ and the initial value equation $y_0 = 1$.

```
In[7]:= RSolve[y[k + 1] == (k + 1) y[k] && y[0] == 1, y, k]
```

```
Out[7]= {{y → Function[{k}, Gamma[1 + k]]}}
```

This shows that the solution satisfies the difference and the initial value equations.

```
In[8]:= y[k+1] == (k+1) y[k] && y[0] == 1 /. % // FullSimplify
```

```
Out[8]:= {True}
```

Example: Second-Order Difference Equation with Boundary Conditions

This solves a second-order difference equation with two boundary conditions.

```
In[9]:= s =
      RSolve[y[k+2] + 2 y[k+1] + y[k] == 0 && y[0] == 1 && y[1] == 2, y, k]
```

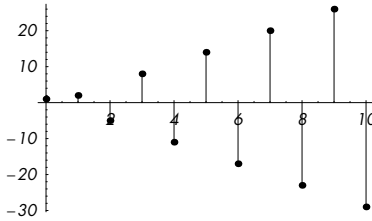
```
Out[9]:= {{y -> Function[{k}, -(-1)^k (-1+3k)]}}
```

We can visualize the resulting sequence using a typical stem plot.

```
In[10]:= << Graphics`MultipleListPlot`
```

```
In[11]:= MultipleListPlot[
      Table[Evaluate[{k, y[k] /. First[s]}], {k, 0, 10}],
      SymbolShape -> Stem];
```

From In[11]:=



Example: Linear Second-Order Equation with Varying Coefficients

This example solves $y(n) n^2 + (n^2 + 1) y(n+1) + y(n+2) = 0$, a linear second-order, varying-coefficient equation.

```
In[12]:= RSolve[y[n+2] + (n^2 + 1) y[n+1] + n^2 y[n] == 0, y, n]
```

```
Out[12]:= {{y -> Function[{n},
      -(-1)^n C[1] - (-1)^n C[2] (HypergeometricPFQ[{1, 1, 1}, {}, 1] -
      Gamma[n]^2 HypergeometricPFQ[{1, n, n}, {}, 1])]]}}
```

Example: System of Difference-Algebraic Equations

This shows a linear system of difference-algebraic equations. Notice how this second-order system only has one general constant in its solution. This is because one degree of freedom has been removed by the algebraic equation.

```
In[13]:= RSolve[x[k+1] == x[k] + 2 y[k] & 0 == x[k] + y[k], {x, y}, k]
```

```
Out[13]:= {{x -> Function[{k}, 1/4 (-1)^k C[1]], y -> Function[{k}, -1/4 (-1)^k C[1]]}}
```

This is the corresponding initial value problem with only one initial value given.

```
In[14]:= RSolve[x[k+1] == x[k] + 2 y[k] & 0 == x[k] + y[k] & x[0] == 1, {x, y}, k]
```

```
Out[14]:= {{x -> Function[{k}, (-1)^k], y -> Function[{k}, -(-1)^k]}}
```

This is the resulting initial value for y .

```
In[15]:= y[0] /. First[%]
```

```
Out[15]= -1
```

Example: Partial Difference Equations

Partial difference equations have several independent variables. In these cases the general solution is parameterized by general functions.

Here is a linear partial difference equation with a general arbitrary function $C[1][l - k]$.

```
In[16]:= RSolve[u[k+1, l+1] == a u[k, l], u, {k, l}]
```

```
Out[16]= {{u -> Function[{k, l}, a-1+k C[1][-k+1]]}}
```

Example: Divide-and-Conquer Equations

RSolve can also solve so-called q -difference equations or divide-and-conquer equations. These are equations of the form $F[k, y[k], y[q k], \dots, y[q^n k]] = 0$.

If you compare these to the normal difference equations, $F[k, y[k], y[k+1], \dots, y[k+n]] = 0$, you can see that the arguments to the dependent function y form an arithmetic sequence for difference equations and a geometric sequence for q -difference equations. Hence, these are also often called arithmetic and geometric difference equations, respectively. Geometric difference equations often come up in the analysis of algorithms, combinatorics, and number theory.

This is a first-order linear q -difference equation that gives the number of comparisons when doing a binary search.

```
In[17]:= RSolve[b[n] == b[n/2] + 1 && b[1] == 1, b, n]
```

```
Out[17]= {{b -> Function[{n},  $\frac{\text{Log}[2] + \text{Log}[n]}{\text{Log}[2]}$ ]}}
```

This gives the arithmetic complexity for a fast Fourier transform operation.

```
In[18]:= RSolve[f[n] == 2 f[n/2] + (3/2) n, f, n]
```

```
Out[18]= {{f -> Function[{n},  $2 n C[1] + \frac{3 n \text{Log}[n]}{2 \text{Log}[2]}$ ]}}
```

□ Reduce

The function **Reduce** has been extended to solve equations involving any combination of equalities, inequalities, existential quantifiers, universal quantifiers, and domain specifications.

Example: Solving $x^2 - 2y^2 = 1$ over Different Domains

In this example we solve the equation $x^2 - 2y^2 = 1$ over different domains. In all cases there are infinitely many solutions.

This solves a simple equation over the complexes.

In[19]:= **Reduce**[$x^2 - 2y^2 == 1$, {**x**, **y**}, **Complexes**]

$$\text{Out[19]}= y == -\frac{\sqrt{-1+x^2}}{\sqrt{2}} \mid \mid y == \frac{\sqrt{-1+x^2}}{\sqrt{2}}$$

This solves the same equation over the reals and requires inequalities to describe the solution.

In[20]:= **Reduce**[$x^2 - 2y^2 == 1$, {**x**, **y**}, **Reals**]

$$\begin{aligned} \text{Out[20]}= & x \leq -1 \ \&\& \left(y == -\frac{\sqrt{-1+x^2}}{\sqrt{2}} \mid \mid y == \frac{\sqrt{-1+x^2}}{\sqrt{2}} \right) \mid \mid \\ & x \geq 1 \ \&\& \left(y == -\frac{\sqrt{-1+x^2}}{\sqrt{2}} \mid \mid y == \frac{\sqrt{-1+x^2}}{\sqrt{2}} \right) \end{aligned}$$

Solving the same equation over the integers requires additional parameters to describe the solution. These equations are called Diophantine equations. The following example is a special form of a Diophantine equation called Pell's equation.

In[21]:= **Reduce**[$x^2 - 2y^2 == 1$, {**x**, **y**}, **Integers**]

$$\begin{aligned} \text{Out[21]}= & C[1] \in \text{Integers} \ \&\& C[1] \geq 0 \ \&\& x == \frac{1}{2} \left(-(3-2\sqrt{2})^{C[1]} - (3+2\sqrt{2})^{C[1]} \right) \ \&\& \\ & y == -\frac{(3-2\sqrt{2})^{C[1]} - (3+2\sqrt{2})^{C[1]}}{2\sqrt{2}} \mid \mid \\ & C[1] \in \text{Integers} \ \&\& C[1] \geq 0 \ \&\& x == \frac{1}{2} \left(-(3-2\sqrt{2})^{C[1]} - (3+2\sqrt{2})^{C[1]} \right) \ \&\& \\ & y == \frac{(3-2\sqrt{2})^{C[1]} - (3+2\sqrt{2})^{C[1]}}{2\sqrt{2}} \mid \mid \\ & C[1] \in \text{Integers} \ \&\& C[1] \geq 0 \ \&\& x == \frac{1}{2} \left((3-2\sqrt{2})^{C[1]} + (3+2\sqrt{2})^{C[1]} \right) \ \&\& \\ & y == -\frac{(3-2\sqrt{2})^{C[1]} - (3+2\sqrt{2})^{C[1]}}{2\sqrt{2}} \mid \mid \\ & C[1] \in \text{Integers} \ \&\& C[1] \geq 0 \ \&\& x == \frac{1}{2} \left((3-2\sqrt{2})^{C[1]} + (3+2\sqrt{2})^{C[1]} \right) \ \&\& \\ & y == \frac{(3-2\sqrt{2})^{C[1]} - (3+2\sqrt{2})^{C[1]}}{2\sqrt{2}} \end{aligned}$$

This solves the equation over the integers modulo 12.

In[22]:= **Reduce**[$x^2 - 2y^2 == 1$, {**x**, **y**}, **Modulus** $\rightarrow$ 12]

$$\begin{aligned} \text{Out[22]}= & x == 1 \ \&\& y == 0 \mid \mid x == 1 \ \&\& y == 6 \mid \mid x == 3 \ \&\& y == 2 \mid \mid x == 3 \ \&\& y == 4 \mid \mid \\ & x == 3 \ \&\& y == 8 \mid \mid x == 3 \ \&\& y == 10 \mid \mid x == 5 \ \&\& y == 0 \mid \mid x == 5 \ \&\& y == 6 \mid \mid \\ & x == 7 \ \&\& y == 0 \mid \mid x == 7 \ \&\& y == 6 \mid \mid x == 9 \ \&\& y == 2 \mid \mid x == 9 \ \&\& y == 4 \mid \mid \\ & x == 9 \ \&\& y == 8 \mid \mid x == 9 \ \&\& y == 10 \mid \mid x == 11 \ \&\& y == 0 \mid \mid x == 11 \ \&\& y == 6 \end{aligned}$$

This solves the same equation over a domain of real x and complex y .

In[23]:= **Reduce**[$x^2 - 2y^2 == 1$ && $x \in \text{Reals}$ && $y \in \text{Complexes}$, { x , y }]

Out[23]= $x < -1$ && $\left(y == -\frac{\sqrt{-1+x^2}}{\sqrt{2}} \mid \mid y == \frac{\sqrt{-1+x^2}}{\sqrt{2}}\right) \mid \mid x == -1$ && $y == 0 \mid \mid$
 $-1 < x < 1$ && $\left(y == -\frac{i\sqrt{1-x^2}}{\sqrt{2}} \mid \mid y == \frac{i\sqrt{1-x^2}}{\sqrt{2}}\right) \mid \mid$
 $x == 1$ && $y == 0 \mid \mid x > 1$ && $\left(y == -\frac{\sqrt{-1+x^2}}{\sqrt{2}} \mid \mid y == \frac{\sqrt{-1+x^2}}{\sqrt{2}}\right)$

Example: Equations with Infinitely Many Roots

Reduce will generate complete solutions. Here is an example in which there are also infinitely many real roots.

In[24]:= **Reduce**[$\text{Tan}[2x]^2 + \text{Cot}[2x]^2 + 2\text{Tan}[2x] + 2\text{Cot}[2x] == 6$ && $x \in \text{Reals}$, x]

Out[24]= $C[1] \in \text{Integers}$ &&
 $\left(x == \frac{1}{8}(\pi + 4\pi C[1]) \mid \mid x == \frac{1}{24}(-5\pi + 12\pi C[1]) \mid \mid x == \frac{1}{24}(-\pi + 12\pi C[1])\right)$

Example: Equations Including Quantifiers

Another type of equation is one that includes quantifiers, such as the existential ($\exists$) and universal ($\forall$) quantifiers defined in *Mathematica* with **Exists** ($\exists$) and **ForAll** ($\forall$).

This reduces the a and b values that make the quadratic polynomial positive for all real x .

In[25]:= **Reduce**[**ForAll**[x , $x \in \text{Reals}$, $x^2 + ax + b \geq 0$], { a , b }, **Reals**]

Out[25]= $b \geq \frac{a^2}{4}$

□ Resolve

The function **Resolve** can eliminate quantifiers from arbitrary polynomial systems in complex or real variables using the same methods used by the function **Reduce**. For cases in which obtaining an implicit quantifier-free form of the system is easier than computing explicit solutions, **Resolve** returns the implicit form. **Resolve** can also eliminate quantifiers involving Boolean variables.

Example: Real Coefficients

This gives the conditions that real coefficients a , b , and c have to satisfy so that a quadratic polynomial in a real variable x is positive.

In[26]:= **Resolve**[**ForAll**[x , { x , a , b , c } $\in \text{Reals}$, $cx^2 + bx + a > 0$]]

Out[26]= $b < 0$ && $c > 0$ && $a > \frac{b^2}{4c} \mid \mid b == 0$ && $c \geq 0$ && $a > 0 \mid \mid b > 0$ && $c > 0$ && $a > \frac{b^2}{4c}$

Example: Does $x^2 + y^2 < 1$ Imply that $y > x^4 - 2$?

This tests whether $x^2 + y^2 < 1$ implies that $y > x^4 - 2$.

```
In[27]:= Resolve[ $\forall_{\{x,y\}}$  Implies[ $x^2 + y^2 < 1$ ,  $y > x^4 - 2$ ]]
```

```
Out[27]:= True
```

In this case we can visually check as well.

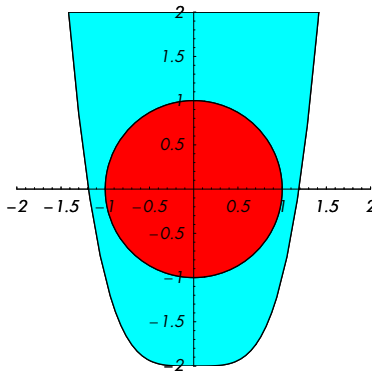
```
In[28]:= << "Graphics`InequalityGraphics`"
```

```
In[29]:= A = InequalityPlot[ $x^2 + y^2 < 1$ , {x}, {y},
  PlotRange → {{-2, 2}, {-2, 2}},
  Fills → {{1, 2}, RGBColor[1, 0, 0]}},
  DisplayFunction → Identity];
```

```
In[30]:= B = InequalityPlot[ $y > x^4 - 2$ , {x, -2, 2}, {y, -2, 2},
  PlotRange → {{-2, 2}, {-2, 2}},
  DisplayFunction → Identity];
```

```
In[31]:= Show[{B, A}, DisplayFunction → $DisplayFunction];
```

From In[31]:=



□ FindInstance

The new function **FindInstance** takes the same input as **Reduce** but gives, at most, the requested number of different solutions. **FindInstance** works for all problems **Reduce** can solve fully. For problems in which the complete solution is not needed, finding instances of solutions can be much faster than trying to find the whole solution set. **FindInstance** may also be able to find instances in some problems that **Reduce** cannot solve.

Example: Finding Real Solutions to $x^2 - 2y^2 = 1$

This example uses **Reduce** to find the general formula for all real solutions to $x^2 - 2y^2 = 1$.

```
In[32]:= Reduce[ $x^2 - 2y^2 == 1$ , {x, y}, Reals]
```

```
Out[32]:=  $x \leq -1 \ \&\& \left( y = -\frac{\sqrt{-1+x^2}}{\sqrt{2}} \ || \ y = \frac{\sqrt{-1+x^2}}{\sqrt{2}} \right) \ ||$   

 $x \geq 1 \ \&\& \left( y = -\frac{\sqrt{-1+x^2}}{\sqrt{2}} \ || \ y = \frac{\sqrt{-1+x^2}}{\sqrt{2}} \right)$ 
```

FindInstance gives instances of real numbers that solve the above equation.

```
In[33]:= FindInstance[x^2 - 2 y^2 == 1, {x, y}, Reals, 3]
```

```
Out[33]= {{x -> -239, y -> -4 Sqrt[1785]},
           {x -> -148, y -> -7 Sqrt[447/2]}, {x -> 223, y -> 4 Sqrt[1554]}}
```

□ Maximize/Minimize

The functions **Maximize** and **Minimize** find the exact global maximum or minimum for a function over a region. They generally work with polynomial input; however, they can handle some transcendental inputs.

Example: Basic Geometrical Properties

Here **Maximize** shows that the rectangle with the maximal area for a given circumference is a square.

```
In[34]:= Maximize[{x y, 2 x + 2 y == 1}, {x, y}]
```

```
Out[34]= {1/16, {x -> 1/4, y -> 1/4}}
```

This gives the maximum volume of a cylinder where the surface area is 24π .

```
In[35]:= Maximize[{Pi r^2 h, 2 Pi r^2 + 2 Pi r h == 24 Pi && r >= 0 && h >= 0}, {r, h}]
```

```
Out[35]= {16 Pi, {r -> 2, h -> 4}}
```

Example: Unconstrained Problems

Minimize and **Maximize** yield lists giving the value attained at the minimum or maximum, as well as with rules specifying where the minimum or maximum occurs.

This finds the minimum of a quadratic function.

```
In[36]:= Minimize[x^2 - 3 x + 6, x]
```

```
Out[36]= {15/4, {x -> 3/2}}
```

Applying the rule for **x** gives the value at the minimum.

```
In[37]:= x^2 - 3 x + 6 /. Last[%]
```

```
Out[37]= 15/4
```

This maximizes with respect to **x** and **y**.

```
In[38]:= Maximize[5 x y - x^4 - y^4, {x, y}]
```

```
Out[38]= {25/8, {x -> -Sqrt[5]/2, y -> -Sqrt[5]/2}}
```

Example: Constrained Problems

Minimize[*expr*, *x*] minimizes *expr* allowing *x* to range over all possible values from $-\infty$ to $+\infty$. **Minimize**[{*expr*, *cons*}, *x*] minimizes *expr* subject to the constraints *cons* being satisfied. The constraints can consist of any combination of equations and inequalities.

This finds the minimum subject to the constraint $x \geq 3$.

```
In[39]:= Minimize[{x^2 - 3 x + 6, x ≥ 3}, x]
```

```
Out[39]:= {6, {x → 3}}
```

This finds the maximum within the unit circle.

```
In[40]:= Maximize[{5 x y - x^4 - y^4, x^2 + y^2 ≤ 1}, {x, y}]
```

```
Out[40]:= {2, {x → -1/√2, y → -1/√2}}
```

This finds the maximum within an ellipse. The result is fairly complicated.

```
In[41]:= Maximize[{5 x y - x^4 - y^4, x^2 + 2 y^2 ≤ 1}, {x, y}]
```

```
Out[41]:= {-Root[-811219 + 320160 #1 + 274624 #1^2 - 170240 #1^3 + 25600 #1^4 &, 1],  
{x → Root[25 - 102 #1^2 + 122 #1^4 - 70 #1^6 + 50 #1^8 &, 2],  
y → Root[25 - 264 #1^2 + 848 #1^4 - 1040 #1^6 + 800 #1^8 &, 1]}}
```

This finds the maximum along a line.

```
In[42]:= Maximize[{5 x y - x^4 - y^4, x + y == 1}, {x, y}]
```

```
Out[42]:= {9/8, {x → 1/2, y → 1/2}}
```

Minimize and **Maximize** can solve any *linear programming* problem in which both the objective function *expr* and the constraints *cons* involve the variables x_i linearly only.

Here is a typical linear programming problem.

```
In[43]:= Minimize[{x + 3 y, x - 3 y ≤ 7 && x + 2 y ≥ 10}, {x, y}]
```

```
Out[43]:= {53/5, {x → 44/5, y → 3/5}}
```

In principle, **Minimize** and **Maximize** can also solve any *polynomial programming* problem in which the objective function and the constraints involve arbitrary polynomial functions of the variables. Many important geometrical and other problems can be formulated in this way.

This solves the simple geometrical problem of maximizing the area of a rectangle with fixed perimeter.

```
In[44]:= Maximize[{x y, x + y == 1}, {x, y}]
```

```
Out[44]:= {1/4, {x → 1/2, y → 1/2}}
```

This finds the maximal volume of a cuboid that fits inside the unit sphere.

```
In[45]:= Maximize[{8 x y z, x^2 + y^2 + z^2 ≤ 1}, {x, y, z}]
```

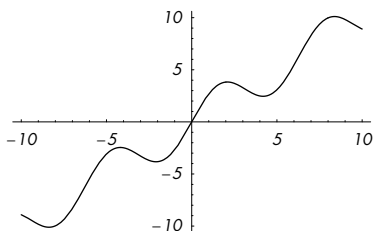
```
Out[45]= { 8/(3 Sqrt[3]), {x → -1/Sqrt[3], y → -1/Sqrt[3], z → 1/Sqrt[3]}}
```

An important feature of **Minimize** and **Maximize** is that they always find *global* minima and maxima. Often functions will have various local minima and maxima at which derivatives vanish, but **Minimize** and **Maximize** use global methods to find absolute minima or maxima, not just local extrema.

Here is a function with many local maxima and minima.

```
In[46]:= Plot[x + 2 Sin[x], {x, -10, 10}];
```

From In[46]:=



Maximize finds the global maximum.

```
In[47]:= Maximize[{x + 2 Sin[x], -10 ≤ x ≤ 10}, x]
```

```
Out[47]= { Sqrt[3] + 8 π/3, {x → 8 π/3} }
```

If you give functions that are unbounded, **Minimize** and **Maximize** will return $-\infty$ and $+\infty$ as the minima and maxima. If you give constraints that can never be satisfied, **Minimize** and **Maximize** will return $+\infty$ and $-\infty$ as the minima and maxima and **Indeterminate** as the values of variables.

One subtle issue is that **Minimize** and **Maximize** allow both *nonstrict* inequalities of the form $x \leq v$, and *strict* ones of the form $x < v$. With nonstrict inequalities there is no problem with a minimum or maximum lying exactly on the boundary $x \rightarrow v$, but with strict inequalities a minimum or maximum must in principle be at least infinitesimally inside the boundary.

With a strict inequality *Mathematica* prints a warning and then returns the point on the boundary.

```
In[48]:= Minimize[{x^2 - 3 x + 6, x > 3}, x]
```

```
Minimize::wksol: Warning: There is no minimum in the region described  
by the constraints; returning a result on the boundary. More...
```

```
Out[48]= {6, {x → 3}}
```

Minimize and **Maximize** normally assume that all variables you give are real. However, by giving a constraint such as $x \in \mathbf{Integers}$, you can specify that a variable must in fact be an integer.

This does maximization only over integer values of x and y .

```
In[49]:= Maximize[{x y, x^2 + y^2 < 120 && (x | y) ∈ Integers}, {x, y}]
```

```
Out[49]:= {56, {x → -8, y → -7}}
```

□ Assuming/Refine

The new function **Refine**[*expr*, *assum*] gives the form of *expr* that would be obtained if symbols in it were replaced by explicit numerical expressions satisfying the assumptions *assum*.

Example: Simplifying a Square Root

```
In[50]:= e = √x y^2 z^4
```

```
Out[50]:= √x y^2 z^4
```

The following refines *e*, assuming that *x* is positive.

```
In[51]:= Refine[e, x > 0]
```

```
Out[51]:= √x √y^2 z^4
```

If *y* is also positive, $\sqrt{y^2}$ simplifies to *y*.

```
In[52]:= Refine[e, x > 0 && y > 0]
```

```
Out[52]:= √x y √z^4
```

```
In[53]:= Clear[e]
```

Example: Using Assumptions in Mathematica Functions

A number of *Mathematica* functions pick up assumptions, (e.g., **Limit** and **Integrate**).

The limit of x^α depends on the domain of α and *x*.

```
In[54]:= Limit[x^α, x → ∞, Assumptions → α > 0]
```

```
Out[54]:= ∞
```

```
In[55]:= Limit[x^α, x → ∞, Assumptions → α < 0]
```

```
Out[55]:= 0
```

Assumptions also propagate through multiple functions.

```
In[56]:= Assuming[σ > 0,
```

$$F = \int_{-\infty}^y \frac{1}{\sqrt{2\pi}\sigma} \text{Exp}\left[-\frac{(x-\mu)^2}{2\sigma^2}\right] dx;$$

```
Limit[F, y → ∞]
```

```
]
```

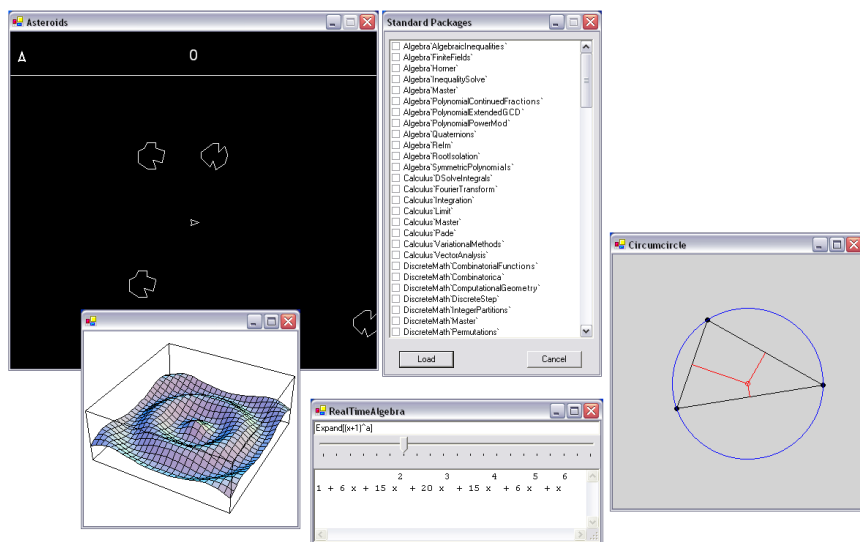
```
Out[56]:= 1
```

■ Import, Export, and Connectivity

□ Connection Technology

.NET/Link Technology Preview

The included .NET/Link technology preview provides full integration with Microsoft's .NET Framework. *Mathematica* users can load any .NET object into *Mathematica* and extend it. .NET/Link also provides an easy way to call any DLL or COM object from within *Mathematica*.



□ Import/Export

A number of graphic, web, and matrix file formats have been added to *Mathematica* 5, bringing the list of supported formats to over 40.

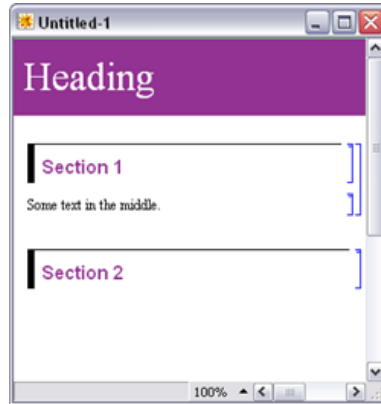
Example: XHTML with CSS

XHTML with CSS allows *Mathematica* users to preserve the look of their *Mathematica* notebooks much more closely when exporting them to the web. The following notebook expression creates a simple notebook in the Classroom style sheet.

```
In[57]:= nb = Notebook[{Cell["Heading", "Title"], Cell["Section 1", "Section"],
  Cell["Some text in the middle.", "Text"],
  Cell["Section 2", "Section"]}, StyleDefinitions -> "Classroom.nb"];
```

This is what the notebook looks like in *Mathematica*.

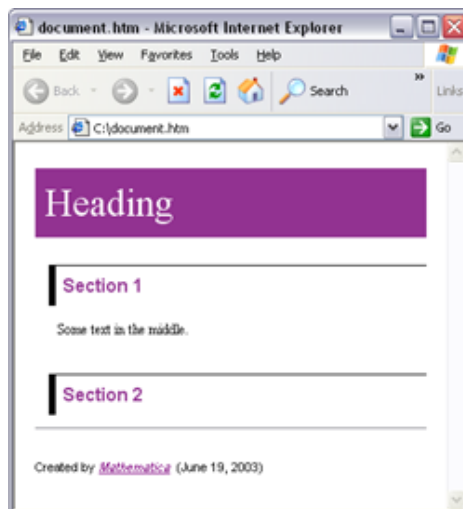
```
In[58]:= NotebookPut[nb];
```



This is the same notebook saved as an XHTML file.

```
In[59]:= Export["document.htm", nb];
```

If you view this file in a web browser, you should see that the look simulates the look inside *Mathematica*.



Example: SVG

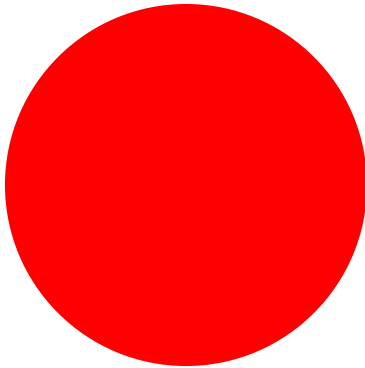
Mathematica 5 supports export to Scalable Vector Graphics (SVG), a language for describing two-dimensional vector and mixed vector/raster graphics in XML. SVG can be viewed with SVG viewers and web browsers with SVG plug-ins.

This defines a red disk in *Mathematica* graphics.

```
In[60]:= g = Show[Graphics[
  {RGBColor[1, 0, 0], Disk[{0, 0}, 1]}, AspectRatio -> Automatic]];

```

From In[60]:=



This exports the SVG to a file ready to view with SVG viewers and web browsers with plug-ins.

```
In[61]:= Export["g.svg", g];

```

```
In[62]:= !! g.svg

```

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
'http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd'>
<svg xmlns='http://www.w3.org/2000/svg'
width='288.0pt'
height='288.0pt'
viewBox='-1.05 -1.05 2.1 2.1'
preserveAspectRatio='xMidYMid meet'>
<g transform='scale(1, -1) translate(0,0)'
style='fill:black; font-family:&apos;Courier New&apos;, Courier,
Symbol, monospace; font-size:0.0729167pt; stroke:black; stroke-
width:0.00182292pt'>
<ellipse fill='#f00'
cx='0'
cy='0'
rx='1.0'
ry='1.0'
stroke='none'/>
</g>
</svg>

```

Example: PNG

Mathematica 5 supports the Portable Network Graphics (PNG) format, a popular bitmap format for the web.

```
In[63]:= owl = Import["OwlAlpha.png"];

```

```
In[64]:= Show[owl, ImageSize -> 144];
```

From In[64]:=



Example: DICOM

Mathematica 5 supports the Digital Imaging and Communications in Medicine (DICOM) Standard. The DICOM Standard facilitates interoperability of medical-imaging equipment and is widely used in radiology.

This imports an image scanned using an MRI scanner.

```
In[65]:= g = Import["MR-MON02-16-knee", "DICOM"];
```

```
In[66]:= Show[g];
```

From In[66]:=



Example: Sparse Arrays

Mathematica 5 supports import and export of files in many standard sparse array formats such as Matrix Market and Harwell-Boeing.

```
In[67]:= s = Import["young1c.mtx", "MatrixMarket"]
```

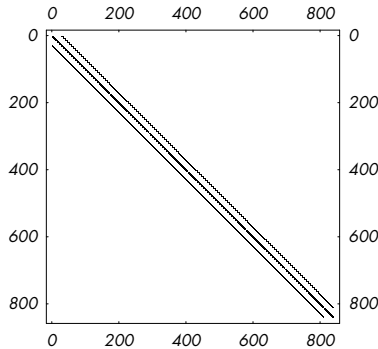
```
Out[67]= SparseArray[<4089>, {841, 841}]
```

The new function **MatrixPlot** can show the structure of the nonzero elements in a sparse array.

```
In[68]:= << "LinearAlgebra`MatrixManipulation`"
```

```
In[69]:= MatrixPlot[s];
```

From In[69]:=



Example: Tabular Import/Export

Many *Mathematica* users are processing increasingly large data sets. Import and export of tabular data has been speeded up by a factor between 10 and 100, depending on the size and structure of the data.

```
In[70]:= data1 = Table[Random[], {5000}, {10}];
```

```
In[71]:= Export["data.dat", data1, "Table"];
```

```
In[72]:= AbsoluteTiming[data2 = Import["data.dat", "Table"];]
```

```
Out[16]= {1.7500112 Second, Null}
```

```
In[73]:= data1 === data2
```

```
Out[73]= True
```

In *Mathematica* 4.2, the preceding took about 12 seconds on the same computer.

■ Other Functions

□ Statistical Plots

The new standard package **StatisticsPlots** offers a variety of plots and charts, including box-and-whisker plots, Pareto plots, and quantile-quantile plots, that are commonly used to gain an overview of data from a statistical perspective.

Example: Box-and-Whisker Plots

This loads the package **StatisticsPlots**.

```
In[1]:= << "Statistics`StatisticsPlots`"
```

The box-and-whisker plot is invaluable for gaining a quick overview of the extent of a numeric data set. It takes the form of a box that spans the distance between two quantiles surrounding the median, typically the 25% quantile to the 75% quantile. Commonly, “whiskers,” lines that extend to span either the full data set or the data set excluding outliers, are added. Outliers are defined as points

beyond $3/2$ the interquantile range from the edge of the box; far outliers are points beyond three times the interquantile range.

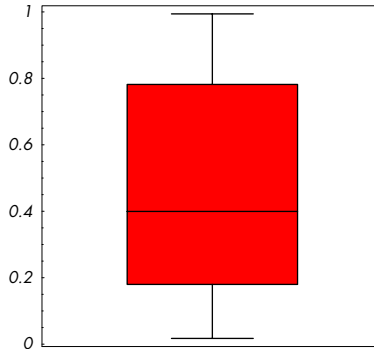
This generates some numbers to plot.

```
In[2]:= dat = Table[Random[], {50}];
anotherdat = Table[Random[], {50}, {3}];
```

In its most basic form the function takes a simple vector of data.

```
In[4]:= BoxWhiskerPlot[dat];
```

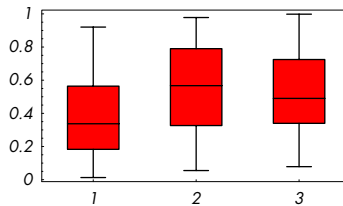
From In[4]:=



When the data is multivariate, a box is produced for each column.

```
In[5]:= BoxWhiskerPlot[anotherdat];
```

From In[5]:=



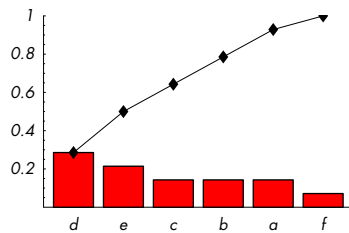
Example: Pareto Plots

The Pareto plot is a quality-control plot that combines a bar chart displaying percentages of categories in the data with a line plot showing cumulative percentages of the categories.

In the most basic form, **ParetoPlot** takes a list of data that is assumed to consist of discrete categories. It determines the frequency of each category in the list, converts the frequencies to percentages, and creates the plot.

```
In[6]:= ParetoPlot[{a, b, c, d, d, d, e, d, e, e, f, a, b, c}];
```

From In[6]:=



In data where the frequencies are precomputed, we can plot it directly by providing *{category, frequency}* pairs instead of the raw data to **ParetoPlot**.

The data quantities have been precomputed for this Pareto plot.

```
In[7]:= ParetoPlot[
  {"Oats", 34.3}, {"Wheat", 72.1}, {"Rye", 10.2}, {"Soy", 68.2}];
```

From In[7]:=



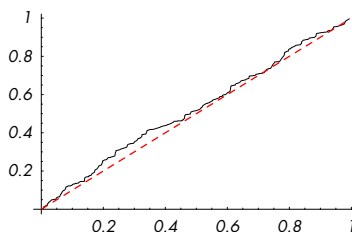
Example: Quantile-Quantile Plots

Quantile-quantile plots are used to determine whether two data sets come from populations with a common distribution. If the points of the plot, which are formed from the quantiles of the data, are roughly on a line with a slope of 1, then the distributions are the same.

Here two data sets are compared. Because they have identical distributions, the plot falls roughly along the reference line.

```
In[8]:= QuantilePlot[Table[Random[], {300}],
  Table[Random[], {300}], SymbolShape -> None, PlotJoined -> True,
  ReferenceLineStyle -> {Hue[0], Dashing[{0.02}]}];
```

From In[8]:=



Example: Pairwise Scatter Plots

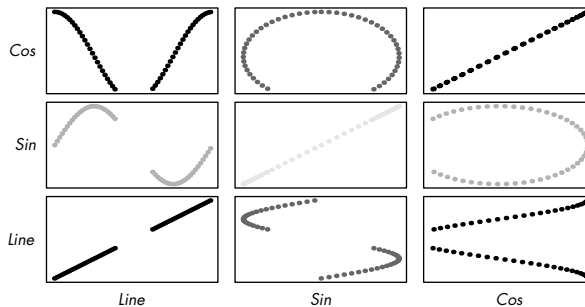
The pairs or matrix scatter plot allows the individual columns in a multivariate set of data to be plotted against each other in order to investigate relationships between the variables. The resulting plot is a matrix of subgraphs.

The following creates a simple data set and plots it with some options.

```
In[9]:= dat = Table[{x, Sin[x], Cos[x]}, {x, 0, 2  $\pi$ , 0.1}];

In[10]:= PairwiseScatterPlot[dat, DataSpacing  $\rightarrow$  0.1,
    PlotStyle  $\rightarrow$  {{GrayLevel[0], GrayLevel[0.7]},
        {GrayLevel[0.4], GrayLevel[0.9]}},
    DataRanges  $\rightarrow$  {All, All, {-0.75, 1}},
    DataLabels  $\rightarrow$  {"Line", "Sin", "Cos"}, AspectRatio  $\rightarrow$  0.5];
```

From In[10]:=



□ Sow and Reap

The functions **Sow** and **Reap** can be used together to accumulate lists of intermediate results in an evaluation.

Example: Mathematica's Function DigitCount

In this example, **Sow** and **Reap** are used to determine the number of times the digit 9 appears in the base-10 representation of a specified integer. The *Mathematica* function **DigitCount** works in a very similar way.

```
In[11]:= dcount[n_Integer, digit_Integer, base_Integer] :=
    First@First[Reap[Apply[Plus, Last[
        Reap[Sow[1, IntegerDigits[n, base]], digit, #2 &]], {1, -1}]]]
```

```
In[12]:= dcount[28093440230911048, 9, 10]
```

```
Out[12]= 2
```

Example: StepMonitor and EvaluationMonitor

Here we accumulate the steps taken by **NDSolve** for an initial value problem.

```

In[13]:= {y1, t1} = Reap[NDSolve[y''[t] == -y[t] && y[0] == 0 && y'[0] == 1,
    y, {t, 0, 3π}, StepMonitor -> Sow[t]]]

Out[13]:= {{y -> InterpolatingFunction[{{0., 9.42478}}, <>]}},
    {{0.000102076, 0.000204151, 0.00378919, 0.00737423, 0.0109593,
    0.0109844, 0.0110096, 0.0110347, 0.0110598, 0.011085, 0.0111353,
    0.0111856, 0.0112358, 0.0117387, 0.0122415, 0.0127444, 0.0132472,
    0.0182757, 0.0233042, 0.0283326, 0.0333611, 0.0383896, 0.0886742,
    0.138959, 0.189244, 0.239528, 0.289813, 0.340097, 0.443103,
    0.546108, 0.649113, 0.752119, 0.855124, 0.95813, 1.06113,
    1.18519, 1.30924, 1.43329, 1.55734, 1.6814, 1.80545, 1.9295,
    2.05355, 2.1776, 2.30166, 2.4383, 2.57495, 2.7116, 2.84824,
    2.98489, 3.12154, 3.25819, 3.39483, 3.53148, 3.66813, 3.80477,
    3.93185, 4.05893, 4.18601, 4.3131, 4.44018, 4.56726, 4.69434,
    4.82142, 4.9485, 5.07558, 5.20266, 5.31387, 5.42508, 5.53629,
    5.64751, 5.75872, 5.86993, 5.98114, 6.09235, 6.2251, 6.35786,
    6.49061, 6.62336, 6.75611, 6.88886, 7.02162, 7.15437, 7.28712,
    7.41987, 7.55263, 7.71329, 7.87396, 8.03463, 8.19529, 8.35596,
    8.51663, 8.6773, 8.83796, 8.99863, 9.1593, 9.29204, 9.42478}}}

In[14]:= yf = y /. First[y1];
    s = Map[{#, yf[#]} &, First[t1]];

```

This visualizes the steps used by **NDSolve**.

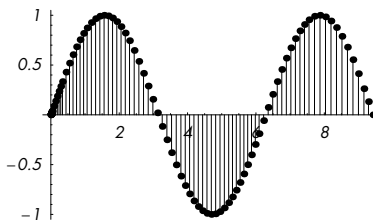
```

In[16]:= << Graphics`MultipleListPlot`

In[17]:= g1 = MultipleListPlot[s, SymbolShape -> Stem];

```

From In[17]:=



This uses **EvaluationMonitor** to visualize the sampling used by **NIntegrate**.

```

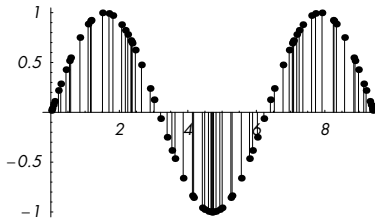
In[18]:= {y1, t1} =
    Reap[NIntegrate[Sin[t], {t, 0, 3π}, EvaluationMonitor -> Sow[t]]]

Out[18]:= {2., {{4.71239, 9.34978, 0.074996, 8.98266, 0.442117, 8.26632, 1.15846,
    7.24987, 2.17491, 6.03012, 3.39466, 2.35619, 4.67489, 0.037498,
    4.49133, 0.221059, 4.13316, 0.579231, 3.62493, 1.08746, 3.01506,
    1.69733, 7.06858, 4.74989, 9.38728, 4.93345, 9.20372, 5.29162,
    8.84555, 5.79985, 8.33732, 6.40972, 7.72745, 8.24668, 7.08733,
    9.40603, 7.17911, 9.31425, 7.3582, 9.13516, 7.61231, 8.88105,
    7.91725, 8.57611, 5.89049, 7.04983, 4.73114, 6.95805, 4.82292,
    6.77897, 5.002, 6.52486, 5.25612, 6.21992, 5.56105, 1.1781,
    2.33745, 0.018749, 2.24567, 0.110529, 2.06658, 0.289616, 1.81247,
    0.543728, 1.50753, 0.848665, 3.53429, 2.37494, 4.69364, 2.46672,
    4.60186, 2.64581, 4.42277, 2.89992, 4.16866, 3.20486, 3.86372}}}

```

```
In[19]:= MultipleListPlot[
  Map[{#, Sin[#]} &, First[t1]], SymbolShape -> Stem];
```

From In[19]:=



□ Timing Functions

The *Mathematica* function **Timing** calculates the amount of CPU time the kernel spends on a calculation. It does not take into account latency or other applications running on the CPU. The function **AbsoluteTiming** provides a type of “wall-clock timing” that measures the total time a command takes until the result is displayed.

Example: Importing a Large Data Set

Timing gives only the time spent by the *Mathematica* kernel.

```
In[20]:= << Optimization`MPSTData`
In[21]:= krnltime = Timing[Import["80bau3b.mps", "MPSTData"];]
Out[21]:= {0.031 Second, Null}
```

Because **Import** uses *MathLink* to connect the kernel to an external converter, the total time for reading this data set is much longer.

```
In[22]:= walltime = AbsoluteTiming[Import["80bau3b.mps", "MPSTData"];]
Out[22]:= {0.9062558 Second, Null}
```

The time spent in the external converter and in transferring the data between the converter and the *Mathematica* kernel plus the operating system spends doing something else, is the difference between the **Timing** and **AbsoluteTiming** results.

```
In[23]:= latency = First@walltime - First@krnltime
Out[23]:= 0.875256 Second
```

□ New Linear Algebra Functions

Mathematica 5 introduces new functionality in the area of linear algebra including generalized eigenvalues, matrix norms, Cholesky decomposition, new singular value operations, characteristic polynomials, and matrix rank.

Example: Norm

The new function **Norm** has been optimized to be fast for machine numbers. Here the 2-norm of a vector of length 100,000 is calculated.

```
In[24]:= tbl = Table[Random[], {100000}];
```

```
In[25]:= Norm[Table[Random[], {100000}], 2] // Timing
```

```
Out[25]:= {0.032 Second, 182.393}
```

Norm can also be used on symbolic vectors. Here the p -norm of a symbolic vector is determined.

```
In[26]:= Norm[{a, b, c}, p]
```

```
Out[26]:= (Abs[a]^p + Abs[b]^p + Abs[c]^p)^(1/p)
```

Here is an expanded version of the above example. In the following, various p -norms of the same symbolic matrix are determined.

```
In[27]:= Norm[{{a, b}, {c, d}}, 1]
```

```
Out[27]:= Max[Abs[a] + Abs[c], Abs[b] + Abs[d]]
```

```
In[28]:= Norm[{{a, b}, {c, d}}, Infinity]
```

```
Out[28]:= Max[Abs[a] + Abs[b], Abs[c] + Abs[d]]
```

```
In[29]:= Norm[{{a, b}, {c, d}}, Frobenius]
```

```
Out[29]:= Sqrt[Abs[a]^2 + Abs[b]^2 + Abs[c]^2 + Abs[d]^2]
```

Here, **Norm** is used to further restrict the definition of itself, and the following definition of the matrix norm is used to obtain the norm of the matrix $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$.

$$\|A\| = \max_{x \neq 0} \frac{\|A \cdot x\|}{\|x\|}$$

```
In[30]:= A = {{1, 2}, {3, 4}};
```

```
FindMaximum[Norm[A.x, 2] / Norm[x, 2], {x, {0.001, 0.001}}]
```

```
Out[31]:= {5.46499, {x -> {0.000832741, 0.00118166}}}
```

□ Algebraic Number Objects

Mathematica 5 brings high-performance arithmetic for algebraic numbers. *Mathematica 5* represents algebraic numbers as **Root** objects. A **Root** object contains the minimal polynomial of the algebraic number and the root number, an integer indicating which of the roots of the minimal polynomial the **Root** object represents.

This loads the package **AlgebraicNumberFields**.

```
In[32]:= << "NumberTheory`AlgebraicNumberFields`"
```

Algebraic objects are now supported.

```
In[33]:= a = Algebraic[#1^3 - #1 + 1 &, {1, 2, 3}, 1]
```

```
Out[33]= Algebraic[1 - #1 + #1^3 &, {1, 2, 3}, 1]
```

Algebraic objects can be used in other functions as well.

```
In[34]:= N[a, 20]
```

```
Out[34]= 3.6151970842505862282
```

Arithmetic can also be carried out with algebraic objects.

```
In[35]:= a = Algebraic[#1^4 + 7 #1 - 21 &, {1, 2, 3, 4}, 1];
```

```
b = Algebraic[#1^4 + 7 #1 - 21 &, {9, 8, 7, 5}, 1];
```

$$\frac{2a^2 - 3ab + 5b^5 - 3}{a^8 - b^4 + \frac{1}{2}} + 9$$

```
Out[37]= Algebraic[-21 + 7 #1 + #1^4 &,
{
41286695899369558776723710439212189982056327290172063
4586375026009762651263976115838375027468985058462049,
6520802026300441952691134470541521717177617572114,
13759125078029287953791928347515125082406955175386147,
3688721281596550115065494536738395724701865336152,
13759125078029287953791928347515125082406955175386147,
2274021184276897634212701763901059483341282983762,
13759125078029287953791928347515125082406955175386147}, 1]
```

The function **ToCommonField** can be used to find a common finite extension of rationals containing the given algebraic numbers.

```
In[38]:= ToCommonField[{√2, √3}]
```

```
Out[38]= {Algebraic[1 - 10 #1^2 + #1^4 &, {0, -9/2, 0, 1/2}, 4],
Algebraic[1 - 10 #1^2 + #1^4 &, {0, 11/2, 0, -1/2}, 4]}
```

Arithmetic within a fixed finite extension of rationals is much faster than arithmetic within the field of all complex algebraic numbers. Here are some definitions.

```
In[39]:= {a, b, c} = {i, √2, Root[#1^3 - 2 #1 + 3 &, 1]};
f = (-2 y z (7 + x - y + z^2) + (6 + x^2 + 2 y) (-11 + x y + z^2)) /
(2 y z (-4 - x + 3 y z) - (6 + x^2 + 2 y) (2 - 2 x + z^3));
```

An integral basis of an algebraic number field K is a list of algebraic numbers forming a basis of the $\mathbb{Z}$ -module of algebraic integers of K . $\{a_1, \dots, a_n\}$ is an integral basis of an algebraic number field K iff $a_i \in K$ are algebraic integers and every algebraic integer $z \in K$ can be uniquely represented as $z = k_1 a_1 + \dots + k_n a_n$ with integer coefficients k_i .

This gives an integral basis of the field generated by the first root of $533 + 429 \#1 + 18 \#1^2 + \#1^3 \&$.

```
In[41]:= IntegralBasis[533 + 429 #1 + 18 #1^2 + #1^3 &, 1]
```

```
Out[41]= {1, Algebraic[533 + 429 #1 + 18 #1^2 + #1^3 &, {0, 1, 0}, 1],  
Algebraic[533 + 429 #1 + 18 #1^2 + #1^3 &, {742/759, 94/759, 1/759}, 1]}
```

The discriminant of a number field K is the discriminant of an integral basis $\{a_1, \dots, a_n\}$ of K (i.e., the determinant of the matrix with elements **NumberFieldTrace** $[a_i, a_j]$). The value of the determinant does not depend on the choice of integral basis.

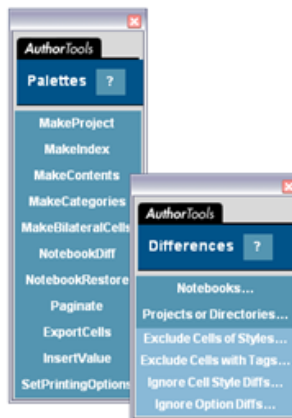
Here is the discriminant of $\mathbb{Q}(2 - \sqrt{3} + 5^{1/4})$.

```
In[42]:= NumberFieldDiscriminant[2 - Sqrt[3] + 5^(1/4)]
```

```
Out[42]= 5184000000
```

□ Authoring and Presentation

AuthorTools, first introduced in *Mathematica* 4.2, has been expanded and now includes tools for operations such as comparing differences between notebooks.



Mathematica 5 also adds a new authoring palette for slide shows and an improved slide show environment for all style sheets.



XML and Mathematica

Pavi Sandhu

Using *Mathematica*, you can import and transform arbitrary XML documents, export notebooks as XML files, and work with popular XML formats such as SVG and MathML. This article gives an overview of *Mathematica*'s support for XML and includes detailed information on the large number of new XML-related functions and options.

■ **Introduction**

Starting with Version 4.2, *Mathematica* includes comprehensive support for XML, the metamarkup language developed by the World Wide Web Consortium (W3C) for describing structured documents and data. Using the new XML features introduced in Version 4.2, you can do any of the following.

- Import any arbitrary Extensible Markup Language (XML) document in the form of a *Mathematica* expression.
- Analyze the contents of an XML document or transform its structure using *Mathematica*'s sophisticated programming and symbolic manipulation abilities.
- Export the resulting expression back as an XML document to share it with other users and applications.
- Save *Mathematica* notebooks in an XML format using the new NotebookML document type definition (DTD) defined for this purpose.
- Import, export, and evaluate equations in MathML—the W3C standard for representing mathematical formulas on the web.
- Export graphics in Scalable Vector Graphics (SVG) format—the W3C standard for representing graphics on the web.

These new features make *Mathematica* a powerful development tool for creating and processing XML documents. They ensure complete interoperability between *Mathematica* and other XML applications and between notebooks and other XML document formats. This article provides detailed information on *Mathematica*'s XML-related capabilities.

□ **Native XML Formats**

Mathematica has built-in support for four XML formats: MathML, SVG, NotebookML, and ExpressionML. More details about each of these formats are given below.

MathML

Mathematical Markup Language (MathML) is an XML format developed by the W3C for describing the structure and symbolic meaning of mathematical formulas. It provides a standard way of displaying mathematical notation in web pages. Version 4.1 of *Mathematica* included limited support for import and export of MathML. In Version 4.2, the support was greatly expanded with many new functions for generating and manipulating MathML and for converting between MathML and the expressions used internally by *Mathematica* to represent mathematics.

The new MathML features make *Mathematica* an excellent tool for authoring and editing MathML content. You can, for example, use *Mathematica*'s powerful typesetting system to create properly formatted equations and then copy and paste them in MathML format into an HTML document for display on the web. You can also import MathML equations from other applications and evaluate them using *Mathematica*.

SVG

SVG is an XML format developed by the W3C for describing two-dimensional graphics. SVG images can be rescaled without loss of resolution and are usually much smaller in size than comparable JPEG or GIF images. SVG files can also be manipulated with a scripting language to produce dynamic and interactive graphics. Using *Mathematica*, you can directly export any graphics present in a notebook, in SVG format.

NotebookML

NotebookML is an XML format developed by Wolfram Research for representing *Mathematica* notebooks. The tags and attributes used in NotebookML are specified by an XML DTD and correspond closely to the structures used in notebook expressions. Using NotebookML, you can save your notebooks as well-formed XML documents and then import them back into *Mathematica* in a completely lossless way.

NotebookML provides a bridge between *Mathematica* and other XML applications, allowing you to use notebooks with standard web technologies such as Cascading Style Sheets (CSS) and Extensible Stylesheet Language Transformations (XSLT). For example, you can display a NotebookML document in a web browser by using a CSS style sheet to specify how each element in the document should be rendered. Support for MathML and SVG is built into NotebookML, in that you can choose to save all equations in a notebook as MathML and all graphics as SVG when saving the notebook as NotebookML.

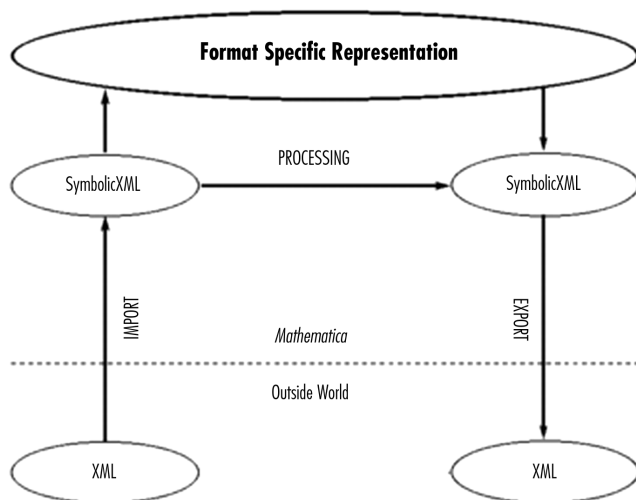
ExpressionML

ExpressionML is a specialized subset of NotebookML. ExpressionML fragments can represent any *Mathematica* expression in an XML format. NotebookML uses ExpressionML fragments to represent *Mathematica* expressions that are embedded within a notebook structure.

□ SymbolicXML

What Is SymbolicXML?

SymbolicXML is the format used by *Mathematica* for representing XML documents. The conversion from XML to SymbolicXML translates the XML document into a *Mathematica* expression, while preserving its structure. Since both XML documents and *Mathematica* expressions have a tree structure, there is a natural mapping from one to the other. You can then manipulate the SymbolicXML expression using the standard techniques of *Mathematica* programming.



You can import XML data into *Mathematica* using the standard **Import** or **ImportString** functions. You can also control various details of the import process, such as how to treat whitespace, whether to recognize entities, or whether to validate against a DTD, by using the conversion options of the **Import** function.

The following command imports an XML data file into *Mathematica*.

```
Import["data.xml", "XML"]
```

Unless the file you are importing is in an XML format recognized by *Mathematica*, such as MathML, the result is a SymbolicXML expression, *expr1*. You can then manipulate the expression using standard *Mathematica* commands to produce another SymbolicXML expression, *expr2*.

expr1 → *expr2*

Finally, you can export the result as an XML file using the standard **Export** function.

```
Export["newdata.xml", expr2, "XML"]
```

You can use conversion options to control various details of the export process, such as the format of the exported XML, using the conversion options of the **Export** function.

The combination of SymbolicXML and *Mathematica* programming provides a useful alternative to other techniques for manipulating XML documents, such as XSLT transformations or the SAX or DOM APIs used with a low-level programming language such as Java. *Mathematica* allows you to achieve the same level of flexibility and control in processing XML documents. The advantage is that you can leverage *Mathematica*'s advanced support for symbolic manipulation and numerical computation to do some very complex and sophisticated transformations that would be difficult or impossible to do using other methods.

For example, you can use pattern-matching techniques to extract specific parts of an XML document, perform numerical computations on the data, and then convert the results into 3D graphics for easy visualization. You can also define transformations to convert one type of XML application to another. For example, you can import a DocBook document as SymbolicXML and then convert it into XHTML format by defining suitable transformation rules to replace one set of element names with another set. For some specific examples of useful applications of SymbolicXML, see Transforming XML.

Support for SymbolicXML is well-integrated with NotebookML, ExpressionML, and MathML. For example, when importing an XML document as SymbolicXML, *Mathematica* recognizes if the document is in NotebookML, ExpressionML, or MathML format and automatically converts it into a notebook expression in the case of NotebookML, an expression in the case of ExpressionML, or a typeset box expression in the case of MathML. You can also override the default behavior and choose to import any of these XML flavors as SymbolicXML if you wish. There is also a large number of kernel functions for quickly and easily converting between strings, boxes, or expressions on the one hand, and NotebookML, MathML, or SymbolicXML on the other.

Note that if you prefer to manipulate XML documents using Java directly, you can still do so using the *J/Link* add-on package. This package integrates *Mathematica* fully with Java, enabling you to call Java commands from *Mathematica* or to call *Mathematica* kernel functions from Java programs. You can, thus, have access to both the computational abilities of *Mathematica* as well as the low-level programming features and classes of Java, and combine the two as needed.

Representing Elements

Each element in an XML document corresponds to an **XMLElement** command in SymbolicXML. An XML expression of the form

```
<element attribute = "value"> data <element>
```

has the following representation in SymbolicXML:

```
XMLElement[element, {attribute → value}, {data}]
```

Each **XMLElement** command has three arguments:

- The first argument specifies the name of the element.
- The second argument specifies the attributes of the element. This argument is a list of zero or more rules, with each rule specifying a single attribute in the form: "attribute" → "value".

- The third argument specifies the actual data contained in the element. This can be raw character data in the form of a string and/or child elements of the element being represented. Each child element is represented by its own **XMLElement** command. You can nest multiple **XMLElement** commands to the level necessary to replicate the nested structure of the original XML expression.

The names of all elements and attributes as well as any character data in the XML document are represented as strings in SymbolicXML. This is to prevent a large number of new symbols from being introduced into the *Mathematica* session, which could lead to possible naming conflicts.

Here is a simple XML fragment.

```
<book type="novel">Moby Dick</book>
```

Here is the representation of this fragment in SymbolicXML.

```
XMLElement["book",{ "type"→"novel"},{"Moby Dick"}]
```

Here is a more complicated XML expression, showing several levels of nesting.

```
<book type='novel'>
  <title>Moby Dick</title>
  <author born='1819' died='1891'>
    <name>
      <first>Herman </first>
      <last>Melville</last>
    </name>
  </author>
</book>
```

Here is the corresponding SymbolicXML expression.

```
XMLElement["book",{ "type"→"novel"},{
  XMLElement["title",{},{ "Moby Dick"}],
  XMLElement["author",{ "born"→"1819", "died"→"1891"},{
    XMLElement["name",{},{
      XMLElement["first",{},{ "Herman"}],
      XMLElement["last",{},{ "Melville"}]}]}]}]
```

Handling Namespaces

If a namespace is specified in an XML element, the syntax of the corresponding SymbolicXML expression is slightly more complex. The exact syntax depends on whether the namespace is specified implicitly, as a default namespace, or explicitly, using a namespace prefix.

Using a default namespace

For any element that lies within a default namespace, the **XMLElement** expression is the same as it would be if no namespace was specified. However, the element in which the default namespace is declared has its **XMLElement** expression modified, as shown in the following example.

Here is a simple XHTML document with a default namespace declared on the **html** element.

```
<html xmlns='http://www.w3.org/1999/xhtml'>
  <head> </head>
  <body>
    <p>Here is some text.</p>
  </body>
</html>
```

Here is the corresponding SymbolicXML expression.

```
XMLElement["html",{{"http://www.w3.org/2000/xmlns/","xmlns"}→
"http://www.w3.org/1999/xhtml"},{
XMLElement["head",{},{}],
XMLElement["body",{},{
XMLElement["p",{},{ "Here is some text."}]}]}]
```

Note that the **XMLElement** expression representing the **html** element has a complex structure. Its second argument is:

```
{"http://www.w3.org/2000/xmlns/","xmlns"}→
"http://www.w3.org/1999/xhtml"}
```

This statement accomplishes two things:

- It identifies the attribute **xmlns** with the namespace defined by the URL **http://www.w3.org/2000/xmlns**, as required by the XML specification.
- It sets the value of the **xmlns** attribute to the URL **http://www.w3.org/1999/xhtml**, thus defining the default namespace.

In other words, when declaring a default namespace on an element, the syntax of the corresponding **XMLElement** structure is:

```
XMLElement[element, {{xmlns-url, "xmlns"} → namespace-url}, {data}]
```

Here *xmlns-url* is the URL associated with the namespace of the **xmlns** attribute, and *namespace-url* is the URL of the default namespace being declared.

Using an explicit namespace prefix

If the namespace is specified explicitly on an element using a namespace prefix, the syntax of the SymbolicXML expression is modified, as shown in the following example.

Here is an XHTML document with some MathML markup embedded in it. The **xmlns:m** attribute in the **math** element binds the MathML namespace to the namespace prefix **m**. All the MathML element names are then written with this namespace prefix attached.

```
<html xmlns='http://www.w3.org/1999/xhtml'>
  <head> <title>Test</title> </head>
  <body>
    <p>Here is some math.</p>
    <p>
      <m:math xmlns:m='http://www.w3.org/1998/Math/MathML'>
        <m:mi>x</m:mi>
        <m:mo>+</m:mo>
```


Declaration

The `XMLObject["Declaration"]` expression is used to represent the XML declaration that typically appears at the start of an XML document. This has the following syntax.

```
XMLObject["Declaration"]["Version"→"1.0", option → value]
```

These two options are allowed.

- **"Standalone"** takes the value **"yes"** if the document references an external DTD and **"no"** otherwise.
- **"Encoding"** specifies the character encoding used in the document. Not all encodings will be honored on export. If an encoding that *Mathematica* cannot export is specified, an error message is produced and the encoding is changed in the document.

Here is a typical XML declaration.

```
<?xml version="1.0" encoding="ascii" standalone="yes"?>
```

Here is the corresponding SymbolicXML expression.

```
XMLObject["Declaration"]["Version"→"1.0", "Encoding"→"ascii"  
"Standalone"→"Yes"]
```

Comment

The `XMLObject["Comment"]` expression is used to represent XML comments. It has the following syntax.

```
XMLObject["comment"]["string"]
```

Here is an example of an XML comment.

```
<!-- Created on 3/6/02. -->
```

Here is the corresponding SymbolicXML expression.

```
XMLObject["Comment"]["Created on 3/6/02."]
```

Document

The most important `XMLObject` is `XMLObject["Document"]`. It is used as a container for the entire document and has the following syntax.

```
XMLObject["Document"][{prolog}, document tree, {epilog}]
```

The prolog may contain an `XMLObject["Declaration"]`, followed by optional processing instructions and DTD declarations. The epilog contains either processing instructions or comments.

Here is an example of a simple document consisting of an XML declaration, a comment, and a single element.

```
<?xml version='1.0'?>  
<!--this is a sample file-->  
<root/>
```

Here is the corresponding SymbolicXML expression.

```
XMLObject["Document"][{XMLObject["Declaration"]["Version"→
"1.0"],XMLObject["Comment"]["this is a sample
file"]}],XMLElement["root",{},{},{}]
```

The only option for **XMLObject["Document"]** is **"Valid"**. This option is set automatically by the parser. If the document was validated on import and validation succeeded, then the option **"Valid"→True** will be included in the **XMLObject** expression. If validation was attempted but failed, then **"Valid"→False** will be included in the **XMLObject**. If validation was not attempted, then the option is omitted from the **XMLObject** expression.

Doctype

The **XMLObject["Doctype"]** expression is used to represent XML document type declarations. It has the following syntax.

```
XMLObject["Doctype"][name, option → value]
```

These three options are allowed.

- **"System"** specifies a DTD in the local file system, either as a relative pathname or a URL.
- **"Public"** specifies a standardized name that is used to publicly identify the DTD.
- **"Internal"** specifies an internal DTD subset. Its value is a string that contains the data in the internal DTD subset.

Here is a Doctype declaration with all three options.

```
<!DOCTYPE catalog PUBLIC "-//F00//DTD catalog 1.1//EN"
"www.foo.com/example/catalog.dtd"
[internal DTD stuff]>
```

Here is the corresponding SymbolicXML expression.

```
XMLObject["Doctype"]["catalog", "Public"→"-//F00//DTD catalog 1.1//EN",
"System"→"www.foo.com/example/catalog.dtd", "InternalSubset"→"internal
DTD stuff"]
```

For more details on XML Doctype declarations, see the W3C XML specification at www.w3.org/TR/REC-xml.

ProcessingInstruction

The **XMLObject["ProcessingInstruction"]** expression is used to represent XML processing instructions. This has the following syntax.

```
XMLObject["ProcessingInstruction"][target string, optional data string]
```

It is common to use attribute-like syntax in processing instructions. These pseudo-attributes are not parsed but are returned as raw strings. Here is a processing instruction that specifies a style sheet.

```
<?xml-stylesheet href="mystyle.css" type="text/css"?>
```

Here is the corresponding SymbolicXML expression. Notice that the double quotes around the attribute values are escaped to distinguish them from the double quotes around the argument as a whole.

```
XMLObject["ProcessingInstruction"]["xml-stylesheet",
"href=\"mystyle.css\" type=\"text/css\""]
```

CDATASection

The `XMLObject["CDATASection"]` expression is used to represent CDATA sections. CDATA is a W3C abbreviation for Character Data. CDATA sections are used in an XML document as a wrapper for raw character data to avoid having to escape special characters such as " and <. These characters would normally have to be indicated as `"e;` and `<`, respectively. CDATA sections are used in XML to enclose character data that would require a lot of escaping, such as programs or math expressions.

Here is a simple fragment from an XML document containing a CDATA section.

```
<![CDATA[ 5 < 7 << 2*10^123]]>
```

Here is the corresponding SymbolicXML expression.

```
XMLObject["CDATASection"][" 5 < 7 << 2*10^123"]
```

By default, `CDATASection` object wrappers are not preserved on import, and only the contents of the CDATA section are retained. To preserve the `CDATASection` wrappers, you must explicitly set the conversion option, `"PreserveCDATASections" → True`. More information about conversion options is given below.

■ Importing XML

□ Functions for Importing XML

Import

You can import XML data into *Mathematica* using the standard **Import** function, which has the following syntax.

```
Import[file]
```

```
Import[file, format]
```

The first argument of the function specifies the file to be imported. You can also specify an optional second argument to control the form of the output. For importing XML data, the relevant file formats are: **"XML"**, **"NotebookXML"**, **"ExpressionML"**, **"MathML"**, and **"SymbolicXML"**.

With **"XML"** as the import format, any XML formats that *Mathematica* does not recognize are returned as SymbolicXML. The formats that *Mathematica* does recognize—NotebookXML, ExpressionML, and MathML—are treated slightly differently. A NotebookML file is imported as a notebook expression. An Express-

sionML file is imported as the corresponding cell expression. A MathML file is returned as the corresponding box expression.

With "**SymbolicXML**" as the import format, even NotebookML, ExpressionML, and MathML files are imported as SymbolicXML. This gives you a way to override the XML format from being automatically interpreted.

Here is a file containing the MathML encoding for x^2 .

`In[1]:= !! eqn.mml`

```
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <semantics>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <annotation-xml encoding="MathML-Content">
      <apply>
        <power/>
        <ci>x</ci>
        <cn type="integer">2</cn>
      </apply>
    </annotation-xml>
  </semantics>
</math>
```

If you import the file and specify "**XML**" as the second argument, the equation is converted into a box expression.

`In[2]:= Import["eqn.mml", "XML"]`

```
Out[2]= FormBox[TagBox[TagBox[SuperscriptBox[x, 2],
  MathMLPresentationTag, AutoDelete → True], AnnotationsTagWrapper[
  TagBox[SuperscriptBox[x, 2], MathMLContentTag, AutoDelete → True]],
  AutoDelete → True], TraditionalForm]
```

If you specify "**SymbolicXML**" as the second argument, the equation is imported as a SymbolicXML box expression.

`In[3]:= Import["eqn.mml", "SymbolicXML"]`

```
Out[3]= XMLObject[Document][{ },
  XMLElement[math, {{http://www.w3.org/2000/xmlns/, xmlns} →
    http://www.w3.org/1998/Math/MathML},
    {XMLElement[semantics, { }, {XMLElement[msup, { },
      {XMLElement[mi, { }, {x}], XMLElement[mn, { }, {2}]}], XMLElement[
        annotation-xml, {encoding → MathML-Content}, {XMLElement[apply,
          { }, {XMLElement[power, { }, {}], XMLElement[ci, { }, {x}],
            XMLElement[cn, {type → integer}, {2}]}]}]}], { }]
```

If **Import** is used with only one argument, *Mathematica* processes the data in the file based on its file extension. Any file with a **.xml** extension is imported as

XML. This means that if it is in one of the XML formats explicitly supported by *Mathematica*, namely NotebookML, ExpressionML, or MathML, the file will be interpreted in the appropriate way. All other XML formats are imported as SymbolicXML.

Mathematica also recognizes the `.mml` extension for MathML files and the `.nbml` extension for NotebookML files. In the following example, we import a file with the `.mml` extension.

```
In[4]:= Import["eqn.mml"]
```

```
Out[4]= FormBox[TagBox[TagBox[SuperscriptBox[x, 2],  
MathMLPresentationTag, AutoDelete → True], AnnotationsTagWrapper[  
TagBox[SuperscriptBox[x, 2], MathMLContentTag, AutoDelete → True]],  
AutoDelete → True], TraditionalForm]
```

You can display the above box expression as conventional mathematical notation by using **DisplayForm**.

```
In[5]:= DisplayForm[%]
```

```
Out[5]//DisplayForm=  
x2
```

You can control the various details of the import process, such as how to treat whitespace, whether to recognize entities, or whether to validate against a DTD, by specifying conversion options to the **Import** function.

ImportString

You can also use the standard **ImportString** function to import XML data from a string. This is useful when you want to generate XML data directly within *Mathematica* instead of importing the data from an external file. The **ImportString** function has the following syntax.

```
ImportString[string, format]
```

With `"XML"` as the import format, any XML formats that *Mathematica* does not recognize are returned as SymbolicXML. Here is an example of a simple XML expression converted to SymbolicXML using **ImportString**.

```
In[6]:= ImportString[  
"<person gender='male'><name>Joe Smith</name></person>", "XML"]
```

```
Out[6]= XMLObject[Document][{ }, XMLElement[person,  
{gender → male}, {XMLElement[name, { }, {Joe Smith}]}], { }]
```

The formats that *Mathematica* does recognize—NotebookML, ExpressionML, and MathML—are treated slightly differently. A NotebookML file is imported as a notebook expression. An ExpressionML file is imported as the corresponding cell expression. A MathML file is returned as the corresponding box expression. Here is an example of importing a simple MathML expression. Notice that the MathML markup is automatically converted to a *Mathematica* box expression.

```
In[7]:= ImportString["<math><mi>x</mi><mo>+</mo><mn>1</mn></math>", "XML"]
Out[7]= FormBox[TagBox[RowBox[{x, +, 1}],
  MathMLPresentationTag, AutoDelete -> True], TraditionalForm]
```

You can stop the automatic interpretation of imported files based on file extension by specifying **"SymbolicXML"** as the second argument. In this example, the imported file is returned as SymbolicXML rather than the usual box expression.

```
In[8]:= ImportString[
  "<math><mi>x</mi><mo>+</mo><mn>1</mn></math>", "SymbolicXML"]
Out[8]= XMLObject[Document][{}, XMLElement[math, {}, {XMLElement[mi, {}, {x}],
  XMLElement[mo, {}, {+}], XMLElement[mn, {}, {1}]}], {}]
```

You can control the various details of the import process, such as how to treat whitespace, whether to recognize entities, or whether to validate against a DTD, by specifying conversion options to the **ImportString** function.

XMLGet

The **XMLGet** function can be used to import an XML document as SymbolicXML. It is very similar to the **Import** function, in that **XMLGet[file]** is equivalent to **Import[file, "SymbolicXML"]**. The advantage of using **XMLGet** is that, unlike **Import**, it can retrieve files over the web. Hence, it is useful if you want to import an XML file posted at a URL.

For example, the following command retrieves stock quotes from a website and returns the data as SymbolicXML.

```
In[9]:= XML`Parser`XMLGet[
  "http://www.webserviceX.net/stockquote.asmx/GetQuote?symbol=IBM"]
Out[9]= XMLObject[Document][
  {XMLObject[Declaration][Version -> 1.0, Encoding -> utf-8]},
  XMLElement[string, {{http://www.w3.org/2000/xmlns/, xmlns} ->
    http://www.webserviceX.NET/},
    {<StockQuotes><Stock><Symbol>IBM</Symbol><Last>88.51</Last><
      Date>1/14/2003</Date><Time>4:00pm</Time><Change>+1.00</
      Change><Open>87.51</Open><High>88.51</High><Low>87.22</
      Low><Volume>7304300</Volume><MktCap>149.6B</MktCap><
      PreviousClose>87.51</PreviousClose><PercentageChange>
      +1.14%</PercentageChange><AnnRange>54.01 - 120.55</
      AnnRange><Earnings>3.18</Earnings><P-E>27.52</P-E><Name>
      INTL BUS MACHINE</Name></Stock></StockQuotes>}], {}]
```

Note that **XMLGet** exists only in the **XML`Parser`** context. Hence, you must use the full name of the function, **XML`Parser`XMLGet**, when doing an evaluation. To use the function without the context name prefix, you must first add the **XML`Parser`** context to your context path.

XMLGet also accepts an optional second argument, which specifies a preinitialized parser object.

XMLGet[file, XMLParserObject]

Initializing the parser involves loading a DTD into memory either from a URL or a local file. This only needs to be done once in each kernel session. Subsequent references to the DTD are then processed much faster because the DTD has already been read and parsed.

You can also specify conversion options for **XMLGet** to control its behavior. The conversion options for **XMLGet** are the same as the ones for **Import**. However, the syntax for specifying conversion options is slightly different. The conversion options can be specified directly in the **XMLGet** function, such that

```
XMLGet[file, option1 → value1, option1 → value2, ...]
```

is equivalent to

```
Import[file, "SymbolicXML", ConversionOptions → {option1 → value1, option2 → value2, ...}]
```

XMLGetString

The **XMLGetString** function can be used to import an XML string as Symbolic-XML. It is very similar to the **ImportString** function, in that **XMLGet**': **String**[string] is equivalent to **ImportString**[string, "SymbolicXML"].

```
In[10]:= XML`Parser`XMLGetString[
  "<math><mi>x</mi><mo>+</mo><mn>1</mn></math>"]
Out[10]= XMLObject[Document][{ }, XMLElement[math, { }, {XMLElement[mi, { }, {x}],
  XMLElement[mo, { }, {+}], XMLElement[mn, { }, {1}]}], { }]
```

Note that **XMLGetString** exists only in the **XML`Parser`** context. Hence, you must use the full name of the function, **XML`Parser`XMLGet**, when doing an evaluation. To use the function without the context name prefix, you must first add the **XML`Parser`** context to your context path.

The advantage of using **XMLGetString** is that it accepts a preinitialized parser object as its second argument.

```
XMLGetString[file, XMLParserObject]
```

Initializing the parser involves loading a DTD into memory either from a URL or a local file. This only needs to be done once in each kernel session. Subsequent references to the DTD are then processed much faster because the DTD has already been read and parsed.

For example, the following command loads the XML package and then preinitializes the parser according to the XHTML DTD located at the specified URL. In this example, the preinitialized parser is given the name **XHTMLParser**.

```
In[11]:= XHTMLParser = XML`Parser`InitializeXMLParser["html",
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"]
Out[11]= XML`Parser`XMLParser[html,
  http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd]
```

Now that the parser is initialized, we import an XML string. The string is validated with respect to the DTD stored in **XHTMLParser** by setting **"Validate"**:

AgainstDTD→**True**. The option **Valid**→**True** in the output indicates that the XML input string was valid XML with respect to the XHTML DTD.

```
In[12]:= XML`Parser`XMLGetString[
  "<html><head><title>test</title></head><body><p>Here is some text.</p></body></html>", XHTMLParser, "ValidateAgainstDTD" → True]

Out[12]:= XMLObject[Document] [{XMLObject[Doctype] [html, System →
  http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd]},
  XMLElement[html, {}, {XMLElement[head, {},
    {XMLElement[title, {}, {test}]}], XMLElement[body, {},
    {XMLElement[p, {}, {Here is some text.}]}]}], {}, Valid → True]
```

You can also specify conversion options for **XMLGetString** to control the various details of the import process. The conversion options for **XMLGetString** are the same as those for **ImportString**. As with **XMLGet**, the conversion options can be specified directly in the **XMLGetString** function, that is:

XMLGetString[*string*, *option1* → *value1*, *option1* → *value2*, ...]

is equivalent to

ImportString[*string*, "SymbolicXML", **ConversionOptions** → {*option1* → *value1*, *option1* → *value2*, ...}]

For more information on the conversion options available for importing XML, see Import Conversion Options.

□ Entities and Validation

An XML document can contain any characters included in the Unicode character set. When importing an XML document into *Mathematica*, all numeric Unicode character entity references are automatically resolved into the corresponding *Mathematica* character.

```
In[13]:= ImportString["<hand>
  <card suit='&#9824;' value='2' />
  <card suit='&#9825;' value='10' />
  <card suit='&#9826;' value='6' />
  <card suit='&#9827;' value='4' />
  <card suit='&#9827;' value='5' />
</hand>", "XML"]

Out[13]:= XMLObject[Document] [{},
  XMLElement[hand, {}, {XMLElement[card, {suit → ♠, value → 2}, {}],
    XMLElement[card, {suit → ♥, value → 10}, {}],
    XMLElement[card, {suit → ♦, value → 6}, {}],
    XMLElement[card, {suit → ♣, value → 4}, {}],
    XMLElement[card, {suit → ♣, value → 5}, {}]}], {}]
```

Other entities that are not built into XML are resolved according to the rules present in the DTD.

```
In[14]:= ImportString["<!DOCTYPE root [<!ELEMENT root (
    #PCDATA)><!ENTITY WRI \"Wolfram Research, Inc.\">]>
    <root>&WRI;</root>", "XML"]

Out[14]:= XMLObject[Document] [
    {XMLObject[Doctype] [root, InternalSubset → <!ELEMENT root (
        #PCDATA)*><!ENTITY WRI 'Wolfram Research, Inc.'>]],
    XMLObject[Element] [root, {}, {Wolfram Research, Inc.}], {},
    Valid → True]
```

In addition to simply converting an XML document to a SymbolicXML expression, **Import** can validate the XML data to ensure that it conforms to a content model defined by a DTD. So long as the document is well formed, a SymbolicXML expression will be returned. If the document is not valid, warning messages will be issued and the document wrapper will indicate the invalid nature of the document with the option **Valid**→**False**.

You can control the various aspects of how entities are treated and whether the document is validated or not by using the conversion options for the **Import** function.

□ Conversion Options

Introduction

The standard **ConversionOptions** feature of **Import** gives you more control over the import process. The syntax for specifying a conversion option is as follows.

```
Import[file, ConversionOptions → {option → value}]
```

Multiple conversion options can be specified by making the right-hand side of **ConversionOptions** a list of lists. There are nine conversion options available for importing XML data.

- "NormalizeWhitespace"
- "AllowRemoteDTDAccess"
- "AllowUnrecognizedEntities"
- "ReadDTD"
- "ValidateAgainstDTD"
- "IncludeDefaultedAttributes"
- "IncludeEmbeddedObjects"
- "IncludeNamespaces"
- "PreserveCDATASections"

NormalizeWhitespace

This conversion option controls how whitespace in the document being imported is processed. Whitespace is defined as a space, tab, or newline character.

If `"NormalizeWhitespace"→True`, all the whitespace inside an element is normalized. This means that all leading and trailing whitespace is stripped and any interior whitespace is reduced to a single whitespace character. `"NormalizeWhitespace"→True` is the default setting for this option.

If `"NormalizeWhitespace"→False`, then all whitespace is preserved as it was in the original XML document.

If `"NormalizeWhitespace"→Automatic`, then ignorable whitespace is removed and nonignorable whitespace is preserved. Whitespace is ignorable when it occurs in places where character data is not permitted according to the content model specified by the DTD. The primary use of ignorable whitespace is to add indentation for formatting purposes.

<i>option</i>	<i>value</i>	<i>effect</i>
"NormalizeWhitespace"	True	all the whitespace inside an element is normalized
	False	all whitespace in the original XML document is preserved
	Automatic	ignorable whitespace is removed and nonignorable whitespace is preserved

The option `"NormalizeWhitespace"` and its possible values.

Here is an example of whitespace handling with the default setting `"NormalizeWhitespace"→True`.

```
In[15]:= ImportString["<foo>  a b </foo>", "XML"]
Out[15]= XMLObject[Document][{}, XMLElement[foo, {}, {a b}], {}]
```

Setting `"NormalizeWhitespace"→False` preserves the whitespace as it appears in the original string.

```
In[16]:= ImportString["<foo>  a b </foo>", "XML",
  ConversionOptions→{"NormalizeWhitespace"→False}]
Out[16]= XMLObject[Document][{}, XMLElement[foo, {}, { a b }], {}]
```

Note: If the option `"NormalizeWhitespace"→False` is specified, pattern matching on the resulting SymbolicXML expression may become problematic because of the intervening whitespace.

AllowRemoteDTDAccess

This conversion option controls whether the parser may access the network in order to retrieve DTDs. If `"AllowRemoteDTDAccess"→True`, the parser will automatically access the network to retrieve DTDs. If `"AllowRemoteDTDAc`.

cess"→**False**, then remote DTDs will not be retrieved, but local DTDs may still be used.

<i>option</i>	<i>value</i>	<i>effect</i>
"AllowRemoteDTDAccess"	True	the parser will automatically access the network to retrieve DTDs
	False	remote DTDs will not be retrieved, but local DTDs can still be used

The option **"AllowRemoteDTDAccess"** and its possible values.

If **"AllowRemoteDTDAccess"**→**False** and the document refers to a remote DTD, the parse will fail and an error message will be generated unless the conversion option **"ReadDTD"** is also set to **False**.

AllowUnrecognizedEntities

This option determines what the parser will do if undefined entity references are encountered in the XML document.

<i>option</i>	<i>value</i>	<i>effect</i>
"AllowUnrecognizedEntities"	True	any undefined entities are wrapped in special entity delimiter characters and no error messages are reported
	False	an error message is reported and the parse fails
	Automatic	an error message is reported for any unrecognized entity, and the entity is wrapped in special entity delimiter characters

The option **"AllowUnrecognizedEntities"** and its possible values.

The following examples contain an undefined entity called **'dogs'**. If **"AllowUnrecognizedEntities"**→**False**, then an error message is reported and the parse fails.

```
In[17]:= ImportString["<!DOCTYPE root [<!ELEMENT
      root (#PCDATA)><!ENTITY cats 'Himalayans'>]>
<root>
  &dogs;
</root>", "XML",
  ConversionOptions -> {"AllowUnrecognizedEntities" -> False}]
```

```
XML'Parser'XMLGet::parseError :
Entity 'dogs' was not found at Line: 1 Character: 88 in /tmp/m0000083951
```

```
Import::probableDTDError :
There was an error parsing the XML document. This may be
due to undeclared entities in the XML document, or an error
loading the DTD. Check that entities are correctly declared
or use the "AllowUnrecognizedEntities" conversion option.
```

```
Out[17]= $Failed
```

If **"AllowUnrecognizedEntities"→Automatic**, an error message is reported for any unrecognized entity, and the entity is wrapped in special entity delimiter characters. However, this does not interrupt the importing and parsing of the XML data. **Automatic** is the default setting for this option.

```
In[18]:= ImportString["<!DOCTYPE root [<!ELEMENT
      root (#PCDATA)><!ENTITY cats 'Himalayans'>]>
<root>
  &dogs;
</root>", "XML",
  ConversionOptions -> {"AllowUnrecognizedEntities" -> Automatic}]
```

```
XML'Parser'XMLGet::nonFatalParseError :
Entity 'dogs' was not found at Line: 1 Character: 88 in /tmp/m0000093951
```

```
Out[18]= XMLObject[Document][{XMLObject[Doctype][root, InternalSubset ->
  <!ELEMENT root (#PCDATA)*><!ENTITY cats 'Himalayans'>]],
  XMLElement[root, {}, { dogs }], {}, Valid -> False]
```

If **"AllowUnrecognizedEntities"→True**, then any undefined entities are wrapped in special entity delimiter characters and no error messages are reported.

```
In[19]:= ImportString["<!DOCTYPE root [<!ELEMENT
      root (#PCDATA)><!ENTITY cats 'Himalayans'>]>
<root>
  &dogs;
</root>", "XML",
  ConversionOptions -> {"AllowUnrecognizedEntities" -> True}]
```

```
Out[19]= XMLObject[Document][{XMLObject[Doctype][root, InternalSubset ->
  <!ELEMENT root (#PCDATA)*><!ENTITY cats 'Himalayans'>]],
  XMLElement[root, {}, { dogs }], {}, Valid -> True]
```

ReadDTD

This conversion option determines whether an external DTD subset is read or not. The most important uses of a DTD are to define a content model for validation and to define character entities.

<i>option</i>	<i>value</i>	<i>effect</i>
"ReadDTD"	True	external DTDs are read
	False	external DTDs are ignored

The option **"ReadDTD"** and its possible values.

Since reading the DTD can directly affect the contents of the document, **"ReadDTD"→True** is the default setting for this option. Setting **"ReadDTD"→False** can improve the efficiency, but you should only make this change if you are certain that no information is required from the DTD. **"ReadDTD"** is ignored if you are using a preinitialized parser.

Setting **"ReadDTD"→False** is the only way to prevent the parser from attempting to read the DTD. Setting **"AllowRemoteDTDAccess"→False** will prevent network access and setting **"ValidateAgainstDTD"→False** will prevent validation from happening, but neither of these options will prevent an error caused by the parser failing to read the DTD.

ValidateAgainstDTD

This conversion option determines whether the XML document is validated or not.

<i>option</i>	<i>value</i>	<i>effect</i>
"ValidateAgainstDTD"	True	a validation attempt will be made on import even if there is no DOCTYPE declaration in the XML document
	False	no validation attempt will be made on import
	Automatic	a validation attempt will be made on import only if there is a DOCTYPE declaration in the XML document

The option **"ValidateAgainstDTD"** and its possible values.

If the document is valid, the parser will set the `XMLObject["Document"]` option `"Valid"→True`. If the document is invalid, the parser will generate validity error messages and will set `"Valid"→False`.

The following is an example of trying to parse a document that is not valid by setting `"ValidateAgainstDTD"→True`. The parser generates error messages because the document is not valid.

```
In[20]:= XML`Parser`XMLGetString["<!DOCTYPE root [
  <!ELEMENT root EMPTY>
]>
<root>
  <child/>
</root>", "ValidateAgainstDTD" → True] // InputForm
```

```
XML`Parser`XMLGetString::nonFatalParseError :
  No character data is allowed by content model at Line: 1 Character: 52
```

```
XML`Parser`XMLGetString::nonFatalParseError :
  Unknown element 'child' at Line: 1 Character: 60
```

```
XML`Parser`XMLGetString::nonFatalParseError :
  No character data is allowed by content model at Line: 1 Character: 61
```

```
General::stop :
  Further output of XML`Parser`XMLGetString::nonFatalParseError will
  be suppressed during this calculation.
```

```
Out[20]//InputForm=
```

```
XMLObject["Document"][[
  {XMLObject["Doctype"]["root",
    "InternalSubset" ->
    " <!ELEMENT root EMPTY> "]],
  XMLElement["root", {}],
  {XMLElement["child", {}, {}]}], {},
  "Valid" -> False]
```

On the other hand, if the document is valid, then no messages are generated and `"Valid"→True` is included in the output.

```
In[21]:= XML`Parser`XMLGetString["<!DOCTYPE root [
  <!ELEMENT root EMPTY>
]>
<root></root>", "ValidateAgainstDTD" → True] // InputForm
```

```
Out[21]//InputForm=
```

```
XMLObject["Document"][[
  {XMLObject["Doctype"]["root",
    "InternalSubset" ->
    " <!ELEMENT root EMPTY> "]],
  XMLElement["root", {}, {}], {}, "Valid" -> True]
```

Parsing both of the examples above with **"ValidateAgainstDTD"→False** generates no error messages, nor does it add a **"Valid"** option to **XMLObject["Document"]**.

```
In[22]:= XML`Parser`XMLGetString["<!DOCTYPE root [
<!ELEMENT root EMPTY>
]>
<root>
  <child/>
</root>", "ValidateAgainstDTD" → False] // InputForm

Out[22]//InputForm=
XMLObject["Document"][[
{XMLObject["Doctype"]["root",
  "InternalSubset" ->
  "<!ELEMENT root EMPTY> "]},
XMLElement["root", {},
{XMLElement["child", {}, {}]}], {}]

In[23]:= XML`Parser`XMLGetString["<!DOCTYPE root [
<!ELEMENT root EMPTY>
]>
<root></root>", "ValidateAgainstDTD" → False] // InputForm

Out[23]//InputForm=
XMLObject["Document"][[
{XMLObject["Doctype"]["root",
  "InternalSubset" ->
  "<!ELEMENT root EMPTY> "]},
XMLElement["root", {}, {}], {}]
```

With **"ValidateAgainstDTD"→True**, validation is attempted even if there is no DOCTYPE declaration.

```
In[24]:= XML`Parser`XMLGetString["<root/>", "ValidateAgainstDTD" → True]
```

```
XML`Parser`XMLGetString::nonFatalParseError :
Unknown element 'root' at Line: 1 Character: 8
```

```
XML`Parser`XMLGetString::nonFatalParseError :
Root element different from DOCTYPE at Line: 1 Character: 8
```

```
Out[24]= XMLObject[Document][{}, XMLElement[root, {}, {}], {}, Valid → False]
```

To have validation on only when there is a DOCTYPE Declaration, use the default setting **"ValidateAgainstDTD"→Automatic**. In the following example, no DTD is specified so the parser does not attempt to validate the XML string.

```
In[25]:= ImportString["<root/>", "XML"]

Out[25]= XMLObject[Document][{}, XMLElement[root, {}, {}], {}]
```

Here the parser tries to validate the input string because a DTD is specified explicitly.

```
In[26]:= ImportString["<!DOCTYPE root [<!ELEMENT root EMPTY>]>
<root>
  <child/>
</root>", "XML"] // InputForm
```

```
XML`Parser`XMLGet::nonFatalParseError :
No character data is allowed by content
model at Line: 1 Character: 50 in /tmp/m0000123951
```

```
XML`Parser`XMLGet::nonFatalParseError :
Unknown element 'child' at Line: 1 Character: 58 in /tmp/m0000123951
```

```
XML`Parser`XMLGet::nonFatalParseError :
No character data is allowed by content
model at Line: 1 Character: 59 in /tmp/m0000123951
```

```
General::stop :
Further output of XML`Parser`XMLGet::nonFatalParseError will be
suppressed during this calculation.
```

```
Out[26]//InputForm=
XMLObject["Document"][[
{XMLObject["Doctype"]["root",
"InternalSubset" ->
"<!ELEMENT root EMPTY>"}],
XMLElement["root", {},
{XMLElement["child", {}, {}]}], {},
"Valid" -> False]
```

Note that even when using a preinitialized parser, **"ValidateAgainstDTD"→Automatic** will not validate unless there is a DOCTYPE declaration in the document.

IncludeDefaultedAttributes

This conversion option determines whether attributes that are specified by the DTD as default attributes are included in the SymbolicXML expression. **"IncludeDefaultedAttributes"→False** is the default setting for this option because the default values for the attributes will be known to application developers, and, therefore, it is unnecessary to include the values in the SymbolicXML expression. Setting **"IncludeDefaultedAttributes"→True** will include them in the SymbolicXML expression.

<i>option</i>	<i>value</i>	<i>effect</i>
"IncludeDefaultedAttributes"	True	default attributes in the DTD are included in the SymbolicXML expression
	False	default attributes are not included

The option **"IncludeDefaultedAttributes"** and its possible values.

Here is a simple example to illustrate how this option works. For brevity, let us assign a variable to represent the XML fragment.

```
In[27]:= xmlString = "<!DOCTYPE test [<!ELEMENT test EMPTY>
<!ATTLIST test
\tMovie\t\tCDATA\t\t#FIXED\t\t\"The Lord of The Rings.\"
>]>
<test/>";
```

This converts the XML fragment into SymbolicXML.

```
In[28]:= ImportString[xmlString, "SymbolicXML"]

Out[28]:= XMLObject[Document] [
  {XMLObject[Doctype] [test, InternalSubset → <!ELEMENT test EMPTY> <
    !ATTLIST test      Movie CDATA #FIXED 'The Lord of The Rings.' >]],
  XMLElement[test, {}, {}], {}, Valid → True]
```

If you want the default attributes to be included in the imported SymbolicXML, set **"IncludeDefaultedAttributes" → True**.

```
In[29]:= ImportString[xmlString, "SymbolicXML",
  ConversionOptions → {"IncludeDefaultedAttributes" → True}]

Out[29]:= XMLObject[Document] [
  {XMLObject[Doctype] [test, InternalSubset → <!ELEMENT test EMPTY> <
    !ATTLIST test      Movie CDATA #FIXED 'The Lord of The Rings.' >]],
  XMLElement[test, {Movie → The Lord of The Rings.}, {}],
  {}, Valid → True]
```

Though default attributes are defined in a DTD, including them in the expression is not the same as validation; thus, default attributes can be included even with **"ValidateAgainstDTD" → False**.

```
In[30]:= ImportString[xmlString, "SymbolicXML", ConversionOptions →
  {"IncludeDefaultedAttributes" → True, "ValidateAgainstDTD" → False}]

Out[30]:= XMLObject[Document] [
  {XMLObject[Doctype] [test, InternalSubset → <!ELEMENT test EMPTY> <
    !ATTLIST test      Movie CDATA #FIXED 'The Lord of The Rings.' >]],
  XMLElement[test, {Movie → The Lord of The Rings.}, {}], {}]
```

IncludeEmbeddedObjects

This conversion option determines the treatment of comments and processing instructions that occur inside the document tree.

<i>option</i>	<i>value</i>	<i>effect</i>
"IncludeEmbeddedObjects"	All	all the embedded objects will be included in the document tree
	None	no embedded objects are included
	Comments	only embedded comments are included
	ProcessingInstructions	only embedded processing instructions are included

The option **"IncludeEmbeddedObjects"** and its possible values.

As before, we set a variable to represent a simple XML fragment to facilitate further examples. **"IncludeEmbeddedObjects"→All** will include all embedded objects in the body of the XML document.

```
In[31]:= xmlexample = "<!--Comment-->
<?Instruction?>
<root>
<blank/>
</root>";
```

With **"IncludeEmbeddedObjects"→All**, all the embedded objects will be included in the document tree.

```
In[32]:= XML`Parser`XMLGetString[xmlexample, "IncludeEmbeddedObjects" → All]
```

```
Out[32]:= XMLObject[Document][{XMLObject[Comment][Comment],
XMLObject[ProcessingInstruction][Instruction]},
XMLElement[root, {}, {XMLElement[blank, {}, {}]}], {}]
```

Since comments and processing instructions are not intended to affect applications that use the XML document, they are usually excluded from the document tree. Including them runs the risk of hampering pattern matching. Hence, the default setting is **"IncludeEmbeddedObjects"→None**.

```
In[33]:= XML`Parser`XMLGetString[xmlexample, "IncludeEmbeddedObjects" → None]
```

```
Out[33]:= XMLObject[Document][{XMLObject[Comment][Comment],
XMLObject[ProcessingInstruction][Instruction]},
XMLElement[root, {}, {XMLElement[blank, {}, {}]}], {}]
```

ProcessingInstructions and Comments

Setting `"IncludeEmbeddedObjects" → "ProcessingInstructions"` or `"IncludeEmbeddedObjects" → "Comments"` will include only the embedded processing instructions or comments, respectively. You can also set `"IncludeEmbeddedObjects" → {"Comments", "ProcessingInstructions"}` so that a list of the embedded comments and processing instructions will be included.

```
In[34]:= XML`Parser`XMLGetString[xmlexample,
      "IncludeEmbeddedObjects" → "ProcessingInstructions"]

Out[34]:= XMLObject[Document] [{XMLObject[Comment] [Comment] ,
      XMLObject[ProcessingInstruction] [Instruction]},
      XMLElement[root, {}, {XMLElement[blank, {}, {}]}], {}]

In[35]:= XML`Parser`XMLGetString[xmlexample,
      "IncludeEmbeddedObjects" → "Comments"]

Out[35]:= XMLObject[Document] [{XMLObject[Comment] [Comment] ,
      XMLObject[ProcessingInstruction] [Instruction]},
      XMLElement[root, {}, {XMLElement[blank, {}, {}]}], {}]

In[36]:= XML`Parser`XMLGetString[xmlexample,
      "IncludeEmbeddedObjects" → {"Comments", "ProcessingInstructions"}]

Out[36]:= XMLObject[Document] [{XMLObject[Comment] [Comment] ,
      XMLObject[ProcessingInstruction] [Instruction]},
      XMLElement[root, {}, {XMLElement[blank, {}, {}]}], {}]
```

IncludeNamespaces

This conversion option determines how namespaces are handled.

<i>option</i>	<i>value</i>	<i>effect</i>
"IncludeNamespaces"	True	specifies the explicit namespace for each element and attribute
	False	no namespace information is reported
	Automatic	the namespace is determined by scoping
	Unparsed	used for handling documents that use namespaces in a nonstandard way

The option `"IncludeNamespaces"` and its possible values.

We will set a variable to represent a simple XML fragment with namespaces to facilitate our examples.

```
In[37]:= xmlString = "<root xmlns='http://mynamespace.com'
      xmlns:same='http://mynamespace.com\'
      xmlns:foo='http://anothernamespace.com'>
      <child attr1='a' same:attr2='b' foo:attr3='c' />
      <foo:child />
      <same:child />
    </root>"
```

True

"IncludeNamespaces"→True reports the namespace information for each element and attribute via a list of the form $\{\textit{namespace}, \textit{localname}\}$. This form is verbose, but it is more faithful to the data model of the XML document. Additionally, this form may be easier to use for pattern matching.

```
In[38]:= XML'Parser'XMLGetString[xmlString, "IncludeNamespaces" → True]

Out[38]:= XMLObject[Document] [{}, XMLElement[{http://mynamespace.com, root},
  {{http://www.w3.org/2000/xmlns/, xmlns} → http://mynamespace.com,
  {http://www.w3.org/2000/xmlns/, same} → http://mynamespace.com\,
  {http://www.w3.org/2000/xmlns/, foo} →
  http://anothernamespace.com},
  {XMLElement[{http://mynamespace.com, child},
    {{}, attr1} → a, {http://mynamespace.com\, attr2} → b,
    {http://anothernamespace.com, attr3} → c}, {}],
  XMLElement[{http://anothernamespace.com, child}, {}, {}],
  XMLElement[{http://mynamespace.com\, child}, {}, {}]}], {}]
```

False

"IncludeNamespaces"→False only reports the local name of each element or attribute. While this setting should not be used in any serious XML application, it is useful for applications that have only a single namespace because it makes the SymbolicXML expression easier to read. Note that the names of all the child elements appear to be identical when parsed this way. Consequently, this option value cannot be trusted whenever multiple namespaces are used.

```
In[39]:= XML'Parser'XMLGetString[xmlString, "IncludeNamespaces" → False]

Out[39]:= XMLObject[Document] [{}, XMLElement[root,
  {xmlns → http://mynamespace.com, same → http://mynamespace.com\,
  foo → http://anothernamespace.com},
  {XMLElement[child, {attr1 → a, attr2 → b, attr3 → c}, {}],
  XMLElement[child, {}, {}], XMLElement[child, {}, {}]}], {}]
```

Automatic

With the default value **"IncludeNamespaces"→Automatic**, the namespace is determined by means of scoping. If the namespace of an element is the same as the default namespace, then the name is represented as a single string for the local name. If the namespace of an element is different, then the name is represented by a list with the structure $\{\textit{namespace}, \textit{localname}\}$.

Here, we see that the only element whose name is represented by a two-string list is the one in namespace **http://anothernamespace.com**. The other elements are implicitly contained in the **http://mynamespace.com** namespace.

Attributes are not compacted because according to the W3C specification, the attributes and the elements have different namespace scoping.

```
In[40]:= XML`Parser`XMLGetString[xmlString]

Out[40]= XMLObject[Document][{ },XMLElement[root,
  {{http://www.w3.org/2000/xmlns/,xmlns}→http://mynamespace.com,
  {http://www.w3.org/2000/xmlns/,same}→http://mynamespace.com\,
  {http://www.w3.org/2000/xmlns/,foo}→
    http://anothernamespace.com},
  {XMLElement[child,{attr1→a,{http://mynamespace.com\,attr2}→b,
    {http://anothernamespace.com,attr3}→c},{ }},
    XMLElement[{http://anothernamespace.com,child},{ },{ }],
    XMLElement[{http://mynamespace.com\,child},{ },{ }]}],{ }]
```

Unparsed

Because the XML namespace recommendation, which extends XML, was made after the initial XML recommendation, there are some documents that use names in a non-namespace-compliant fashion. **"IncludeNamespaces"→"Unparsed"** is provided to allow parsing of these documents. With this value, the name is always represented as a single string: the exact string that appears in the XML file. Unless absolutely necessary, this option value should not be used.

```
In[41]:= XML`Parser`XMLGetString[xmlString,"IncludeNamespaces"→"Unparsed"]

Out[41]= XMLObject[Document][{ },
  XMLElement[root,{xmlns→http://mynamespace.com,
    xmlns:same→http://mynamespace.com\,
    xmlns:foo→http://anothernamespace.com},
  {XMLElement[child,{attr1→a,same:attr2→b,foo:attr3→c},{ }},
    XMLElement[foo:child,{ },{ }],XMLElement[same:child,{ },{ }]}],{ }]
```

PreserveCDATASections

This option controls whether the distinction between CDATA sections and regular character data is maintained on import. The value can be either **True** or **False**. CDATA sections are meant as a convenience for document authors; for most applications, they should not be treated differently from ordinary data. This means that preserving CDATA sections can make pattern matching difficult. For this reason, **"PreserveCDATASections"→False** is the default setting.

<i>option</i>	<i>value</i>	<i>effect</i>
"PreserveCDATASections"	True	information about CDATA sections is preserved
	False	information about CDATA sections is removed

The option **"PreserveCDATASections"** and its possible values.

Here is an example of the default behavior of **PreserveCDATASections**.

```
In[42]:= XML`Parser`XMLGetString["<root><![CDATA[Johnson&Johnson]]></root>"]
```

```
Out[42]:= XMLObject[Document][{ }, XMLElement[root, { }, {Johnson&Johnson}], { }]
```

To preserve CDATA sections, specify **"PreserveCDATASections"→True**.

```
In[43]:= XML`Parser`XMLGetString["<root><![CDATA[Johnson&Johnson]]></root>",
    "PreserveCDATASections" → True]
```

```
Out[43]:= XMLObject[Document][{ }, XMLElement[root,
    { }, {XMLObject[CDATASection][Johnson&Johnson]}], { }]
```

■ Transforming XML

□ Introduction

Mathematica is uniquely suited for processing symbolic expressions because of its powerful pattern matching abilities and its large collection of built-in structural manipulation functions. This section provides a few examples to illustrate the use of *Mathematica* for processing XML data.

When you import an arbitrary XML document into *Mathematica*, it is automatically converted into a SymbolicXML expression. The advantage of converting XML data into SymbolicXML is that you can directly manipulate SymbolicXML using any of *Mathematica*'s built-in functions.

The following command converts an XML string into a SymbolicXML expression.

```
In[1]:= data = ImportString[
    "<colors><blue/><blue/><red/><blue/><red/></colors>", "XML"]
```

```
Out[1]:= XMLObject[Document][{ },
    XMLElement[colors, { }, {XMLElement[blue, { }, { }],
        XMLElement[blue, { }, { }], XMLElement[red, { }, { }],
        XMLElement[blue, { }, { }], XMLElement[red, { }, { }]}], { }]
```

Here we use a simple transformation rule to remove the unwanted "red" element from the list.

```
In[2]:= newdata = data /. XMLElement["red", __] → Sequence[]
```

```
Out[2]:= XMLObject[Document][{ },
    XMLElement[colors, { }, {XMLElement[blue, { }, { }],
        XMLElement[blue, { }, { }], XMLElement[blue, { }, { }]}], { }]
```

You can use **ExportString** to convert SymbolicXML into native XML syntax, which was designed to be easy to read.

```
In[3]:= ExportString[newdata, "XML"]
```

```
Out[3]:= <colors>
    <blue/>
    <blue/>
    <blue/>
</colors>
```

□ Visualizing the XML Tree

Many XML tools display an XML document as a collapsible tree, where the nodes correspond to the elements of the document. This example shows how to produce a similar visualization using cell grouping in a *Mathematica* notebook.

We will do this by recursively traversing the `SymbolicXML` expression, and for each `XMLElement` object, creating a `CellGroupData` expression that contains cells for each of that `XMLElement` object's attributes and children. Each nested `CellGroupData` will be indented from the previous one. We start with the function to process an `XMLElement` object.

```
In[4]:= XMLNote[XMLElement[tag_, attributes_, data_], m_Integer] :=
  Cell[CellGroupData[{
    Cell[TextData[
      StyleBox[tag, FontFamily → "Swiss",
        FontWeight → "Bold", FontSize → 15]],
      Sequence@@(XMLNote[#1, m] &)/@attributes,
      Sequence@@(XMLNote[#1, m + 30] &)/@data
    ], Open],
    CellMargins → {{m, Inherited}, {Inherited, Inherited}}]
```

Notice that we use the integer `m` for indentation. When we map `XMLNote` onto the `XMLElement` object's children, we pass a larger value for `m`, increasing the indentation for the child elements.

A `CellGroupData` expression contains a list of cells. In the above definition, we have only created a cell for the `XMLElement` `x`. However, we have then mapped `XMLNote` onto the attribute list. Since this returns a list, we need to use `Apply[Sequence]` to the result in order to merge that list into the `CellGroupData` expression's list of cells. We then do the same thing to the children of the `XMLElement`.

However, we have not yet defined `XMLNote` to work on attributes. The attributes of an `XMLElement` object are stored in `SymbolicXML` as rules. In most cases, the rule contains two strings: the key and the value. However, when namespaces are involved, the first element of the rule may be a list containing two strings: the namespace and the key. We will need to make two definitions to handle the attributes.

```
In[5]:= XMLNote[{an_String, a_String} → v_String, m_Integer] :=
  Cell[TextData[
    StyleBox[an, FontColor → Hue[0.6]], " ",
    StyleBox[a, FontWeight → "Bold"], " = ",
    StyleBox[v, Background → GrayLevel[0.8]],
    CellMargins → {{m + 5, Inherited}, {Inherited, Inherited}}]

In[6]:= XMLNote[a_String → v_String, m_Integer] :=
  Cell[TextData[
    StyleBox[a, FontWeight → "Bold"], " = ",
    StyleBox[v, Background → GrayLevel[0.8]],
    CellMargins → {{m + 5, Inherited}, {Inherited, Inherited}}]
```

We will need one more definition in order to process simple SymbolicXML expressions. The text nodes in an XML document are stored simply as **String** objects in SymbolicXML. Thus, we need a definition that handles **String** objects.

```
In[7]:= XMLNote[s_String, m_Integer] :=
  Cell[s, Background → GrayLevel[0.9],
    CellMargins → {{m + 25, Inherited}, {Inherited, Inherited}}]
```

With these definitions in place, we can construct a simple notebook to visualize a basic XML document.

```
In[8]:= d = ImportString[ExportString[ $\frac{1}{\sqrt{2}}$ , "MathML"], "SymbolicXML"];

In[9]:= NotebookPut@
  Notebook[{XMLNote[d[[2]], 0]}, CellGrouping → "Manual"]

NotebookObject[<<Untitled-1>>]
```

Because the default value of the option "**IncludeEmbeddedObjects**" is **None**, we did not alter comments, processing instructions, or anything else that would be stored in an **XMLObject**. Adding definitions for these is not difficult and would be a good exercise in processing SymbolicXML.

The notebook produced as the result of the above evaluation is shown here.



□ Manipulating XML Data

XML applications are used for more than just document layout. XML is also an excellent format for storing structured data. Many commercial database vendors are now adding XML support to their products, allowing you to work with databases using XML as an intermediate format.

Mathematica is well suited for extracting and manipulating information from XML documents. To illustrate this, let us manipulate an XML file containing data on major league baseball players. You can download this file from www.mathematica-journal.com.

We first import this file into *Mathematica* as a *SymbolicXML* expression.

```
In[10]:= AL = Import["ALHitters.xml"];
```

Each player's information is stored in a **PlayerRecord** element. We can easily extract this with **Cases**.

```
In[11]:= players = Cases[AL, XMLElement["PlayerRecord", _, _], Infinity];
```

```
In[12]:= Length[players]
```

```
Out[12]= 294
```

As we can see, the XML document contains records for 294 players. Since we do not want to sift through all the American League hitters, we will just take a look at the Yankees. Inside each **PlayerRecord** element, there is a **TEAM** element that specifies a player's team. By passing a slightly more sophisticated pattern to **Cases**, we can extract a list of all players on the Yankees team.

```
In[13]:= yankees = Cases[AL,
    XMLElement["PlayerRecord", _,
        {___, XMLElement["TEAM", _, {"NYY"}], ___}], Infinity];
```

```
In[14]:= Length[yankees]
```

```
Out[14]= 21
```

The variable **yankees** now contains a list of *SymbolicXML* expressions for all the Yankees players. To see the syntax of each **PlayerRecord**, we extract the first element of **yankees**.

```
In[15]:= First[yankees]
```

```
Out[15]= XMLElement[PlayerRecord, {},
  {XMLElement[PLAYER, {}, {Bernie Williams}],
   XMLElement[TEAM, {}, {NY}], XMLElement[POS, {}, {OF}],
   XMLElement[STATUS, {}, {A}], XMLElement[GP, {}, {141}],
   XMLElement[GS, {}, {140}], XMLElement[AB, {}, {537}],
   XMLElement[AVG, {}, {0.307}], XMLElement[R, {}, {108}],
   XMLElement[H, {}, {165}], XMLElement[Doubles, {}, {37}],
   XMLElement[Triples, {}, {6}], XMLElement[HR, {}, {30}],
   XMLElement[RBI, {}, {121}], XMLElement[SH, {}, {0}],
   XMLElement[SF, {}, {3}], XMLElement[BB, {}, {71}],
   XMLElement[SO, {}, {84}], XMLElement[HBP, {}, {5}],
   XMLElement[SB, {}, {13}], XMLElement[CS, {}, {5}],
   XMLElement[GIDP, {}, {0}], XMLElement[OBP, {}, {0.391}],
   XMLElement[Slg, {}, {0.566}], XMLElement[TB, {}, {304}],
   XMLElement[ParkAdj, {}, {0.97882}], XMLElement[Outs, {}, {372}],
   XMLElement[TBxOBP, {}, {118.9}], XMLElement[RP, {}, {116.3}],
   XMLElement[BJRC, {}, {124.5}], XMLElement[PJRP, {}, {111.2}],
   XMLElement[RPPER27Outs, {}, {8.44}],
   XMLElement[BJRCper27Outs, {}, {9.04}],
   XMLElement[PJRPper27Outs, {}, {8.07}],
   XMLElement[Salary, {}, {12357143}],
   XMLElement[DollarsPerRun, {}, {106210}],
   XMLElement[DollarsPerRunCreated, {}, {99230}],
   XMLElement[DollarsPerRunProduced, {}, {111145}],
   XMLElement[PictureIndex, {}, {8190}]}]
```

We can see that the player's name is stored in the **PLAYER** element of each **PlayerRecord** element. Suppose we just want to look at the names of the Yankees hitters we have already extracted. We can extract the name from one **PlayerRecord** easily enough.

```
In[16]:= First[Cases[First[yankees],
  XMLElement["PLAYER", _, {name_}] → name, 2]]
```

```
Out[16]= Bernie Williams
```

We can then use **Map** to extract all the names from **yankees**.

```
In[17]:= Map[
  First[Cases[#,
    XMLElement["PLAYER", _, {name_}] → name, 2]] &,
  yankees]

Out[17]= {Bernie Williams, Derek Jeter, David Justice, Jorge Posada, Paul O'Neill,
  Tino Martinez, Chuck Knoblauch, Jose Canseco, Scott Brosius,
  Luis Polonia, Shane Spencer, Glenallen Hill, Clay Bellinger,
  Jose Vizcaino, Luis Sojo, Chris Turner, Ryan Thompson,
  Alfonso Soriano, Lance Johnson, Felix Jose, Roberto Kelly}
```

Alternately, we could have used **Cases** on **yankees** with an appropriate pattern.

```

In[18]:= Cases[yankees,
  XMLElement["PlayerRecord", _,
    {___, XMLElement["PLAYER", _, {name_}], ___}] → name]

Out[18]= {Bernie Williams, Derek Jeter, David Justice, Jorge Posada, Paul Oneill,
  Tino Martinez, Chuck Knoblauch, Jose Canseco, Scott Brosius,
  Luis Polonia, Shane Spencer, Glenallen Hill, Clay Bellinger,
  Jose Vizcaino, Luis Sojo, Chris Turner, Ryan Thompson,
  Alfonso Soriano, Lance Johnson, Felix Jose, Roberto Kelly}

```

SymbolicXML is a general-purpose format for expressing arbitrary XML data. In some cases, you may find it more useful to convert SymbolicXML into a different type of *Mathematica* expression. This type of conversion is easy to do using pattern matching. In the following example, we import an XML file containing data about baseball pitchers and translate the resulting SymbolicXML expression into a list of *Mathematica* rules. You can download the data file used here from www.mathematica-journal.com.

```

In[19]:= NL = Import["NLPitchers.xml"];

In[20]:= myTransform[XMLElement[_, _, children_]] :=
  Apply[Sequence, Map[myTransform, children]];

In[21]:= myTransform[s_String] := s;

In[22]:= myTransform[XMLObject["Document"][_, root_, _]] :=
  {myTransform[root]};

In[23]:= myTransform[PlayerAttribute[XMLElement[name_, _, {value_}]]] :=
  name → value;

In[24]:= myTransform[XMLElement["PlayerRecord", _, children_]] :=
  Pitcher @@ Map[myTransform[PlayerAttribute[#]] &, children];

In[25]:= First[myTransform[NL]]

Out[25]= Pitcher[PLAYER → Randy Johnson, TEAM → ARI, POS → SP,
  GP → 35, GS → 35, STATUS → A, W → 19, L → 7, S → 0, IP → 248.2,
  ERA → 2.64, ParkAdj → 0.97441, PAERA → 2.57, _90ERA → 5.94,
  _50ERA → 4.54, RPer9inn → 3.4, RP → 93.0, RunRatio → 0.64,
  PAERAWinsOver162 → 123.461, Games → 27.6, PAERAWins → 21.0,
  PAERALoses → 6.6, AVG → 0.213, OBA → 0.276, ShutOut → 3, H → 202,
  R → 89, ER → 73, HR → 23, K → 347, WP → 5, BLK → 2, HBP → 6, E → 3,
  Salary → 13350000, DollarsPerRun → 143563, PictureIndex → 7762]

```

Here, we have transformed the SymbolicXML expression for a **PlayerRecord** node into a simpler expression. All the information about the player is stored in a list of *Mathematica Rule* with **Pitcher** as the head.

In addition to transforming the data into a different expression syntax as above, we can also modify the data and leave the overall expression in SymbolicXML. This way we can alter our data, but still export it to an XML file for use with other applications. As an example, we will work with the salaries of our American League hitters. First, we delete any **PlayerRecord** entries where the salary is not available.

```
In[26]:= AL = AL /. XMLElement["PlayerRecord", __, {___,
    XMLElement["Salary", {}, {"#N/A"}], ___}] -> Sequence[];
```

Next, we create a function to extract name–salary pairs from our **PlayerRecord** data. We will then sort these pairs by salary and look at the top ten.

```
In[27]:= Salaries[x_] := Cases[x,
    XMLElement["PlayerRecord", __, {___,
        XMLElement["PLAYER", __, {name_}], ___,
        XMLElement["Salary", {}, {s_}], ___}] ->
    {name, ToExpression[s]}, Infinity]

In[28]:= rich = Take[Sort[Salaries[AL], (#1[[2]] > #2[[2]]) &], 10] // TableForm

Out[28]//TableForm=
  Albert Belle          12868670
  Bernie Williams      12357143
  Mo Vaughn            11166667
  Raul Mondesi         10000000
  Derek Jeter          10000000
  Rafael Palmeiro      8620921
  Ivan Rodriguez       8600000
  Jim Thome            8175000
  Juan Gonzalez        7500000
  Kenny Lofton         7500000
```

As a simple example of how to change the data in our SymbolicXML expression, we will create a function that doubles players' salaries.

```
In[29]:= DoubleSalary[s_] := ToString[2*ToExpression[s]]

In[30]:= AL = AL /. XMLElement["Salary", {}, {sal_}] ->
    XMLElement["Salary", {}, {DoubleSalary[sal]}];

In[31]:= richer = Take[Sort[Salaries[AL], (#1[[2]] > #2[[2]]) &], 10] // TableForm

Out[31]//TableForm=
  Albert Belle          25737340
  Bernie Williams      24714286
  Mo Vaughn            22333334
  Raul Mondesi         20000000
  Derek Jeter          20000000
  Rafael Palmeiro      17241842
  Ivan Rodriguez       17200000
  Jim Thome            16350000
  Juan Gonzalez        15000000
  Kenny Lofton         15000000
```

□ Visualizing XML Data

Creating a 3D Graphic from an XML File

The following example illustrates how to use *Mathematica* programming and SymbolicXML to visualize data in XML format. The molecule description markup language (MoDL) is an XML application that describes molecules. For

details, see www.oasis-open.org/cover/modl.html. In this example, we convert a MoDL description of the methane molecule into a *Mathematica* 3D graphic.

The following is the MoDL file, which contains the description of the methane molecule. You can download this file from www.mathematica-journal.com.

In[32]:= !! methane.xml

```
<?xml version="1.0"?>
<modl>
<head animation="on" clockperiod="5s" stepsize="1s" loop="true">
  <meta name="title" content="Methane Dance" />
  <DEFINE name="C">
    <atom radius="0.3" color="1 1 0" />
  </DEFINE>
  <DEFINE name="H">
    <atom radius="0.25" color="1 0 0" />
  </DEFINE>
  <DEFINE name="CH4">
    <molecule>
      <atom type="C" id="c1" position="0 0 0.4" />
      <atom type="H" id="h1" position="0.5 0.1 -0.4" />
      <atom type="H" id="h2" position="-1 0.1 0.3" />
      <atom type="H" id="h3" position="0.3 0.7 1" />
      <atom type="H" id="h4" position="0.2 -0.9 0.8" />
      <bond atom1="c1" atom2="h1" />
      <bond atom1="c1" atom2="h2" />
      <bond atom1="c1" atom2="h3" />
      <bond atom1="c1" atom2="h4" />
    </molecule>
  </DEFINE>
</head>
<body>
  <molecule type="CH4" id="m" />

  <TRANSLATE object="m" t="0.2" position="-3 1 -2" />
  <TRANSLATE object="m" t="0.4" position="-1 0 -3" />
  <TRANSLATE object="m" t="0.6" position="1 -1 -1" />
  <TRANSLATE object="m" t="0.8" position="1 0 1" />
  <TRANSLATE object="m" t="1" position="0 0 0" />

  <ROTATE object="m" t="0.2" axis="1 0 0" />
  <ROTATE object="m" t="0.4" axis="0 0 1" angle="-1.571"/>
  <ROTATE object="m" t="0.6" axis="0 -1 0" />
  <ROTATE object="m" t="0.8" axis="-1 0 0" angle="-0.78" />
  <ROTATE object="m" t="1" axis="0 0 1" />
</body>
</modl>
```

Here, we import the file into *Mathematica* in the form of a SymbolicXML expression.

In[33]:= methane = Import["methane.xml"];

In order to convert the resulting SymbolicXML expression into a **Graphic3D** expression, we will need the standard package **Graphics`Shapes`**.

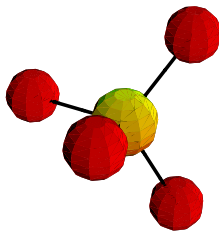
```
In[34]:= << Graphics`Shapes`
```

The following code defines a function called **MoDLToGraphics3D** that turns the SymbolicXML expression into a **Graphics3D** expression. This function relies on a number of auxiliary functions that are defined in a later part of this section, which deals with the details of implementation.

```
In[35]:= MoDLToGraphics3D[modl_] :=
  Block[{
    defs = Cases[modl, XMLElement["DEFINE", __], Infinity],
    body = First[Cases[modl, XMLElement["body", __], Infinity]],
    moldef, themols, theatoms},
  ProcessDefinition/@defs;
  themols =
  Flatten@Cases[body,
    XMLElement["molecule", {___, "type" → t_, ___}, __] > moldef[t],
    Infinity];
  theatoms = Cases[body,
    XMLElement["atom", {___, "type" → t_, ___, "position" → p_, ___},
      __] > Append[moldef[t], MolStringListToList[p]], Infinity];
  Graphics3D[MolToGraphics /@ Join[themols, theatoms], Boxed → False]
]
```

Applying this function to the original SymbolicXML expression generates a 3D graphic representing the methane molecule.

```
Show[MoDLToGraphics3D[metthane]]
```



```
-Graphics3D-
```

The details of implementation of the **MoDLToGraphics3D** function, which performs the actual transformation from SymbolicXML to a 3D graphic, are provided here.

Implementation Details

Notice that the original MoDL file contains a head and a body. In the head, a number of definitions are made that are used throughout the body. We have extracted these definitions into the variable **defs**. We then map the function **ProcessDefinition** across the list of definitions. The function **ProcessDefinition**:

tion constructs a *Mathematica* expression out of a definition and stores it in the variable **moldef**, which is dynamically scoped inside of **MoDLToGraphics3D**.

A **DEFINE** element in the head typically defines either an atom or a molecule. First, consider an atom definition.

```
In[36]:= First[Cases[methane,
  XMLElement["DEFINE", _, {XMLElement["atom", __]}],
  Infinity]]

Out[36]:= XMLElement[DEFINE, {name → C},
  {XMLElement[atom, {radius → 0.3, color → 1 1 0}, {}]}]
```

The **DEFINE** element essentially associates a unique key (in this case **C**) to an atom element. The **atom** element specifies its color and radius. We will turn this into a *Mathematica* expression of the form **Atom[radius, color]**. We will then store it in **moldef[name]**, where **name** is the key specified in the **name** attribute of the **DEFINE** element.

```
In[37]:= ProcessDefinition[XMLElement["DEFINE", {"name" → name_String},
  {XMLElement["atom", {a__}, {}]}]] :=
  (moldef[name] = Atom[GetRad[a], GetColor[a]]);
```

In this case, **a** is the entire sequence of attributes of the **atom** element. The **GetRad** and **GetColor** functions, which we define later, extract the radius and color from this sequence. For now, assume that **GetRad** returns a number and that **GetColor** returns an **RGBColor** expression. We now need to process definitions of molecule elements. Like the atom definitions, molecules are given a unique key in the name attribute of the **DEFINE** element. The **molecule** element then contains **atom** elements and **bond** elements.

```
In[38]:= First[Cases[methane,
  XMLElement["DEFINE", _, {XMLElement["molecule", __]}],
  Infinity]]

Out[38]:= XMLElement[DEFINE, {name → CH4}, {XMLElement[molecule, {,
  {XMLElement[atom, {type → C, id → c1, position → 0 0 0.4}, {}],
  XMLElement[atom, {type → H, id → h1, position → 0.5 0.1 -0.4}, {}],
  XMLElement[atom, {type → H, id → h2, position → -1 0.1 0.3}, {}],
  XMLElement[atom, {type → H, id → h3, position → 0.3 0.7 1}, {}],
  XMLElement[atom, {type → H, id → h4, position → 0.2 -0.9 0.8}, {}],
  XMLElement[bond, {atom1 → c1, atom2 → h1}, {}],
  XMLElement[bond, {atom1 → c1, atom2 → h2}, {}],
  XMLElement[bond, {atom1 → c1, atom2 → h3}, {}],
  XMLElement[bond, {atom1 → c1, atom2 → h4}, {}]}]}]
```

The **atom** elements contain three attributes: **type**, **id**, and **position**. The **type** attribute references the key from previous atom definitions. The **id** attribute is a unique key for this instance of the type of atom defined. In other words, what was defined previously in the atom definitions were types of atoms, like carbon or hydrogen. The atom elements inside a molecule element represent a distinct atom of some previously defined type.

The **molecule** element also contains **bond** elements. These have two attributes: **atom1** and **atom2**. These reference the **id** of the **atom** elements in that molecule expression.

When we call **ProcessDefinition** on a molecule definition, we will want to store a list of the atoms and bonds in **moldef**.

```
In[39]:= ProcessDefinition[XMLElement["DEFINE", {"name" → name_String},
  {XMLElement["molecule", {}, subdef_]}]] :=
  (moldef[name] = ProcessSubdef /@ subdef);

In[40]:= ProcessSubdef[
  XMLElement["atom", {"type" → t_, "id" → id_, pos_}, {}]] :=
  (moldef[id] = Append[moldef[t], GetPos[pos]]);

In[41]:= ProcessSubdef[
  XMLElement["bond", {"atom1" → a1_, "atom2" → a2_}, {}]] :=
  Bond[moldef[a1][[3]], moldef[a2][[3]]];
```

In the definition of **ProcessDefinition**, **subdef** is the list of atoms and bonds. We map **ProcessSubdef** onto this list. That is, what we assign to **moldef[name]** is a list of the result of **ProcessSubdef** on each atom and bond element in the molecule. When **ProcessSubdef** is called on an atom, it extracts that atom's type from **moldef**, appends the position to that expression, and stores the results under **moldef[id]**. When **ProcessSubdef** is called on a bond, it simply returns a **Bond** expression containing the positions of the two atoms it references.

We need an auxiliary function before we define **GetRad**, **GetColor**, and **GetPos**. Since positions are written as space-separated lists of numbers in MoDL, we first write a function that turns this string into a *Mathematica* list.

```
In[42]:= MolStringListToList[molpos_String] :=
  Module[{stream = StringToStream[molpos], thelist},
    thelist = Read[stream, {Number, Number, Number}];
    Close[stream];
    thelist];
```

The functions **GetPos**, **GetRad**, and **GetColor** should take in a sequence of attributes of any length and create a list from the position attribute. In Symbolic-XML, attributes are stored as *Mathematica* rules. Both **GetPos** and **GetColor** will need to use **MolStringListToList**. **GetRad** needs only to convert a string to a number.

```
In[43]:= GetPos[___, "position" → p_, ___] := MolStringListToList[p];

In[44]:= GetRad[___, "radius" → r_, ___] := ToExpression[r];

In[45]:= GetColor[___, "color" → c_, ___] :=
  RGBColor @@ MolStringListToList[c];

In[46]:= GetColor[___] := GrayLevel[1];
```

Here is the definition of **MoDLToGraphics3D** again for reference.

```

In[47]:= MoDLToGraphics3D[modl_] := Block[{
  defs = Cases[modl, XMLElement["DEFINE", ___], Infinity],
  body = First[Cases[modl, XMLElement["body", ___], Infinity]],
  moldef, themols, theatoms},
ProcessDefinition/@defs;
themols = Flatten@Cases[body,
  XMLElement["molecule", {___, "type" → t_, ___}, _] :=> moldef[t],
  Infinity];
theatoms = Cases[body,
  XMLElement["atom", {___, "type" → t_, ___, "position" → p_, ___},
  _] :=> Append[moldef[t], MolStringListToList[p]], Infinity];
Graphics3D[MolToGraphics /@ Join[themols, theatoms], Boxed → False]
]

```

Block scopes the variables **defs**, **body**, **moldef**, **themols**, and **theatoms**. We already discussed **moldef**, and **defs** simply contains a list of the **DEFINE** elements. The function **body** just contains the **body** element of the SymbolicXML expression. That leaves **themols** and **theatoms**.

After **ProcessDefinition** is mapped to **defs**, **themols** is defined. Molecules in the body have a **type** attribute, which references the key of the molecule type defined in the head. The **Cases** statement then matches the molecules in the body and returns that molecule's type definition in **moldef**.

```

In[48]:= body = First[Cases[methane, XMLElement["body", _], Infinity]];

In[49]:= Flatten@Cases[body,
  XMLElement["molecule", {___, "type" → t_, ___}, _] :=> moldef[t],
  Infinity]

Out[49]:= {Atom[0.3, RGBColor[1, 1, 0], {0, 0, 0.4}],
  Atom[0.25, RGBColor[1, 0, 0], {0.5, 0.1, -0.4}],
  Atom[0.25, RGBColor[1, 0, 0], {-1, 0.1, 0.3}],
  Atom[0.25, RGBColor[1, 0, 0], {0.3, 0.7, 1}],
  Atom[0.25, RGBColor[1, 0, 0], {0.2, -0.9, 0.8}],
  Bond[{0, 0, 0.4}, {0.5, 0.1, -0.4}], Bond[{0, 0, 0.4}, {-1, 0.1, 0.3}],
  Bond[{0, 0, 0.4}, {0.3, 0.7, 1}], Bond[{0, 0, 0.4}, {0.2, -0.9, 0.8}]}

```

In our example, we only have one molecule in the body. If more molecules existed, the lists of **Atom** and **Bond** expressions would be merged together by **Flatten**. As we will see, the **Graphics3D** expression is simply made by drawing each **Atom** and **Bond**. Also, the body may contain other atoms as well. The definition of **theatoms** simply matches these elements, reads their type from **moldef**, and appends their positions. Thus, **theatoms** would contain a list of more atoms to be drawn.

The last line of **MoDLToGraphics3D** joins the **Atom** and **Bond** expressions in **themols** with the **Atom** expressions in **theatoms**. It then maps **MolToGraphics** onto this list. **MolToGraphics** is simply a function that returns a sphere for **Atom** expressions and a line for **Bond** expressions. Of course, we also need to define **MolToGraphics**. The definition is straightforward, provided you are familiar with **Graphics`Shapes`** and **Graphics** expressions in *Mathematica*.

```

In[50]:= MolToGraphics[Bond[pts_]] := {Thickness[0.01], Line[{pts}]};

In[51]:= MolToGraphics[Atom[r_, col_, pos_]] :=
  {SurfaceColor[col], EdgeForm[],
   TranslateShape[Sphere[r, 10, 10], pos]};

```

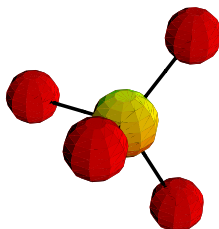
The result is a 3D graphic of methane or any other molecule you have defined in MoDL.

```

In[52]:= Show[MoDLToGraphics3D[methane]]

```

From In[52]:=



```

Out[52]= -Graphics3D-

```

□ Comparing XSLT and *Mathematica*

In many situations, there is a need to transform a document from one XML format into another. One popular technique used for this purpose is XSLT transformations. However, *Mathematica*'s pattern matching and transformation abilities allow you to do similar transformations simply by importing the original document and then manipulating the resulting SymbolicXML expression. This section gives examples of some basic XSLT transformations and explains how to do the equivalent transformations in *Mathematica*.

A Simple Template

Let us consider a very simple example. Say our XML dialect uses the **code** tag to enclose program code. Typically, this is displayed in a monospace font. If we were to convert such a document to XHTML, we would probably want to use the **pre** tag for program code. The following XSLT template would do this.

```

<xsl:template match="code">
  <pre class="code">
    <xsl:value-of select="."/>
  </pre>
</xsl:template>

```

In *Mathematica*, you can create a function to do the same.

```

In[53]:= xslt[XMLElement["pre", _, {value_String}]] :=
  XMLElement["pre", {"class" -> "code"}, {value}];

In[54]:= xslt[XMLElement["code", _, data_]] :=
  XMLElement["pre", {"class" -> "code"}, data];

```

Inserting Attribute Values

Now consider an XML application that uses the **termdef** element to indicate the definition of a new term. Again, we will convert this to XHTML. We would like to anchor the definition with a named **a** element so that we can link directly to that location in the document. Assuming we have templates to handle whatever string formatting is inside the **termdef** element, we can use the following XSLT.

```
<xsl:template match="termdef">
  <span class="termdef">
    <a name="{@id}">[Definition: ]&nbsp;&nbsp;&nbsp;</a>
    <xsl:apply-templates/>
  </span>
</xsl:template>
```

Notice that the **name** attribute in the resultant XHTML gets the value of the **id** attribute of the original **termdef** element. In *Mathematica*, you can do the following.

```
In[55]:= xslt[XMLElement["termdef", {___, "id" -> id_, ___}, contents_]] :=
  XMLElement["span", {"class" -> "termdef"},
    {XMLElement["a", {"name" -> id}, {"[Definitions:] "}],
     Sequence @@ xslt /@ contents}];
```

Using Predicates

Let us consider a more complicated example that involves XPath predicates. Assume we would like to match a **note** element, but only if it either has a **role** attribute set to **example** or if it contains an **eg** element as a child. Let us look at an XSLT template, and then explain what it does.

```
<xsl:template match="note[@role='example' or child::eg]">
  <div class="exampleOuter">
    <div class="exampleHeader">Example</div>
    <xsl:if test="*[1][self::p]">
      <div class="exampleWrapper">
        <xsl:apply-templates select="*[1]" />
      </div>
    </xsl:if>
    <div class="exampleInner">
      <xsl:apply-templates select="eg" />
    </div>
    <xsl:if test="*[position()>1 and self::p]">
      <div class="exampleWrapper">
        <xsl:apply-templates
          select="*[position>1 and self::p]" />
      </div>
    </xsl:if>
  </div>
</xsl:template>
```

The first **xsl:if** element checks to see if the first child element is a **p** element. If it is, then **xsl:apply-templates** is called on that child. This is similar to calling **Map** across the results of **Cases**. In the second **xsl:if** element, we check if there

are **p** child elements beyond the first child. If so, **xsl:apply-templates** is called on those. Here is the corresponding *Mathematica* code.

```
In[56]:= xslt[
 XMLElement["note",
    atts : {___, "role" -> "example", ___}, contents] |
  XMLElement["note", atts_, contents :
    {___, XMLElement["eg", ___], ___}] :=
XMLElement["div", {"class" -> "exampleOuter"}, {
  XMLElement["div", {"class" -> "exampleHeader"}, {"Example"}],
  If[MatchQ[contents, {XMLElement["p", ___], ___}],
    XMLElement["div",
      {"class" -> "exampleWrapper"}, xslt[contents[[1]]],
      Sequence[],
      XMLElement["div", {"class" -> "exampleInner"},
        Map[xslt, Cases[contents, XMLElement["eg", ___]]],
        Sequence @@ Map[xslt, Cases[Drop[contents, 1],
          XMLElement["p", ___]]]]];

In[57]:= {a, If[False, bob, Sequence @@ {}], b}

Out[57]= {a, b}
```

Traversing Upwards

So far, all the examples we have given in XSLT have had a very simple implementation in *Mathematica* using SymbolicXML. In each of these cases, however, we were selecting expressions that were nested inside of the given expression. What if we wanted to select an ancestor or sibling? The following examples show how this can be done in *Mathematica*.

To clarify the problem and find a solution, we have to realize that an XML document is just a stream of characters that follows a specific syntax. Tools for manipulating XML documents treat XML according to some model. In the case of XSLT (and its path-selection language, XPath), this model is that of a tree. Since *Mathematica* is a list-based language, it treats XML as nested expression lists.

While these two models are similar, they have important differences. Most notably, in nested lists you do not inherently have any concept of the containing list. Technically, any transformation that can be done with axis types, like ancestor, can be done without them. However, it is often convenient to traverse up the XML document.

Let us look at an example and then discuss how to implement the same behavior in *Mathematica*. While it will involve a slightly different technique than we have used above, it will nonetheless be rather simple. Consider the following XML document.

```

In[58]:= myxml = XML`Parser`XMLGetString["<document>
      <body>
        <p>This is a paragraph.</p>
        <p>This paragraph contains a bibliography reference:
          <bibref ref=\"mybibref\"/>
        </p>
      </body>
      <bibliography>
        <bibl id=\"mybibref\">My Bibliography Reference</bibl>
      </bibliography>
    </document>"];

```

We will assume we simply want to have a template that matches **bibref** elements and replaces them with the text inside of the corresponding **bibl** element. In XSLT, we would write the following template.

```

<xsl:template match="bibref">
  <xsl:param name="ref">
    <xsl:value-of select="@ref"/>
  </xsl:param>
  <xsl:value-of select="/bibliography/bibl[@id = $ref][1]"/>
</xsl:template>

```

The problem with using the same approach in *Mathematica* is that once we have matched a **bibref** element, we no longer have any information about the elements containing it. As a remedy, we will instead pass an expression containing the entire SymbolicXML expression. Notice that the **bibref** element in question can be obtained from

```

In[59]:= Part[myxml, 2, 3, 1, 3, 2, 3, 2]
Out[59]= XMLElement[bibref, {ref → mybibref}, {}]

```

Rather than pass the **XMLElement** expression, we can pass this expression wrapped in **Hold**. That way, we can easily obtain the **bibref** element by calling **ReleaseHold**, and we can access ancestors by dropping indices from the **Part** expression. However, we will need to write a pattern matching function so that we can match these in definitions of functions.

```

In[60]:= XMLMatchQ[expr_, patt_] :=
  And[
    MatchQ[expr, Hold[Part[____]]],
    MatchQ[ReleaseHold[expr], patt]];

In[61]:= XMLMatchQ[Hold[Part[myxml, 2, 3, 1, 3, 2, 3, 2]],
  XMLElement["bibref", _, _]]
Out[61]= True

```

The *Mathematica* transformation then becomes relatively simple.

```

In[62]:= Clear[xslt]

```

```

In[63]:= xslt[xml_?(XMLMatchQ[#, XMLElement["bibref", {___}, _]] &)] :=
Module[{ref},
  ref = "ref" /. ReleaseHold[xml][[2]];
  First[Cases[Part[xml, 1, 1][[2]],
    XMLElement["bibliography", _, {___},
      XMLElement["bibl", {"id" → ref}, {s_}, ___] → s, 2]]]

In[64]:= xslt[Hold[Part[myxml, 2, 3, 1, 3, 2, 3, 2]]]

Out[64]= My Bibliography Reference

```

□ Converting a Notebook to HTML

Suppose you need to export a notebook in a specific XML format (apart from standard formats listed under the Save As Special menu). One option would be to export to NotebookML and then use some external tool (e.g., XSLT rules) to transform to the desired form of XML. But often it is just as easy to perform the manipulation within *Mathematica*, after first converting the notebook expression into SymbolicXML. As an example, let us recreate a simplified version of the File > Save As Special > HTML menu's functionality.

You can download the sample notebook used in this example from www.mathematica-journal.com. The following commands import the notebook expression so we can manipulate it.

```

In[65]:= nb = NotebookOpen[ToFileName[Directory[], "Shakespeare.nb"]];

In[66]:= data = NotebookGet[nb]

Out[66]= Notebook[
  {Cell[CellGroupData[{Cell[Quotations, Title], Cell[CellGroupData[
    {Cell[Shakespeare, Section], Cell[TextData[{Now is the
      winter of our discontent made glorious summer by this ,
      StyleBox[Son of York, FontSlant → Italic],
      . And all the clouds that lowered on our house,
      in the Ocean's deep bosom buried.}], Text],
    Cell[TextData[{Forgive me, though bleeding piece of earth,
      that I am meek and gentle with these butchers. Thou art
      the ruineſt of the nobleſt man that ever lived
      in the tide of time. Woe to the hands ,
      StyleBox[that ſhed this coſty blood, FontWeight → Bold],
      !}], Text]], Open]], Open]],
  FrontEndVersion → 4.1 for X, ScreenRectangle →
    {{0., 1280.}, {0., 1024.}}, WindowSize → {520., 600.},
    WindowMargins → {{22., Automatic}, {Automatic, 63.}},
    CellLabelAutoDelete → True]

```

Our method will be to define a recursive function, **transform**, to process the original (notebook) expression from top to bottom, similar to the templates of XSLT. First, we establish a default definition to discard anything not explicitly matched by other patterns. (Given our “top-down” approach, perhaps this

should be the last definition, but we place it here to reduce extraneous output in the intermediate results.)

```
In[67]:= Clear[transform];
         transform[___] := Sequence[];
```

The above definition uses **Sequence[]** for the following reason: since **transform** will be applied recursively, the best “null” result is one that can be dropped in the midst of a list of arguments without disrupting the syntax.

We start with the notebook expression itself.

```
In[69]:= transform[Notebook[contents_List, ___]] := XMElement[
         "html", {}, {XMElement["body", {}, transform/@contents]}];
```

The following points are worth noting.

- The argument pattern must be robust enough to accept all variants. (Even though the notebook options are discarded in this conversion, a **BlankNullSequence** (___) is included to allow for them.)
- The only thing done with the **contents** argument is to pass it back to **transform**.
- The third argument is *always* a **List**. Forgetting this is a common pitfall.
- Notice that we have dropped the **head** section that is usually included in an HTML document.

The same general theme is followed for the remaining definitions.

Next, we discard cell-grouping information because the HTML has no use for it.

```
In[70]:= transform[Cell[CellGroupData[contents_List, ___]]] :=
         Sequence@@(transform/@contents);
```

Mathematica sectional heads are translated to their HTML counterparts.

```
In[71]:= transform[Cell[title_String, "Title"]] :=
         XMElement["h1", {}, {title}];
         transform[Cell[title_String, "Section"]] :=
         XMElement["h2", {}, {title}];
```

The **Text** cells introduce a complication: the contents of a *Mathematica Text* cell can be either a simple string or a **TextData**-wrapped list, if the text has additional information, such as font changes, specified. Thus, we need a definition for both cases.

```
In[73]:= transform[Cell[contents_String, "Text"]] :=
         XMElement["p", {}, {contents}];
         transform[Cell[TextData[contents_List], "Text"]] :=
         XMElement["p", {}, transform/@contents];
```

Simple strings should just be passed on as is. Once again, in keeping with a top-down style, this should be placed later in the sequence of definitions. But placing it earlier helps make the intermediate results more meaningful.

```
In[75]:= transform[contents_String] := contents;
```

Finally, we deal with (simple) font changes.

```
In[76]:= transform[StyleBox[contents_String, FontSlant -> "Italic"]] :=
  XMLElement["i", {}, {contents}];
transform[StyleBox[contents_String, FontWeight -> "Bold"]] :=
  XMLElement["b", {}, {contents}];
```

Here is the final product.

```
In[78]:= result = transform[data]

Out[78]= XMLElement[html, {},
  {XMLElement[body, {}, {XMLElement[h1, {}, {Quotations}],
    XMLElement[h2, {}, {Shakespeare}], XMLElement[p, {},
      {Now is the winter of our discontent made glorious summer by this ,
        XMLElement[i, {}, {Son of York}],
        . And all the clouds that lowered on our house, in
          the Ocean's deep bosum buried.}], XMLElement[p, {},
        {Forgive me, though bleeding piece of earth, that I am meek and
          gentle with these butchers. Thou art the ruineſt of the nobleſt
          man that ever lived in the tide of time. Woe to the hands ,
          XMLElement[b, {}, {that shed this costly blood}], !]]]]]
```

You can get output in a more human-readable form by using **ExportString**.

```
In[79]:= ExportString[result, "XML"]

Out[79]= <html>
  <body>
    <h1>Quotations</h1>
    <h2>Shakespeare</h2>
    <p>Now is the winter of our discontent made glorious summer
      by this <i>Son of York</i>. And all the clouds that lowered
      on our house, in the Ocean&apos;s deep bosum buried.</p>
    <p>Forgive me, though bleeding piece of earth, that I am
      meek and gentle with these butchers. Thou art the ruineſt
      of the nobleſt man that ever lived in the tide of time.
      Woe to the hands <b>that shed this costly blood</b>!</p>
  </body>
</html>
```

We can verify that this is well-formed XML.

```
In[80]:= SymbolicXMLErrors[result]

Out[80]= {}
```

And, of course, the SymbolicXML can be exported to a file, suitable for viewing with a web browser.

```
In[80]:= Export["Quotes.html", result, "XML"]

Out[80]= Quotes.html
```

An alternative to a recursive function is to apply a list of replacement rules using **ReplaceRepeated**.

```
In[81]:= nb2html = {
  Notebook[contents_List, ___] ->
    XMLElement["html", {}, {XMLElement["body", {}, contents]}],
  Cell[CellGroupData[{contents___}, ___] -> contents,
  Cell[title_, "Title"] -> XMLElement["h1", {}, {title}],
  Cell[title_, "Section"] -> XMLElement["h2", {}, {title}],
  Cell[contents_, "Text"] -> XMLElement["p", {}, {contents}],
  TextData[{contents___}] -> contents,
  StyleBox[contents_, FontSlant -> "Italic"] ->
    XMLElement["i", {}, {contents}],
  StyleBox[contents_, FontWeight -> "Bold"] ->
    XMLElement["b", {}, {contents}]
};
result2 = data //. nb2html

Out[82]= XMLElement[html, {},
  {XMLElement[body, {}, {XMLElement[h1, {}, {Quotations}],
    XMLElement[h2, {}, {Shakespeare}], XMLElement[p, {},
      {Now is the winter of our discontent made glorious summer by this ,
        XMLElement[i, {}, {Son of York}], .
And all the clouds that lowered on our house, in the Ocean'
  s deep bosom buried.}], XMLElement[p, {},
    {Forgive me, though bleeding piece of earth, that I am meek
      and gentle with these          butchers. Thou art the
        ruineſt of the nobleſt man that ever lived in the tide of time.
Woe to the hands , XMLElement[b, {}, {that ſhed this coſty blood}],
!}}]]}]}
```

The two methods produce identical results.

```
In[83]:= result===result2
```

```
Out[83]= True
```

Here is how the two methods differ.

- Since the recursion occurs implicitly *via* **ReplaceRepeated**, the latter implementation is cleaner in spots. In particular, contrast the handling of **Text** cells: the **TextData** rule can be separated from the **Cell** rule. The same could be accomplished for the recursive function, but at the cost of additional patterns for the various forms that **contents** might take (e.g., **List** *versus* **String** and so on). **ReplaceRepeated**, by acting on all subexpressions, obviates this need.
- There is no default rule for the second method. Any unhandled parts of the original *Mathematica* expression will pass through unchanged, probably rendering invalid XML.

Finally, we use **Clear** to remove the definitions of all the symbols.

```
In[84]:= Clear[nb, data, transform, result, nb2html, result2];
```

□ Verifying SymbolicXML Syntax

You can use the function **XML`SymbolicXMLErrors** to find errors with a SymbolicXML expression. This function returns a part specification that you can use with functions like **Part** or **Extract** to access the problematic part of your SymbolicXML expression. Let us return to the American League hitters from our earlier example.

```
In[85]:= AL = Import["ALHitters.xml", "SymbolicXML"];
```

We saw earlier that there were **PlayerRecord** nodes for which the **Salary** node contained the string "#N/A". Suppose we decide that any player whose salary is not available to us, must be making \$1,000,000. However, when we do our transformation, we make an easily overlooked mistake.

```
In[86]:= AL = AL /.
  XMLElement["Salary", {}, {"#N/A"}] →
  XMLElement["Salary", {}, "1000000"];
```

We have now created incorrect SymbolicXML. We have put the string "1000000" as the third element of the **XMLElement** expression, rather than using a list containing that string. Suppose we did not know this, though, and that later on we find that **Export** produces errors when we try to write our modified XML to a file. We can use **SymbolicXMLErrors** to find the problematic expressions.

```
In[87]:= AErrors = XML`SymbolicXMLErrors[AL] // Short
```

```
XMLElement::contentsList :
1000000 in XMLElement[Salary, {}, 1000000] is not a
List of contents. The third item in an XMLElement
must be a List of contents, even if it is an empty List.
```

```
XMLElement::contentsList :
1000000 in XMLElement[Salary, {}, 1000000] is not a
List of contents. The third item in an XMLElement
must be a List of contents, even if it is an empty List.
```

```
XMLElement::contentsList :
1000000 in XMLElement[Salary, {}, 1000000] is not a
List of contents. The third item in an XMLElement
must be a List of contents, even if it is an empty List.
```

```
General::stop : Further output of
XMLElement::contentsList will be suppressed during this calculation.
```

```
Out[87]//Short=
{{2, 3, 39, 3, 35}, {2, 3, 69, 3, 35},
 <<84>>, {2, 3, 293, 3, 35}, {2, 3, 294, 3, 35}}
```

Notice that the output is preceded by several **XMLElement::name** error messages. This indicates that something is wrong with the input. The output of **SymbolicXMLErrors**, however, tells us exactly where something went wrong.

ALerrors now contains a list of part specifications where the errors occurred. Here we examine the first error.

```
In[88]:= First[ALerrors]
Out[88]= {2, 3, 39, 3, 35}

In[89]:= Part[AL, Sequence@@First[ALerrors]]
Out[89]= XMLElement[Salary, {}, 1000000]

In[90]:= Cases[FoldList[Part, AL, First[ALerrors]], XMLElement[s_, __] -> s]
Out[90]= {AmericanLeagueIndividualHitters, PlayerRecord, Salary}
```

This problem is easy enough to fix.

```
In[91]:= Part[AL, Sequence@@First[ALerrors]] =
  Part[AL, Sequence@@First[ALerrors]] /.
    XMLElement["Salary", p_, s_String] -> XMLElement["Salary", p, {s}]
Out[91]= XMLElement[Salary, {}, {1000000}]
```

We can see that the rest of the errors are of the same nature.

```
In[92]:= Extract[AL, Rest[ALerrors]] // Short
Out[92]//Short=
  {XMLElement[Salary, {}, {1000000}],
    <<85>>, XMLElement[Salary, {}, {1000000}]}
```

By using **Map**, we can fix the rest of the errors in the same way.

```
In[93]:= Map[(Part[AL, Sequence@@#] = Part[AL, Sequence@@#] /.
  XMLElement["Salary", p_, s_String] -> XMLElement["Salary", p, {s}]) &,
  ALerrors] // Short
Out[93]//Short=
  {XMLElement[Salary, {}, {1000000}],
    <<86>>, XMLElement[Salary, {}, {1000000}]}
```

We can then verify that we have fixed the error using **SymbolicXMLerrors** again.

```
In[94]:= XML`SymbolicXMLerrors[AL]
Out[94]= {}
```

■ Exporting XML

□ Functions for Exporting XML

Export

You can export XML data from *Mathematica* using the standard **Export** function.

```
Export[file, expr]
Export[file, expr, format]
```

The first argument of the function specifies the file to which the data should be exported. The second argument specifies the data to be exported. For exporting

XML data, this can be a SymbolicXML expression or any other *Mathematica* expression. You can also specify an optional third argument to control the form of the output. For exporting XML data, the relevant file formats are "**XML**", "**NotebookXML**", "**ExpressionML**", "**MathML**", and "**SVG**".

With "**XML**" as the export format, all expressions are imported as NotebookXML or ExpressionML.

```
In[1]:= Export["test.xml", x^2, "XML"]
```

```
Out[1]= test.xml
```

```
In[2]:= !! test.xml
```

```
<?xml version='1.0'?>
<!DOCTYPE Expression SYSTEM
'http://www.wolfram.com/XML/notebookxml1.dtd'>
<Expression xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
  <Function>
    <Symbol>Power</Symbol>
    <Symbol>x</Symbol>
    <Number>2</Number>
  </Function>
</Expression>
```

With "**MathML**" specified as the export format, the same expression is written out as MathML.

```
In[3]:= Export["test.mml", x^2, "MathML"]
```

```
Out[3]= test.mml
```

```
In[4]:= !! test.mml
```

```
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <semantics>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <annotation-xml encoding='MathML-Content'>
      <apply>
        <power/>
        <ci>x</ci>
        <cn type='integer'>2</cn>
      </apply>
    </annotation-xml>
  </semantics>
</math>
```

If **Export** is used with only two arguments, *Mathematica* determines the export format based on the filename extension. The ".**xml**" extension is associated with

XML. Hence, **Export**[*filename.xml*, *expr*] is equivalent to **Export**[*filename.xml*, *expr*, "XML"], as seen in the example below.

```
In[5]:= Export["test.xml", x^2]
```

```
Out[5]= test.xml
```

```
In[6]:= !! test.xml
```

```
<?xml version='1.0'?>
<!DOCTYPE Expression SYSTEM
'http://www.wolfram.com/XML/notebookxml1.dtd'>
<Expression xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
  <Function>
    <Symbol>Power</Symbol>
    <Symbol>x</Symbol>
    <Number>2</Number>
  </Function>
</Expression>
```

The .mml extension is associated with MathML. Hence, **Export**[*filename.mml*, *expr*] is equivalent to **Export**[*filename.mml*, *expr*, "MathML"], as seen in the example below.

```
In[7]:= Export["test.mml", x^2]
```

```
Out[7]= test.mml
```

```
In[8]:= !! test.mml
```

```
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <semantics>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <annotation-xml encoding='MathML-Content'>
      <apply>
        <power/>
        <ci>x</ci>
        <cn type='integer'>2</cn>
      </apply>
    </annotation-xml>
  </semantics>
</math>
```

You can control the various details of the export process using the conversion options feature of the **Export** function.

The following commands delete the test files created by evaluating the commands in this section.

```
In[9]:= DeleteFile[{ToFileName[{Directory[]}, "test.xml"]}] ;
DeleteFile[{ToFileName[{Directory[]}, "test.mml"]}]
```

ExportString

You can convert *Mathematica* expressions into XML strings using the **ExportString** function. This function has the following syntax.

ExportString[*expr*, *format*]

For exporting as XML, the relevant formats are "XML", "NotebookML", "ExpressionML", "MathML", and "SVG".

This command produces a SymbolicXML expression.

```
In[11]:= ImportString["<root><child/></root>", "SymbolicXML"]
Out[11]= XMLObject[Document][{ },
      XMLElement[root, { }, {XMLElement[child, { }, { }]}], { }
```

If the SymbolicXML expression is supplied as the first argument of **ExportString**, the resulting output is ordinary XML.

```
In[12]:= ExportString[%, "XML"]
Out[12]= <root>
      <child/>
      </root>
```

If the first argument is some other type of expression, the output is in the form of ExpressionML.

```
In[13]:= ExportString["some text", "XML"]
Out[13]= <?xml version='1.0' ?>
      <!DOCTYPE Expression SYSTEM '
      http://www.wolfram.com/XML/notebookxml1.dtd'>
      <Expression xmlns:mathematica='http://www.wolfram.com/XML/'
      xmlns='http://www.wolfram.com/XML/'>
      <String>some text</String>
      </Expression>
```

You can control various details of the export process using the conversion options feature of the **ExportString** function.

□ Conversion Options

Introduction

The standard **ConversionOptions** feature of **Export** can be used for controlling the export process. The syntax for specifying a conversion option is as follows.

Export[*file*, *expr*, *format*, **ConversionOptions** →
 {*option1* → *value1*, *option2* → *value2*, ...}]

Multiple conversion options can be specified by making the right-hand side of **ConversionOptions** a list of lists. For exporting XML data, the following conversion options are available.

- "Annotations"
- "AttributeQuoting"
- "CheckXML"
- "ElementFormatting"
- "Entities"
- "NamespacePrefixes"

Annotations

This conversion option controls which “annotations” are added to the output MathML. The value of this option is a list whose elements can be any combination of the following: **"DocumentHeader"**, **"XMLDeclaration"**, or **"DOCTYPEDeclaration"**. The order of the elements in the list is irrelevant.

XMLDeclaration

When **"XMLDeclaration"** is one of the annotations, then an XML declaration is included in the header. That is, the statement `<?xml version="1.0"?>` appears in the header.

```
In[14]:= ExportString[x, "XML",
      ConversionOptions →
        {"Annotations" → {"DocumentHeader", "XMLDeclaration"}}]

Out[14]= <?xml version='1.0'?>
<Expression xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
  <Symbol>x</Symbol>
</Expression>
```

DOCTYPEDeclaration

When **"DOCTYPEDeclaration"** is one of the annotations, then an XML document type declaration of the form `<!DOCTYPE ... >` appears in the header. This is a statement that specifies the DTD for the XML application in which the output is written.

```
In[15]:= ExportString[x, "XML",
      ConversionOptions →
        {"Annotations" → {"DocumentHeader", "DOCTYPEDeclaration"}}]

Out[15]= <!DOCTYPE Expression SYSTEM '
  http://www.wolfram.com/XML/notebookml1.dtd'>
<Expression xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
  <Symbol>x</Symbol>
</Expression>
```

DocumentHeader

With the setting, **"Annotations" → {"DocumentHeader", "XMLDeclaration", "DOCTYPEDeclaration"}**, a header containing an XML declaration and a document type declaration for the MathML DTD are automatically added to the output.

```

In[16]:= ExportString[x, "XML",
  ConversionOptions → {"Annotations" →
    {"DocumentHeader", "XMLDeclaration", "DOCTYPEDeclaration"}}]

Out[16]= <?xml version='1.0'?>
  <!DOCTYPE Expression SYSTEM '
    http://www.wolfram.com/XML/notebookxml1.dtd'>
  <Expression xmlns:mathematica='http://www.wolfram.com/XML/'
    xmlns='http://www.wolfram.com/XML/'>
    <Symbol>x</Symbol>
  </Expression>

```

When **"Annotations"** does not contain **"DocumentHeader"**, then the output has no header. This is true even if the **"Annotations"** contains other elements such as **"XMLDeclaration"** or **"DOCTYPEDeclaration"**. Thus **"DocumentHeader"** is an overall switch that controls whether the structure has a header or not.

```

In[17]:= ExportString[x, "XML",
  ConversionOptions →
    {"Annotations" → {"XMLDeclaration", "DOCTYPEDeclaration"}}]

Out[17]= <Expression xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
  <Symbol>x</Symbol>
</Expression>

```

The **"Annotations" → "DocumentHeader"** is useful for controlling the form of SymbolicXML generated. For instance, you can explicitly add an **XMLElement[Document]** to the SymbolicXML output, as shown below.

```

In[18]:= XML`MathML`ExpressionToSymbolicMathML[
  x + 2, "Annotations" → {"DocumentHeader"},
  "Formats" → "PresentationMathML"]

Out[18]= XMLObject[Document][{ },
  XMLElement[math, {xmlns → http://www.w3.org/1998/Math/MathML},
    {XMLElement[mrow, { }, {XMLElement[mi, { }, {x}],
      XMLElement[mo, { }, {+}], XMLElement[mn, { }, {2}]}]}], { }

```

Here, **"Annotations" → {"DocumentHeader"}** is not specified so the **XMLElement[Document]** is omitted from the output.

```

In[19]:= XML`MathML`ExpressionToSymbolicMathML[
  x + 2, "Formats" → "PresentationMathML"]

Out[19]= XMLElement[math, {xmlns → http://www.w3.org/1998/Math/MathML},
  {XMLElement[mrow, { }, {XMLElement[mi, { }, {x}],
    XMLElement[mo, { }, {+}], XMLElement[mn, { }, {2}]}]}]

```

AttributeQuoting

This conversion option determines whether attribute values are enclosed by single quotes or double quotes.

<i>option</i>	<i>value</i>	<i>effect</i>
"AttributeQuoting"	'	attribute values are enclosed by single quotes
	\"	attribute values are enclosed by double quotes

The option **"AttributeQuoting"** and its possible values.

With the default setting, **"AttributeQuoting"** $\rightarrow$ **"' "**, attribute values are enclosed in single quotes. This ensures that there is no conflict with *Mathematica* strings, which are typically enclosed in double quotes.

```
In[20]:= ExportString[XMLElement["foo", {"color"  $\rightarrow$  "blue"}, {"some data"}],
           "XML", ConversionOptions  $\rightarrow$  {"AttributeQuoting"  $\rightarrow$  "' "}]
```

```
Out[20]:= <foo color='blue'>some data</foo>
```

For certain applications, you might prefer attribute values to be enclosed in double quotes. You can achieve this by setting **"AttributeQuoting"** $\rightarrow$ **"\ "**. Note that the double quote character must be preceded by a forward slash to escape it.

```
In[21]:= ExportString[XMLElement["foo", {"color"  $\rightarrow$  "blue"}, {"some data"}],
           "XML", ConversionOptions  $\rightarrow$  {"AttributeQuoting"  $\rightarrow$  "\" "}]
```

```
Out[21]:= <foo color="blue">some data</foo>
```

CheckXML

This conversion option determines whether the SymbolicXML expression being exported is first checked for errors. By default this option is set to **True**.

<i>option</i>	<i>value</i>	<i>effect</i>
"CheckXML"	True	the file being exported is checked for errors
	False	the file being exported is not checked for errors

The option **"CheckXML"** and its possible values.

You can set this option to **False** if you are confident the SymbolicXML is correct, because checking the XML for errors can cause processing delays. The following example shows the delay produced by checking a small file for errors.

```
In[22]:= largeSymbolicXML = Import["test.xml", "SymbolicXML"];
```

```
In[23]:= Timing[Export["myFile", largeSymbolicXML, "XML"];]
```

```
Out[23]:= {16.0833 Second, Null}
```

```
In[24]:= Timing[Export["myFile", largeSymbolicXML,
  "XML", ConversionOptions -> {"CheckXML" -> False}];]
```

```
Out[24]:= {13.7667 Second, Null}
```

The error checking provided by this option can be quite useful because a small error may completely ruin the exported form of a large SymbolicXML expression. With the option on, small errors can often be fixed. Here the SymbolicXML has an error, but it is fixed to give a reasonable result.

```
In[25]:= Export["myFile",XMLElement["foo"], "XML"]
```

```
XMLElement::argrx :
XMLElement called with 1 arguments; 3 arguments are expected.
```

```
Export::autoFix :
There is an error in the SymbolicXML expression begin exported.
An automated attempt has been made to fix the error, however
the fix may yield XML different to that desired. The function
XML'SymbolicXMLErrors will detail the errors in the SymbolicXML expression.
```

```
Out[25]:= myFile
```

```
In[26]:= !! myFile
```

```
<foo/>
```

On the other hand, with **"CheckXML"→False**, nothing is output to the file.

```
In[27]:= Export["myFile",XMLElement["foo"],
  "XML", ConversionOptions -> {"CheckXML" -> False}]
```

```
Export::unrecov :
An unrecoverable error occurred while exporting the SymbolicXML.
```

```
Out[27]:= $Failed
```

ElementFormatting

This conversion option controls how elements are indented in the XML file. Possible values of the option are **All**, **None**, **Automatic**, or a user-defined function. The default value is **Automatic**.

<i>option</i>	<i>value</i>	<i>effect</i>
"ElementFormatting"	All	all child nodes of an element are indented and long lines are wrapped
	None	no extra indentation is added
	Automatic	elements with mixed content are not indented, while elements with element-only content are indented
	<i>fn</i>	a user-defined function

The option **"ElementFormatting"** and its possible values.

The following example shows the result of using **"ElementFormatting"**→**All**.

```
In[28]:= ExportString[
  XMLElement["root", {}, {"a", XMLElement["child", {}, {}], "b"}],
  "XML", ConversionOptions → {"ElementFormatting" → All}]

Out[28]= <root>
         a
        <child/>
         b
       </root>

In[29]:= ExportString[
  XMLElement["p", {}, {"This is a long paragraph. It is made of several
    sentences. It keeps going and going. It doesn't look very nice on
    one long line, but if it is indented, it is much easier to read."}],
  "XML", ConversionOptions → {"ElementFormatting" → All}]

Out[29]= <p>
This is a long paragraph. It is made of several
sentences. It keeps going and going. It doesn't
look very nice on one long line, but if it is
indented, it is much easier to read.
</p>
```

With **"ElementFormatting"**→**None**, no extra indentation is added.

```
In[30]:= ExportString[
  XMLElement["root", {}, {"a", XMLElement["child", {}, {}], "b"}],
  "XML", ConversionOptions → {"ElementFormatting" → None}]

Out[30]= <root>a<child/>b</root>
```

With **"ElementFormatting"**→**Automatic**, elements with mixed content (strings as content) are not indented, while elements with element-only content are indented.

```

In[31]:= ExportString[
 XMLElement["element-only", {}, {XMLElement["empty", {}, {}],
    XMLElement["string-only", {}, {"a string"}], XMLElement["mixed",
      {}, {"a", XMLElement["empty", {}, {}], "b"}], XMLElement[
        "element-only", {}, {XMLElement["empty", {}, {}]}]], "XML"]

Out[31]= <element-only>
  <empty/>
  <string-only>a string</string-only>
  <mixed>a<empty/>b</mixed>
  <element-only>
    <empty/>
  </element-only>
</element-only>

```

We saw that with **"ElementFormatting"→All**, long strings are line wrapped. This can be used to produce output similar to HTML. On the other hand, **ElementFormatting→Automatic** produces one long line of text.

```

In[32]:= ExportString[XMLElement["p", {},
  {"This is a long paragraph. It is made of several sentences. It keeps
    going and going. It doesn't look very nice on one long line,
    but if it is indented, it is much easier to read."}], "XML"]

Out[32]= <p>This is a long paragraph. It is made of several sentences. It
  keeps going and going. It doesn't look very nice on one
  long line, but if it is indented, it is much easier to read.</p>

```

Advanced users can also specify a function to determine the formatting. The function is passed a two-element list, *{namespace, localName}*. The function should return **True** when indenting is wanted, **False** when no indenting is wanted, and **Automatic** for cases where the element-only content should be indented and mixed content should not be indented.

```

In[33]:= myIndentFunction[{"", "indent"}] := True;
myIndentFunction[{"", "noIndent"}] := False;
myIndentFunction[{"", "autoIndent"}] := Automatic;

In[36]:= ExportString[XMLElement["indent", {},
  {XMLElement["noIndent", {}, {XMLElement["indent", {}, {}]}],
    XMLElement["autoIndent", {},
      {XMLElement["autoIndent", {}, {"foo"}]}]], "XML",
  ConversionOptions → {"ElementFormatting" → myIndentFunction}]

Out[36]= <indent>
  <noIndent><indent/></noIndent>
  <autoIndent>
  <autoIndent>foo</autoIndent>
</autoIndent>
</indent>

```

Entities

When exporting XML documents, it is sometimes desirable to represent special characters using named character entities. The **"Entities"** conversion option supports output of these named character entities.

<i>option</i>	<i>value</i>	<i>effect</i>
"Entities"	"HTML"	resolves entity names defined in HTML
	"MathML"	resolves entity names defined in MathML
	<i>list</i>	a list of character replacement rules

The option **"Entities"** and its possible values.

You can also specify a list of entities as the value of this option as a list. For example, if you want to export both HTML and MathML entities, you could use this setting.

```
"Entities"→{"HTML", "MathML"}
```

If neither the **"HTML"** nor **"MathML"** settings are used, all characters are still output correctly in XML. However, they may be numeric entities or encoded in UTF-8.

Here we use the **"HTML"** setting to turn an α in the input into the named character entity **α**.

```
In[37]:= xmlString = ExportString[XMLElement["foo", {}, {" $\alpha$ "}],  
    "XML", ConversionOptions → {"Entities" → "HTML"}]
```

```
Out[37]:= <foo>&alpha;</foo>
```

You can also enter your own list of character replacement rules to be used. If you do this, then you are also responsible for including some basic escaping required by XML. For example:

```
"Entities"→{  
    "&"→"&amp;",  
    "<"→"&lt;",  
    ">"→"&gt;",  
    "\""→"&quot;",  
    "'"→"&apos;",  
    "h"→"&lowerCaseH;"  
}
```

If you specify any value for **Entities**, it is your responsibility to ensure that appropriate entity declarations are present. For example, by using the **"HTML"** setting, you can easily generate XML with HTML entities. In this example, the

Icelandic character “thorn” is exported as the corresponding character entity reference.

```
In[38]:= xmlString = ExportString[XMLElement["foo", {}, {"p"}],
      "XML", ConversionOptions -> {"Entities" -> "HTML"}]
```

```
Out[38]= <foo>&thorn;</foo>
```

Here, the **thorn** entity is not declared, and so the character is exported as a numeric character reference.

```
In[39]:= ExportString[XMLElement["foo", {}, {"p"}], "XML"]
```

```
Out[39]= <foo>&#254;</foo>
```

NamespacePrefixes

This option lets you generate XML markup with a specific namespace declaration and namespace prefixes. The option is specified in the form

"NamespacePrefixes" -> {url -> prefix}

where *url* and *prefix* are strings specifying the URL of the namespace and the namespace prefix. In the following example, the **"NamespacePrefix"** option is used to generate presentation markup with each MathML element having a namespace prefix **"mml"** associated with the MathML namespace.

```
In[40]:= Export["test.mml", x^2, "MathML", ConversionOptions ->
      {"Formats" -> {"PresentationMathML"}, "NamespacePrefixes" ->
      {"http://www.w3.org/1998/Math/MathML" -> "m"}}]
```

```
Out[40]= test.mml
```

```
In[41]:= !! test.mml
```

```
<m:math xmlns:m='http://www.w3.org/1998/Math/MathML'
  xmlns='http://www.w3.org/1998/Math/MathML'>
  <m:msup>
    <m:mi>x</m:mi>
    <m:mn>2</m:mn>
  </m:msup>
</m:math>
```

■ Working with NotebookML

□ Introduction

NotebookML is an XML format for describing *Mathematica* notebooks. It involves a mapping of a notebook expression to a similar XML tree structure. The names of elements and attributes in NotebookML are chosen to match the names of the corresponding parts of the original notebook expression. Here is an example of a simple notebook expression.

```
Notebook[{Cell["Here is some text!","Text"]}]
```

Here is the same expression in NotebookML.

```
<Notebook>
<List>
<Cell>
  <String>This is a cell.</String>
  <Style>
    <String>Section</String>
  </Style>
</Cell>
</List>
</Notebook>
```

Note the direct correspondence between the parts of the notebook expression and their XML counterparts. The conversion is done on the **FullForm** of the notebook expression. For example, while a list can be denoted by `{}` in a notebook, the underlying representation of the list is still `List[...]`. Hence, in NotebookML, a list would be represented by a `List` element of the form `<List>...</List>`.

NotebookML is useful for exporting complete notebooks in XML format. However, you can also export individual cells, mathematical formulas, or other types of content in a notebook as XML, using what is called ExpressionML. This is a subset of NotebookML that enables you to save arbitrary *Mathematica* expressions in XML format. Here is the **FullForm** for a mathematical formula.

```
In[1]:= Sin[x + 1] // FullForm
Out[1]//FullForm=
Sin[Plus[1, x]]
```

Here is the ExpressionML representation of the same formula. You can generate the NotebookML or ExpressionML for any type of notebook expression, using **Export** or **ExportString**.

```
<Expression>
<Function>
  <Symbol>Sin</Symbol>
</Function>
  <Symbol>Plus</Symbol>
    <Number>1</Number>
    <Symbol>x</Symbol>
  </Function>
</Function>
</Expression>
```

NotebookML and ExpressionML are both 100% well-formed, standards-compliant XML. Hence, they make it easy to integrate *Mathematica* notebooks or parts of notebooks into any XML framework or workflow. The namespace for NotebookML and ExpressionML is specified by the URL www.wolfram.com/XML. Also, the DTD for NotebookML and ExpressionML is located at the URL www.wolfram.com/XML/DTD/notebookml1.dtd.

□ Syntax of NotebookML

Strings, Numbers, and Symbols

The **String**, **Number**, and **Symbol** elements are the only NotebookML elements that can directly contain character data. All other NotebookML elements are either empty elements or can only contain other elements. For example, here is a simple cell in **Text** style.

```
Cell["This is a text cell", "Text"]
```

Here is the corresponding NotebookML.

```
In[2]:= ExportString[Cell["This is a text cell", "Text"], "XML"]

Out[2]= <?xml version='1.0'?>
<!DOCTYPE Cell SYSTEM 'http://www.wolfram.com/XML/notebookml1.dtd'>
<Cell class='Text'
  CreatedBy='Mathematica 4.2'
  xmlns:xhtml='http://www.w3.org/1999/xhtml'
  xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
  <String>This is a text cell</String>
  <Style>
    <String>Text</String>
  </Style>
</Cell>
```

Note that each string in the cell expression is represented by a **String** element in NotebookML. One benefit of using a **String** element to describe string data is that it provides a clearer indication to XML-processing applications that the whitespace inside the string data is significant.

In addition to strings, the contents of a cell can contain numbers or symbols. These structures are represented in NotebookML using the elements **Number** and **Symbol**. For example, the *Mathematica* expression

```
CellFrame[{1,0},{0,1}]
```

has the following NotebookML representation.

```
<CellFrame>
  <List>
    <Number>1</Number>
    <Number>0</Number>
  </List>
  <List>
    <Number>0</Number>
    <Number>1</Number>
  </List>
</CellFrame>
```

Symbol elements lack any special structure. As an example, consider the following *Mathematica* code fragment.

```
Sin[x]
```

The corresponding NotebookML is shown below. The **Function** element is used to enclose the names of all built-in functions.

```
<Function>
  <Symbol>Sin</Symbol>
  <Symbol>x</Symbol>
</Function>
```

In notebooks, two-dimensional structures such as fractions, subscripts, superscripts, and so on, are represented by box expressions. Here is the expression for a cell containing both text and a box expression.

```
Cell[TextData[{"Here is some math: ",
  Cell[BoxData[SuperscriptBox["x", "2"]]]], "Text"]
```

Here is the NotebookML representation of the above cell. Note the use of the **TextData**, **BoxData**, and **SuperscriptBox** elements in NotebookML to represent the corresponding parts of the cell expression.

```
In[3]:= ExportString[Cell[TextData[{"Here is some math: ",
  Cell[BoxData[SuperscriptBox["x", "2"]]]], "Text"], "XML"]

Out[3]= <?xml version='1.0'?>
  <!DOCTYPE Cell SYSTEM 'http://www.wolfram.com/XML/notebookml1.dtd'>
  <Cell class='Text'
    CreatedBy='Mathematica 4.2'
    xmlns:xhtml='http://www.w3.org/1999/xhtml'
    xmlns:mathematica='http://www.wolfram.com/XML/'
    xmlns='http://www.wolfram.com/XML/'>
    <TextData>
      <List>
        <String>Here is some math: </String>
        <Cell>
          <BoxData>
            <List>
              <SuperscriptBox>
                <String>x</String>
                <String>2</String>
              </SuperscriptBox>
            </List>
          </BoxData>
        </Cell>
      </List>
    </TextData>
    <Style>
      <String>Text</String>
    </Style>
  </Cell>
```

Options

In *Mathematica*, various properties of notebooks, cells, and the contents of a cell are described by options. A notebook expression with options specified has the following underlying structure.

Notebook[{*cell1*, *cell2*, *cell3*, ...}, *options*]

Each option has the general form *name* $\rightarrow$ *value*, where the left-hand side specifies the option name and the right-hand side specifies the value of that option. In NotebookML, an option is represented using the form shown below.

```
<Option>
  <Symbol>name</Symbol>
  NotebookML representation of value
</Option>
```

For example, this notebook expression will include a ruler in the toolbar at the top of the displayed window.

Notebook[{Cell["Some text", "Text"]}, WindowToolbars $\rightarrow$ {"RulerBar"}]

Here is the corresponding NotebookML.

```
In[4]:= ExportString[Notebook[{Cell["Some text", "Text"]},
  WindowToolbars  $\rightarrow$  {"RulerBar"}], "XML"]
```

```
Out[4]:= <?xml version='1.0'?>
  <!DOCTYPE Notebook SYSTEM '
    http://www.wolfram.com/XML/notebookml1.dtd'>
  <Notebook CreatedBy='Mathematica 4.2'
    xmlns:xhtml='http://www.w3.org/1999/xhtml'
    xmlns:mathematica='http://www.wolfram.com/XML/'
    xmlns='http://www.wolfram.com/XML/'>
    <List>
      <Cell class='Text'>
        <String>Some text</String>
        <Style>
          <String>Text</String>
        </Style>
      </Cell>
    </List>
    <Options>
      <Option>
        <Symbol>WindowToolbars</Symbol>
        <Function>
          <Symbol>List</Symbol>
          <String>RulerBar</String>
        </Function>
      </Option>
    </Options>
  </Notebook>
```

Multiple options are grouped together in an **Options** element. Although this is not necessary, it is useful (for ease of programming and efficiency reasons) in the context of many XML applications like XSLT. For example, the cell expression:

```
Cell["Some text",FontSize→8,FontWeight→"Bold"]
```

has the following representation in NotebookML.

```
In[5]:= ExportString[
  Cell["Some text", FontSize → 8, FontWeight → "Bold"], "XML"]

Out[5]= <?xml version='1.0'?>
<!DOCTYPE Cell SYSTEM 'http://www.wolfram.com/XML/notebookxml1.dtd'>
<Cell CreatedBy='Mathematica 4.2'
  xmlns:xhtml='http://www.w3.org/1999/xhtml'
  xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
  <String>Some text</String>
  <Options>
    <Option>
      <Symbol>FontSize</Symbol>
      <Number>8</Number>
    </Option>
    <Option>
      <Symbol>FontWeight</Symbol>
      <String>Bold</String>
    </Option>
  </Options>
</Cell>
```

□ Importing NotebookML

NotebookML files can be imported through the kernel using **Import**. The first argument of **Import** is the filename of the file you are importing. If the file carries the **.xml** or **.nbml** extension and is in NotebookML format, then **Import** will automatically convert the file into a notebook expression. Here is an example.

```
In[6]:= Import["test.xml"]

Notebook[Cell[Hello. This is a test notebook., Text],
  FrontEndVersion→4.2 for Microsoft Windows,
  ScreenRectangle→{{0., 1024.}, {0., 695.}},
  WindowSize→{496., 599.},
  WindowMargins→{{18., Automatic}, {Automatic, 23.}}]
```

You can also specify the import format explicitly, by using **"SymbolicXML"** or **"NotebookML"** as the second argument to **Import**. This is useful when the file you are importing does not have a **.xml** or **.nbml** extension to indicate the nature of its contents.

The import process is much faster, especially for large files, if you specify **"NotebookML"** as the import format. However, you should only use this form if you are sure that the file is a NotebookML document.

```
Import["test.xml", "NotebookML"];
```

You can also import the file as SymbolicXML using the following command.

```
Import["test.xml", "SymbolicXML"];
```

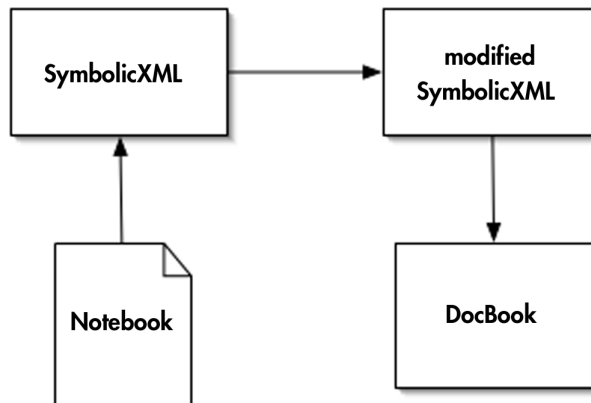
□ Processing NotebookML

Converting between NotebookML and SymbolicXML

Suppose you have written a technical paper in *Mathematica* and now want to submit the paper to a journal that only accepts submissions in a specific XML format, say DocBook. You, therefore, need to transform your notebook into DocBook format. One way to do this is to use XSLT to transform the NotebookML output from *Mathematica* into DocBook.



However, you can easily perform the same transformation completely within *Mathematica* by using SymbolicXML as an intermediary. *Mathematica*'s pattern matching and symbolic processing can make the task quite easy. A schematic of the process might look like the following.



Here is a simple example of the type of manipulations you can perform. This command defines a rule for replacing tab characters ('\t') with four nonbreaking spaces. This is something one might want to do before presenting work on the web.

```
In[7]:= fixCharRule = {XMLElement["String", a_, {s_String}] →  
      XMLElement["String", a, {StringReplace[s, "\t" → "   "]}];
```

This applies the rule to a notebook called **nb** and exports the result as an XML file.

```

In[8]:= nb = Notebooks[[2]]

NotebookObject[<<test.nb>>]

In[9]:= Export["changedchars.xml", ToSymbolicXML[nb] /. fixCharRule]

changedchars.xml

```

Preprocess

Preprocessing allows any function to be applied to the notebook before the XML conversion functions are applied to it. For example, one might want to ensure that all closed cell groups are open before proceeding with the conversion. This can be done with the following command.

```

In[10]:= openCellGroups[eachCell_] := eachCell //.
        {CellGroupData[cells_, Closed] → CellGroupData[cells, Open]};

```

One could then preprocess the notebook by using **openCellGroups** in the following way.

```

In[11]:= Export["test2.xml", openCellGroups[NotebookGet[nb]], "XML"]

Out[11]= test2.xml

```

Postprocess

Postprocessing provides a handle on the SymbolicXML object, representing the NotebookXML before it is exported. This gives you total freedom to manipulate the SymbolicXML in any fashion you choose. However, it is your responsibility to output legitimate SymbolicXML. *Mathematica* will try to automatically correct minor errors but if there is a more serious error, the export process will fail.

Here, we use a postprocessing rule to replace characters.

```

In[12]:= changeChars[XMLObject["Document"][pre_, root_, postopts___]] :=
        XMLObject["Document"][pre, changeChars[root], postopts];

In[13]:= changeChars[XMLElement[name_, atts_, contents_]] :=
        XMLElement[name, atts, Flatten[changeChars /@ contents]];

In[14]:= changeChars[s_String] := Characters[s] /. charRules

In[15]:= charRules = Dispatch[{"α" → "Mathematica is cool"}];

```

You could then postprocess the symbolic NotebookXML by using **changeChars** in the following way.

```

In[16]:= Export["test3.xml",
        changeChars[XML`ToSymbolicXML[NotebookGet[nb]]], "XML"]

Out[16]= test3.xml

```

Of course, you can use both pre- and postprocessing together.

```

In[17]:= Export["test4.xml", changeChars[
        XML`ToSymbolicXML[openCellGroups[NotebookGet[nb]]]], "XML"]

Out[17]= test4.xml

```

□ Exporting NotebookML

Introduction

You can export notebooks as NotebookML using the **Export** function. Just as when exporting a notebook as HTML, when you export a notebook as NotebookML, graphics and boxes are automatically saved as GIF images.

The first argument for **Export** specifies the file to export to, and the second argument specifies the data to be exported. If you specify the file extension of the exported document as **".xml"**, then **Export** will automatically generate NotebookML. For example, the following will export the current notebook as a file named **"anothertest.xml"** in NotebookML.

```
In[18]:= Export["anothertest.xml", InputNotebook[]]
```

```
Out[18]:= anothertest.xml
```

If you export a **Cell** with the file extension **.xml**, **Export** will automatically generate ExpressionML. The following example will export the cell as ExpressionML.

```
In[19]:= Export["celltest.xml", Cell["Some text", "Text"]]
```

```
Out[19]:= celltest.xml
```

If the filename for the exported data does not carry the **.xml** extension, then you must specify **"XML"** or **"NotebookML"** as the second argument for **Export**. The data for export is then specified as the third argument. Here is an example.

```
In[20]:= Export["moretests", InputNotebook[], "XML"]
```

```
Out[20]:= moretests
```

You can also export data using **ExportString**. This will return NotebookML for a notebook expression or ExpressionML for a cell expression without assigning the output a particular filename. The first argument for **ExportString** is the data to be exported, and the second argument is the desired export format. In this example, the notebook expression is exported as NotebookML.

```

In[21]:= ExportString[Notebook[{Cell["Some text", "Text"]}], "NotebookML"]

Out[21]:= <?xml version='1.0'?>
<!DOCTYPE Notebook SYSTEM '
  http://www.wolfram.com/XML/notebookml1.dtd'>
<Notebook CreatedBy='Mathematica 4.2'
  xmlns:xhtml='http://www.w3.org/1999/xhtml'
  xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
<List>
<Cell class='Text'>
<String>Some text</String>
<Style>
<String>Text</String>
</Style>
</Cell>
</List>
</Notebook>

```

Conversion Options for Export

You can use **ConversionOptions** to control the behavior of **Export** and **ExportString**. There are four conversion options available for exporting NotebookML.

- **"Annotations"** includes or excludes XML declarations, DOCTYPE declarations, and style advisories in any desired combination.
- **"BoxFormats"** specifies the export formats for various typeset box objects in the document.
- **"GraphicsFormats"** specifies the export formats for graphics objects in the document.
- **"Stylesheets"** will associate a style sheet (CSS or XSLT) for presentation on the web.

Annotations

The **Annotations** option takes a list whose elements can be any combination of the following values: **"DocumentHeader"**, **"DOCTYPEDeclaration"**, **"XMLDeclaration"**, and **"StyleAdvisories"**.

"DocumentHeader" determines if any header information should be added at the beginning of the NotebookML file. The **"DOCTYPEDeclaration"** and **"XMLDeclaration"** settings only take effect if **"DocumentHeader"** is specified as one of the annotations.

If **"DOCTYPEDeclaration"** is included in the list of annotations, then a DOCTYPE declaration is included in the header of the exported document. The default setting for **"Annotations"** includes **"DOCTYPEDeclaration"**, as shown in the following example.

```
In[22]:= ExportString[Cell["Some text", "Text"], "XML"]
```

```
Out[22]= <?xml version='1.0'?>
<!DOCTYPE Cell SYSTEM 'http://www.wolfram.com/XML/notebookxml1.dtd'>
<Cell class='Text'
  CreatedBy='Mathematica 4.2'
  xmlns:xhtml='http://www.w3.org/1999/xhtml'
  xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
  <String>Some text</String>
  <Style>
    <String>Text</String>
  </Style>
</Cell>
```

If you exclude **"DOCTYPEDeclaration"** from the list of **Annotations** values, the DOCTYPE declaration is omitted from the header of the exported document. Excluding the DOCTYPE declaration can sometimes shorten processing time taken to read the resulting XML document into an application, because the parser does not have to reference a DTD.

If **"XMLDeclaration"** is included in the list of annotation values, an XML declaration is also included in the header of the exported document. The default setting for **"Annotations"** includes **"XMLDeclaration"**, as shown in the following example.

```
In[23]:= ExportString[Cell["Some text", "Text"], "XML"]
```

```
Out[23]= <?xml version='1.0'?>
<!DOCTYPE Cell SYSTEM 'http://www.wolfram.com/XML/notebookxml1.dtd'>
<Cell class='Text'
  CreatedBy='Mathematica 4.2'
  xmlns:xhtml='http://www.w3.org/1999/xhtml'
  xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
  <String>Some text</String>
  <Style>
    <String>Text</String>
  </Style>
</Cell>
```

If **"XMLDeclaration"** is excluded from **Annotations**, it is also omitted from the header of the exported document. Excluding the XML declaration is useful if a user wishes to create a NotebookML fragment for insertion in another XML document.

```
In[24]:= ExportString[Cell["some text", "Text"], "XML", ConversionOptions →
{"Annotations" → {"DOCTYPEDeclaration", "StyleAdvisories"}}]
```

```
Out[24]= <Cell class='Text'
  CreatedBy='Mathematica 4.2'
  xmlns:xhtml='http://www.w3.org/1999/xhtml'
  xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
  <String>some text</String>
  <Style>
  <String>Text</String>
  </Style>
</Cell>
```

If **"StyleAdvisories"** is included in the list of annotations, class attributes are returned for **Cells** and **StyleBoxes** that have styles associated with them. The default setting for **"Annotations"** includes **"StyleAdvisories"**, as shown in the following example.

```
In[25]:= ExportString[Cell["Some text", "Text"], "XML"]
```

```
Out[25]= <?xml version='1.0'?>
<!DOCTYPE Cell SYSTEM 'http://www.wolfram.com/XML/notebookxml1.dtd'>
<Cell class='Text'
  CreatedBy='Mathematica 4.2'
  xmlns:xhtml='http://www.w3.org/1999/xhtml'
  xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
  <String>Some text</String>
  <Style>
  <String>Text</String>
  </Style>
</Cell>
```

If you exclude **"StyleAdvisories"** from **Annotations**, the class attributes associated with **Cell** are omitted.

```
In[26]:= ExportString[Cell["Some text", "Text"], "XML", ConversionOptions →
{"Annotations" → {"DOCTYPEDeclaration", "XMLDeclaration"}}]
```

```
Out[26]= <Cell CreatedBy='Mathematica 4.2'
  xmlns:xhtml='http://www.w3.org/1999/xhtml'
  xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
  <String>Some text</String>
  <Style>
  <String>Text</String>
  </Style>
</Cell>
```

BoxFormats

BoxFormats exports box data as NotebookML, GIF, or MathML. This option can be set to one or more of the following values: **Automatic**, **"GIF"**, or **"MathML"**. The default setting is **"BoxFormats"→Automatic**.

With **"BoxFormats"→Automatic**, the boxes will be losslessly exported as NotebookML. This is the only box format setting that is guaranteed to be lossless.

With **"BoxFormats"→GIF**, the box data will be represented as a GIF file. The exported NotebookML document will reference the exported GIF file by using an XHTML tag such as ****.

With **"BoxFormats"→MathML**, the box data will be exported as MathML, which will be embedded inside the NotebookML.

If **BoxFormats** has more than one value, then the **BoxData** parent element will have multiple child elements associated with it, one element per format.

For example, given some **BoxData** expression:

BoxData[{...}]

and the following values for **BoxFormats**

BoxFormats → {Automatic, GIF, MathML}

a NotebookML expression of the following form is generated.

```
<BoxData>
  <List>
    ...
  </List>
  <xhtml:img src="Images/myFile_1.gif"/>
  <math>
    ...
  </math>
</BoxData>
```

GraphicsFormats

The **"GraphicsFormats"** option changes the format of notebook graphics into another format for export. This is often useful since some external applications do not support *Mathematica* notebook graphics. This option can be set to one or more of the following values: **Automatic**, **"Bitmap"**, **"GIF"**, **"Metafile"**, **"PICT"**, **"PostScript"**, or **"QuickTime"**.

Automatic is the default setting for this option. It is necessary to set **"GraphicsFormats"→Automatic** to ensure lossless import and export of a notebook.

For the **"Bitmap"**, **"Metafile"**, **"PICT"**, **"PostScript"**, and **"QuickTime"** values, the original notebook graphic will be converted to the specified graphic type and then exported.

For the **"GIF"** setting, an external GIF file is created from the original notebook graphic. An **<xhtml:img src>** element is inserted in the exported file. Here is an example.

```
"GraphicsFormats" → {Automatic, "GIF"}
```

```
<GraphicsData>
  <PostScript>(some postscript data...)</PostScript>
  <xhtml:img src="Images/myFile_1.gif"/>
</GraphicsData>
```

If **"GraphicsFormats"** has more than one value, then multiple child elements are created for each specified format inside the **GraphicsFormats** parent element.

Stylesheets

You can associate your NotebookML documents with a style sheet (CSS or XSLT) by using the **Stylesheets** option. This option takes a list of rules that represent the pseudo-attributes in the XML-style sheet processing instruction. For more information about the various pseudo-attributes, see: www.w3.org/TR/xml-stylesheet.

The following is a list of rules for **"Stylesheets"**.

```
href
type
title
media
charset
alternate
```

Each of these rules must have a string as its value. For example, the following option

```
In[27]:= "StyleSheets" → {"href" → "mystyle.css", "type" → "text/css"}
```

```
Out[27]= StyleSheets → {href → mystyle.css, type → text/css}
```

would result in the following XML statement.

```
<?xml-stylesheet href="mystyle.css" type="text/css"?>
```

Here is an example that uses this conversion option.

```
In[28]:= ExportString[Cell["Some text"], "XML", ConversionOptions →
  {"StyleSheets" → {"href" → "mystyle.css", "type" → "text/css"}}]

Out[28]= <?xml version='1.0'?>
<!DOCTYPE Cell SYSTEM 'http://www.wolfram.com/XML/notebookml1.dtd'>
<?xml-stylesheet href="mystyle.css" type="text/css"?>
<Cell CreatedBy='Mathematica 4.2'
  xmlns:xhtml='http://www.w3.org/1999/xhtml'
  xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
  <String>Some text</String>
</Cell>
```

"Stylesheets" can also take a list of rules, each of which may take a list of values as a sublist. Each sublist corresponds to one **xml-stylesheet** processing instruction.

```

In[29]:= "StyleSheets" → {
  {"href" → "basicStyle.css", "type" → "text/css",
   "alternate" → "yes", "title" → "basic"},
  {"href" → "bigStyle.css", "type" → "text/css",
   "alternate" → "yes", "title" → "big"}}

Out[29]= StyleSheets → {{href → basicStyle.css,
  type → text/css, alternate → yes, title → basic},
  {href → bigStyle.css, type → text/css, alternate → yes, title → big}}

<?xml-stylesheet href="basicStyle.css" type="text/css" alternate="yes"
title="basic"?>
<?xml-stylesheet href="bigStyle.css" type="text/css" alternate="yes"
title="big"?>

```

Using a CSS style sheet, NotebookML can be displayed in many current generation browsers. For example, *Internet Explorer* 5 or later and *Netscape* 6 or later have built-in support for CSS. The style sheet can even mimic the behavior of *Mathematica*'s environments (like Working, Printout, Presentation, etc.). To save a notebook as NotebookML with a CSS style sheet, use the **Export** function with a conversion option of **"StyleSheets"** pointing to the relevant style sheet. Here is an example.

```

In[30]:= Export["NotebookML.xml", InputNotebook[],
  ConversionOptions → {"StyleSheets" →
    {"type" → "text/css", "href" → "css/presentation.css"}}]

Out[30]= NotebookML.xml

```

If you save a notebook as NotebookML and use a CSS style sheet that contains definitions for the various elements, the *resulting* file can be rendered in a web browser. The advantage of this approach (instead of simply converting the notebook to HTML) is that you only need to create a single document, which can be viewed either in web browsers or in *Mathematica*. A non-*Mathematica* user can view the document in any web browser. But a *Mathematica* user will be able to open and edit the document as a notebook, evaluate the input, manipulate the graphics, and so on.

■ Working with MathML

□ Introduction

MathML is an XML-based markup language for representing mathematics. It was developed by the W3C to provide an effective way to display math in web pages and to facilitate the transfer and reuse of mathematical content between applications. MathML can encode information about both the meaning and the appearance of mathematical notation. For example, a MathML equation can be copied out of a web page and directly pasted into an application like *Mathematica* for evaluation.

Wolfram Research was a key participant in the development of MathML and is committed to supporting this important web technology. *Mathematica* includes

full support for MathML 2.0, the latest version of the language. You can import MathML equations into a *Mathematica* notebook and evaluate them, or export equations from a notebook as MathML and paste them into an HTML document for viewing in a web browser. There are also several kernel commands for converting between MathML and the boxes and expressions used internally by *Mathematica* to represent mathematics.

□ Syntax of MathML

Overview

Since it is an XML application, the syntax rules of MathML are defined by the XML specification. Each MathML expression consists of a series of elements, written in the angle bracket syntax similar to HTML. Each element can take several attributes. The allowed elements and attributes are determined by the MathML DTD.

All MathML elements fall into one of three categories: interface elements, presentation elements, and content elements. Interface elements, such as the top-level **math** element, determine how a MathML expression is embedded in other XML documents.

Presentation elements encode information about the appearance of a mathematical expression, that is, its two-dimensional structure. For example, the **mrow**, **mfrac**, **msqrt**, and **msub** elements represent a row, a fraction, a square root, and a subscripted expression, respectively.

Content elements encode information about the logical meaning of a mathematical expression. For example, **plus** and **sin** represent addition and the trigonometric sine function, and **apply** represents the operation of applying a function.

A given equation can be represented in several different ways in MathML.

- Presentation elements only. This type of markup is called presentation markup. It is useful in situations where only the display of mathematics is important. For example, you can use presentation markup for displaying equations in a web page.
- Content elements only. This type of markup is called content markup. It is useful in situations where it is important to encode mathematical meaning. For example, you can use content markup in a web page to allow students to copy equations from their browser and paste it into *Mathematica* for evaluation.
- A combination of content and presentation elements. This type of markup is called combined markup and is used when you want to encode both the appearance and meaning of equations. For example, you can use combined markup to specify a nonstandard notation for a common mathematical construct or to associate a specific mathematical meaning with a certain type of notation that usually has a different meaning.

Using *Mathematica*, you can generate presentation markup, content markup, or combined markup for any equation.

Presentation MathML

Presentation MathML consists of about 30 elements and 50 attributes, which encode the visual two-dimensional structure of a mathematical expression. For example, the *Mathematica* typeset expression $x + 1$ would have the following MathML representation.

$x + 1$

```
<math>
  <mrow>
    <mi>x</mi>
    <mo>+</mo>
    <mn>1</mn>
  </mrow>
</math>
```

The entire expression is enclosed in a **math** element. This must be the root element for every instance of MathML markup. The above example also uses four other presentation elements.

- **mrow** displays its sub-elements in a horizontal row.
- **mi** represents an identifier, such as the name of a function or a variable.
- **mo** represents an operator or a delimiter.
- **mn** represents a number.

Identifiers, operators, and numbers are each represented by a different type of element because each type of object has slightly different typesetting conventions for fonts, spacing, and so on. For example, variables are typically rendered in an italic font, numbers are displayed in a normal font, and operators are rendered with extra space around them, depending on whether they occur in a prefix, postfix, or infix position.

In addition to the **mi**, **mn**, and **mo** elements, there are presentation elements corresponding to common notational structures, such as fractions, square roots, subscripts, superscripts, and matrices. Any given formula can then be represented by decomposing it into its constituent parts and replacing each notational construct by the corresponding presentation elements. For example, the typeset expression $\frac{\sqrt{x}}{y^2 - 1}$ would have the following MathML representation.

$$\frac{\sqrt{x}}{y^2 - 1}$$

```
<math>
  <mfrac>
    <msqrt>
      <mi>x</mi>
    </msqrt>
    <mrow>
      <msup>
```

```

<mi>y</mi>
<mn>2</mn>
</msup>
<mo>-</mo>
<mn>1</mn>
</mrow>
</mfrac>
</math>

```

Here, the **mfrac**, **msqrt**, and **msup** elements represent a fraction, a square root, and a superscripted expression. Each of these elements takes a fixed number of child elements, which have a specific meaning based on their position. These child elements are called *arguments*. For example, both the **mfrac** and **msup** elements take two arguments with the following syntax.

```

<mfrac> numerator denominator </mfrac>
<msup> base superscript </msup>

```

As before, the **mrow** element is used to enclose other elements that need to be displayed in a horizontal row. For example, the typeset expression $\int e^{-x} dx$ would have the following MathML representation.

```

<math>
<mrow>
  <msubsup>
    <mo>&int;</mo>
    <mn>0</mn>
    <mi>&inf;</mi>
  </msubsup>
  <msup>
    <mi>&exp;</mi>
    <mrow>
      <mo>-</mo>
      <mi>x</mi>
    </mrow>
  </msup>
  <mrow>
    <mo>&dd;</mo>
    <mi>x</mi>
  </mrow>
</mrow>
</math>

```

Here, the limits of the integral are shown using the presentation element **msubsup**, which takes three arguments, with the following syntax.

```

<msubsup> base subscript superscript </msubsup>

```

Another notable feature is that the symbols representing the integral sign, the exponential, and the differential *d* are represented using the character entities **∫**, **&exp;**, and **ⅆ**, respectively. These are among approximately two

thousand special symbols defined by the MathML DTD. These can be included in a document using a named entity reference or a character entity reference that uses the Unicode character code for that symbol.

The **mstyle** element is used for applying styles to an equation. Any attributes specified in an **mstyle** element are inherited by all its child elements. Hence, you can use this element to specify properties like the font size and color for an equation, as in the example below. Note the use of the entity **⁢** to denote multiplication.

$$x^2 - 2x + 1$$

```
<math>
<mstyle mathsize='10 pt' mathcolor='red'>
  <mrow>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <mo>-</mo>
    <mrow>
      <mn>2</mn>
      <mo>&InvisibleTimes;</mo>
      <mi>x</mi>
    </mrow>
    <mo>+</mo>
    <mn>1</mn>
  </mrow>
</mstyle>
</math>
```

The above examples are intended only to illustrate how presentation markup works through a sampling of some of its elements. To see a complete listing of all the presentation elements and attributes, see the MathML specification at www.w3.org/TR/MathML2.

Content MathML

Content MathML consists of about 140 elements and 12 attributes, which encode the logical meaning of a mathematical expression. The content elements **ci** and **cn** are used to represent identifiers and numbers, respectively. They are analogous to the **mi** and **mn** elements in presentation markup. For example, the typeset expression $x + 1$ would have the following content MathML representation.

$$x + 1$$

```
<math>
<apply>
  <plus/>
  <ci>x</ci>
  <cn>1</cn>
</apply>
</math>
```

The **apply** element is used to apply operators or functions to expressions. The first argument of the **apply** element is usually an empty element indicating an operator or function. The remaining arguments represent one or more expressions to which the first argument is applied. In the above example, the first argument of the **apply** function is the empty element **plus**, which denotes addition.

The **type** attribute of **cn** describes the type of number encoded. It can take values **real**, **integer**, **rational**, **complex-polar**, **complex-cartesian**, and **constant**. The empty element **sep** is used to separate different parts of a number, such as the numerator and denominator of a fraction or the real and imaginary parts of a complex number. For example:

$$\frac{1}{2}$$

```
<math>
  <cn type="rational">1<sep/>2</cn>
</math>
```

$$3 + 4i$$

```
<math>
  <cn type="complex-cartesian">3<sep/>4</cn>
</math>
```

The majority of content elements are empty elements representing specific operators or functions. The various elements are organized into groups named after the following specific elementary subfields of mathematics:

- Arithmetic, Algebra, and Logic
- Elementary Functions
- Calculus
- Linear Algebra
- Sequences and Series
- Theory of Sets
- Statistics

There are elements corresponding to most operators and functions that are encountered in high school mathematics. For example, basic arithmetic operators are represented by **plus**, **minus**, **times**, **divide**, and **power**.

$$\frac{\sqrt{x}}{y^2 - 1}$$

```
<math>
  <apply>
    <times/>
    <apply>
      <power/>
      <ci>x</ci>
    </apply>
  </times>
  <power/>
  <ci>y</ci>
  <minus/>
  <ci>1</ci>
</math>
```

```

<cn>1<sep/>2</cn>
</apply>
<apply>
  <power/>
  <apply>
    <plus/>
    <apply>
      <power/>
      <ci>y</ci>
    <cn>2</cn>
  </apply>
  <cn>-1</cn>
</apply>
<cn>-1</cn>
</apply>
</apply>
</math>

```

Integrals are specified using the **int** element. The variable of integration is represented using the element **bvar**. The upper and lower limits of integration are usually specified using the elements **lowlimit** and **uplimit**.

$$\int_a^b x \, dx$$

```

<math>
<apply>
  <int/>
  <bvar>
    <ci>x</ci>
  </bvar>
  <lowlimit>
    <ci>a</ci>
  </lowlimit>
  <uplimit>
    <ci>b</ci>
  </uplimit>
  <ci>x</ci>
</apply>
</math>

```

The **interval** element is used to specify closed and open intervals. It takes the attribute **closure**, which can take the values **closed**, **open**, **closed-open**, and **open-closed** corresponding to the four types of intervals possible. The default value for **closure** is **closed**.

[a, b]

```

<math>
<interval>
  <ci>a</ci>
  <ci>b</ci>
</interval>
</math>

```

(0, 1]

```
<math>
  <interval closure='open-closed'>
    <cn>0</cn>
    <cn>1</cn>
  </interval>
</math>
```

You can also use the **interval** element to specify the limits of a definite integral as an alternative to using **uplimit** and **lowlimit**.

$$\int_a^b x \, dx$$

```
<math>
  <apply>
    <int/>
    <bvar>
      <interval>
        <ci>a</ci>
        <ci>b</ci>
      </interval>
    <ci>x</ci>
  </apply>
</math>
```

The **matrix** and **matrixrow** elements are used to represent a matrix and a row of a matrix, respectively. The **eq** element is used to express equality.

$$M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

```
<math>
  <apply>
    <eq/>
    <ci>A</ci>
    <matrix>
      <matrixrow>
        <ci>a</ci>
        <ci>b</ci>
      </matrixrow>
      <matrixrow>
        <ci>c</ci>
        <ci>d</ci>
      </matrixrow>
    </matrix>
  </apply>
</math>
```

The above examples are intended only to illustrate how content markup works through a representative sampling of some of its elements. To see a complete listing of all the content elements and attributes, see the MathML specification at www.w3.org/TR/MathML2.

□ Importing MathML

There are two ways to import MathML equations into *Mathematica*.

- Copy and paste MathML equations from another application, such as a web browser, directly into a notebook. When you paste a valid MathML expression into a notebook, *Mathematica* brings up a dialog box asking if you want to paste the literal markup or interpret it. If you choose to interpret the markup, it is automatically converted into a *Mathematica* expression.
- Use the **Import** command with **"MathML"** as the import format.

```
In[1]:= Import["test.mml", "MathML"]
```

```
Out[1]= FormBox[TagBox[SuperscriptBox[x, 2],
  MathMLPresentationTag, AutoDelete -> True], TraditionalForm]
```

By default, MathML markup is imported as a *Mathematica* box expression. You can convert the boxes into an expression using the **ToExpression** command.

```
In[2]:= ToExpression[%]
```

```
Out[2]= x2
```

MathML Conversion Options

The standard **ConversionOptions** feature of the **Import** function can be used for greater control over the export process. The syntax for specifying a conversion option is as follows.

```
Import[file, expr, "MathML", ConversionOptions ->
  {option1 -> value1, option1 -> value2, ...}]
```

For more information on the conversion options available for importing MathML, see XML Import Conversion Options.

□ Generating MathML

Mathematica includes several functions for generating MathML from the boxes and expressions used internally by *Mathematica* to represent equations. You can enter an equation in a notebook using palettes, menus, or keyboard shortcuts and then convert it into MathML using one of these conversion functions. All the MathML conversion functions are located in the **XML`MathML`** context.

You can use **BoxesToMathML** to generate MathML from a box structure. By default, this generates presentation markup only.

```
In[3]:= XML`MathML`BoxesToMathML[SuperscriptBox["x", "2"]]
```

```
Out[3]= <math xmlns='http://www.w3.org/1998/Math/MathML'>
  <msup>
    <mi>x</mi>
    <mn>2</mn>
  </msup>
</math>
```

Alternatively, you can use **ExpressionToMathML** to convert a typeset equation in a notebook into MathML. By default, this generates combined markup with both the presentation markup and content markup for the equation enclosed in a **semantics** element.

```
In[4]:= XML`MathML`ExpressionToMathML[x^2]

Out[4]= <math xmlns='http://www.w3.org/1998/Math/MathML'>
  <semantics>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <annotation-xml encoding='MathML-Content'>
      <apply>
        <power/>
        <ci>x</ci>
        <cn type='integer'>2</cn>
      </apply>
    </annotation-xml>
  </semantics>
</math>
```

The **annotation-xml** element is used to provide additional information of a type specified by its encoding attribute. Here, the encoding attribute has the value **"MathML-Content"**, indicating that the **annotation-xml** element contains content MathML.

You can use the option **"Formats"** to generate either presentation MathML or content MathML only. You can also set the option **"Annotations"→{}** to suppress the header information.

```
In[5]:= XML`MathML`ExpressionToMathML[x^2 + 1, "Annotations" → {},
  "Formats" → {"PresentationMathML"}]

Out[5]= <math xmlns='http://www.w3.org/1998/Math/MathML'>
  <mrow>
    <msup>
      <mi>x</mi>
      <mn>2</mn>
    </msup>
    <mo>+</mo>
    <mn>1</mn>
  </mrow>
</math>
```

Note that **ExpressionToMathML** evaluates its first argument before converting it to MathML. Hence, if you supply an expression that can be simplified on evaluation, you may get unexpected results. For example, suppose you want to generate the presentation markup for the following definite integral.

$$\text{In}[6]:= \int_0^{\infty} e^{-x} dx$$

$\text{Out}[6]= 1$

Because the integral evaluates to give the result 1, the following command generates the MathML representation of 1 instead of the integral.

```
 $\text{In}[7]:= \text{XML} \backslash \text{MathML} \backslash \text{ExpressionToMathML} \left[ \int_0^{\infty} e^{-x} dx, \text{"Annotations"} \rightarrow \{\}, \right. \\ \left. \text{"Formats"} \rightarrow \{\text{"PresentationMathML"}\} \right]$ 
```

```
 $\text{Out}[7]= <\text{math xmlns}=\text{'http://www.w3.org/1998/Math/MathML'}> \\ <\text{mn}>1</\text{mn}> \\ </\text{math}>$ 
```

To get the MathML representation of the integral, you must force the integral to remain unevaluated by wrapping the **Unevaluated** function around it.

```
 $\text{In}[8]:= \text{XML} \backslash \text{MathML} \backslash \text{ExpressionToMathML} \left[ \right. \\ \text{Unevaluated} \left[ \int_0^{\infty} e^{-x} dx \right], \text{"Annotations"} \rightarrow \{\}, \\ \left. \text{"Formats"} \rightarrow \{\text{"PresentationMathML"}\} \right]$ 
```

```
 $\text{Out}[8]= <\text{math xmlns}=\text{'http://www.w3.org/1998/Math/MathML'}> \\ <\text{mrow}> \\ <\text{msubsup}> \\ <\text{mo}>\&\#8747;</\text{mo}> \\ <\text{mn}>0</\text{mn}> \\ <\text{mi}>\&\#8734;</\text{mi}> \\ </\text{msubsup}> \\ <\text{mrow}> \\ <\text{msup}> \\ <\text{mi}>\&\#8519;</\text{mi}> \\ <\text{mrow}> \\ <\text{mo}>-</\text{mo}> \\ <\text{mi}>x</\text{mi}> \\ </\text{mrow}> \\ </\text{msup}> \\ <\text{mo}>\&\#8290;</\text{mo}> \\ <\text{mrow}> \\ <\text{mo}>\&\#8518;</\text{mo}> \\ <\text{mi}>x</\text{mi}> \\ </\text{mrow}> \\ </\text{mrow}> \\ </\text{math}>$ 
```

Setting Options

These are all the functions that generate MathML as output.

- **BoxesToMathML**
- **ExpressionToMathML**
- **BoxesToSymbolicMathML**
- **ExpressionToSymbolicMathML**
- **Import**

These functions all accept the following options.

- **"Annotations"**
- **"Formats"**
- **"NamespacePrefixes"**

Using these options, you can control various features of the generated MathML, such as including an XML declaration or DTD declaration, generating presentation markup, content markup, or both, and using an explicit namespace declaration and namespace prefix.

You can specify the options explicitly each time you evaluate one of the MathML functions. Alternatively, you can use the **SetOptions** command to change the default values of the options for a particular function. The option values you set are then used for all subsequent evaluations of that function.

For example, evaluating the following command ensures that the output for all subsequent evaluations of **ExpressionToMathML** will, by default, generate presentation MathML only and omit the header information in the output.

```
In[9]:= SetOptions[XML'MathML'ExpressionToMathML,
  "Annotations" -> {}, "Formats" -> {"PresentationMathML"}]

Out[9]:= {Annotations -> {}, CheckXML -> True,
  ElementFormatting -> Automatic, Entities -> None,
  Formats -> {PresentationMathML}, NamespacePrefixes -> {}}
```

□ Exporting MathML

Introduction

You can use *Mathematica*'s sophisticated typesetting capabilities to create properly formatted equations and then convert them into MathML for display on the web. There are several ways to export mathematical expressions from a *Mathematica* notebook as MathML.

- Use the Edit ▸ Copy As ▸ MathML menu command. This copies the selected expression into the Clipboard in MathML format. This is a convenient way to copy a specific mathematical formula from a notebook and paste it into an HTML document.
- Use the File ▸ Save As Special ▸ HTML+MathML menu command. This converts your entire notebook into HTML with all equations in the notebook saved as MathML. The equations are embedded in the

HTML file in the form of MathML “data islands,” which can be displayed by a web browser, either directly or using a special plug-in.

- Use the File > Save As Special > XML (NotebookML+MathML) menu command. This converts your entire notebook into NotebookML with all equations in the notebook saved as MathML.
- Use the **Export** function, with **"MathML"** as the export format or with **.mml** as the file extension.

```
In[10]:= Export["test", x2, "MathML"]
```

```
Out[10]= test
```

```
In[11]:= !! test
```

```
<math xmlns='http://www.w3.org/1998/Math/MathML'>
<semantics>
<msup>
<mi>x</mi>
<mn>2</mn>
</msup>
<annotation-xml encoding='MathML-Content'>
<apply>
<power/>
<ci>x</ci>
<cn type='integer'>2</cn>
</apply>
</annotation-xml>
</semantics>
</math>
```

```
In[12]:= Export["test.mml", x2]
```

```
Out[12]= test.mml
```

```
In[13]:= !! test.mml
```

```
<math xmlns='http://www.w3.org/1998/Math/MathML'>
<semantics>
<msup>
<mi>x</mi>
<mn>2</mn>
</msup>
<annotation-xml encoding='MathML-Content'>
<apply>
<power/>
<ci>x</ci>
<cn type='integer'>2</cn>
</apply>
</annotation-xml>
</semantics>
</math>
```

The resulting output contains both presentation markup and content markup for the expression, enclosed in a **semantics** element. You can choose to generate either presentation markup or content markup by changing the value of the option **"Formats"**. Note that the **xmlns** attribute is added to the top-level **math** element to provide information about the namespace of the enclosed elements.

Symbols for MathML Elements

Certain content elements in MathML do not have a direct analog in *Mathematica*. Therefore, a few symbols are specially defined in the **XML`MathML`** context to represent these elements. These symbols are listed below.

In[14]:= ? XML`MathML`Symbols`*

XML`MathML`Symbols`		
Cardinality	Declare	IntervalClosedOpen
CartesianProduct	DerivativeIndex	IntervalOpen
Codomain	Divides	IntervalOpenClosed
ConditionedIntegrate	Domain	Moment
ConditionedLimit	DomainIntegrate	MomentAbout
ConditionedList	DomainOfApplication	Multiscripts
ConditionedMax	DomainProduct	Otherwise
ConditionedMin	DomainSum	Piece
ConditionedProduct	Image	Piecewise
ConditionedSet	IndefiniteProduct	\$Unparsable
ConditionedSum	IndefiniteSum	
CovariantDerivativeIndex	IntervalClosed	

MathML Conversion Options

The standard **ConversionOptions** feature of the **Export** function can be used for greater control over the export process. The syntax for specifying a conversion option is as follows.

Export[*file*, *expr*, "MathML", **ConversionOptions** →
{*option1* → *value1*, *option2* → *value2*, ...}]

The following conversion options are specially useful when exporting MathML.

- **"Annotations"** controls whether to include an XML declaration or DTD declaration in the exported file.
- **"Formats"** controls whether to export presentation MathML, content MathML, or both.
- **"NamespacePrefixes"** controls whether to include a namespace prefix for each MathML element.
- **"UseUnicodePlane1Characters"** controls whether to include characters belonging to plane 1 of Unicode or to replace them with corresponding plane 0 characters.

Annotations

This option controls which annotations are added to the output MathML. The value of this option is a list whose elements can be **"DocumentHeader"**, **"XMLDeclaration"**, or **"DOCTYPEDeclaration"**. The order of the elements in the list is not relevant.

XMLDeclaration

When **"XMLDeclaration"** is one of the annotations, then an XML declaration is included in the header. This means that the statement `<?xml version="1.0"?>` appears in the header.

```
In[15]:= Export["test.mml", x^2, "MathML",
  ConversionOptions -> {"Formats" -> {"PresentationMathML"},
    "Annotations" -> {"DocumentHeader", "XMLDeclaration"}}]
```

```
Out[15]= test.mml
```

```
In[16]:= !! test.mml
```

```
<?xml version='1.0'?>
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <msup>
    <mi>x</mi>
    <mn>2</mn>
  </msup>
</math>
```

DOCTYPEDeclaration

When **"DOCTYPEDeclaration"** is one of the annotations, then an XML document type declaration of the form `<!DOCTYPE ... >` appears in the header. This is a statement that specifies the DTD for the XML application in which the output is written.

```
In[17]:= Export["test.mml", x^2, "MathML",
  ConversionOptions -> {"Formats" -> {"PresentationMathML"},
    "Annotations" -> {"DocumentHeader", "DOCTYPEDeclaration"}}]
```

```
Out[17]= test.mml
```

```
In[18]:= !! test.mml
```

```
<!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0/EN"
'http://www.w3.org/TR/MathML2/dtd/mathml2.dtd'>
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <msup>
    <mi>x</mi>
    <mn>2</mn>
  </msup>
</math>
```

DocumentHeader

With the setting **"Annotations" → {"DocumentHeader", "XMLDeclaration", "DOCTYPEDeclaration"}**, a header containing an XML declaration and a document type declaration for the MathML DTD is automatically added to the output.

```
In[19]:= Export["test.mml", x^2, "MathML", ConversionOptions →
  {"Formats" → {"PresentationMathML"}, "Annotations" →
    {"DocumentHeader", "DOCTYPEDeclaration", "XMLDeclaration"}}]

Out[19]= test.mml

In[20]:= !! test.mml
```

```
<?xml version='1.0'?>
<!DOCTYPE math PUBLIC "-//W3C//DTD MathML 2.0//EN"
'http://www.w3.org/TR/MathML2/dtd/mathml2.dtd'>
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <msup>
    <mi>x</mi>
    <mn>2</mn>
  </msup>
</math>
```

When **"Annotations"** does not contain **"DocumentHeader"**, then the output MathML has no header. This is true even if the **"Annotations"** contains other elements, such as **"XMLDeclaration"** or **"DOCTYPEDeclaration"**. Thus, **"DocumentHeader"** is an overall switch that controls whether the structure has a header or not.

```
In[21]:= Export["test.mml", x^2, "MathML",
  ConversionOptions → {"Formats" → {"PresentationMathML"},
    "Annotations" → {"DOCTYPEDeclaration", "XMLDeclaration"}}]

Out[21]= test.mml

In[22]:= !! test.mml
```

```
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <msup>
    <mi>x</mi>
    <mn>2</mn>
  </msup>
</math>
```

Formats

This option controls which type of MathML markup is generated. The value of the option is a list that can have as its elements **"PresentationMathML"**, or **"ContentMathML"**, or both. The default setting is **"Formats" → {"PresentationMathML", "ContentMathML"}**, which generates both presentation and content MathML in parallel.

The following command exports presentation MathML only.

```
In[23]:= ExportString[x2, "MathML",
  ConversionOptions → {"Formats" → {"PresentationMathML"}}]

Out[23]= <math xmlns='http://www.w3.org/1998/Math/MathML'>
  <msup>
    <mi>x</mi>
    <mn>2</mn>
  </msup>
</math>
```

This command exports content MathML only.

```
In[24]:= ExportString[x2, "MathML",
  ConversionOptions → {"Formats" → {"ContentMathML"}}]

Out[24]= <math xmlns='http://www.w3.org/1998/Math/MathML'>
  <apply>
    <power/>
    <ci>x</ci>
    <cn type='integer'>2</cn>
  </apply>
</math>
```

NamespacePrefixes

This conversion option lets you generate XML markup with a specific namespace declaration and namespace prefixes. The option is specified in the form

"NamespacePrefixes" → {url → prefix}

where *url* and *prefix* are strings specifying the URL of the namespace and the namespace prefix. In the following example, the **"NamespacePrefix"** option is used to generate presentation markup with each MathML element having a namespace prefix **"mml"** associated with the MathML namespace.

```
In[25]:= ExportString[x2, "MathML", ConversionOptions →
  {"Formats" → {"PresentationMathML"}, "NamespacePrefixes" →
  {"http://www.w3.org/1998/Math/MathML" → "m"}}]

Out[25]= <m:math xmlns:m='http://www.w3.org/1998/Math/MathML'
  xmlns='http://www.w3.org/1998/Math/MathML'>
  <m:msup>
    <m:mi>x</m:mi>
    <m:mn>2</m:mn>
  </m:msup>
</m:math>
```

UseUnicodePlane 1 Characters

This conversion option controls whether characters belonging to plane 1 of Unicode should be replaced with similar characters in plane 0. This option is useful because currently most browsers cannot properly display plane 1 characters.

<i>option</i>	<i>value</i>	<i>effect</i>
"UseUnicodePlane1Characters"	True	special characters belonging to plane 1 of Unicode are exported as is
	False	special characters belonging to plane 1 of Unicode are replaced by plane 0 characters with an attached <code>mathvariant</code> attribute

The option **"UseUnicodePlane1Characters"** and its possible values.

With the default setting, **"UseUnicodePlane1Characters"→True**, special characters that belong to plane 1 of Unicode (such as Gothic, scripted, and double-struck characters), are written out with their plane 1 numeric character codes. Here is an example involving the Gothic "g" character.

```
In[26]:= ExportString[g, "MathML"]
```

```
Out[26]= <math xmlns='http://www.w3.org/1998/Math/MathML'>
  <semantics>
    <mi>&#120100;</mi>
    <annotation-xml encoding='MathML-Content'>
      <ci>&#120100;</ci>
    </annotation-xml>
  </semantics>
</math>
```

With the setting, **"UseUnicodePlane1Characters"→False**, any special character that belongs to plane 1 of Unicode, is replaced by a corresponding plane 0 character with a suitable value of the `mathvariant` attribute specified.

```
In[27]:= ExportString[g, "MathML",
  ConversionOptions → {"UseUnicodePlane1Characters" → False}]
```

```
Out[27]= <math xmlns='http://www.w3.org/1998/Math/MathML'>
  <semantics>
    <mi>&#120100;</mi>
    <annotation-xml encoding='MathML-Content'>
      <ci>&#120100;</ci>
    </annotation-xml>
  </semantics>
</math>
```

■ Acknowledgments

Jason Harris and Thomasanna Hail were helpful in reviewing and editing the content of this article. The examples in the section on transforming XML were contributed by Buddy Ritchie, Chris Hill, and Shaun McCance.

■ Additional Material

The following data files, used in some of the examples, are available at www.mathematica-journal.com.

ALHitters.xml. An XML file containing data on baseball hitters.

NLPitchers.xml. Another XML file containing data on baseball pitchers.

methane.xml. A file specifying the structure of the methane molecule in MoDL syntax.

Shakespeare.nb. A notebook containing a quotation from Shakespeare.

About the Author

Pavi Sandhu is a technical writer and manager at Wolfram Research and a member of the W3C Math Working Group. He is also the author of *The MathML Handbook*, published by Charles River Media in November 2002.

Pavi Sandhu

Technical Writing Manager
Wolfram Research, Inc.
pavi@wolfram.com

Building User Interfaces Using J/Link

Todd Gayley

Java is a powerful and enormously popular programming environment, and *Mathematica*'s built-in *J/Link* toolkit integrates Java and *Mathematica* very tightly. This article presents a quick-start introduction to creating *Mathematica* user interfaces using *J/Link*. By the end, you will have been introduced to all the major techniques and be ready to create your own interfaces.

■ Introduction

By some measures, Java is the most popular programming language in the world. It features a vast and ever-growing library of classes for just about anything you would want to do with a computer—everything from image and sound processing to database access to internet protocols to advanced user interfaces. Virtually every new computing technology nowadays shows up first in a Java-language implementation. Now imagine for a moment that all of these Java classes were written in *Mathematica* instead, so that all the power and functionality they provide was available directly to *Mathematica* programmers. Would that change the rules about what could be done in *Mathematica*?

Welcome to *J/Link*, a product that integrates *Mathematica* and Java so tightly that it is as if everything in Java was actually written in *Mathematica*. *J/Link* lets you load Java classes into *Mathematica*, create objects of those classes, and call methods on them. You can do this with any Java class, without any modification or preparation whatsoever. And you can do it all from the comfortable, integrated, line-at-a-time development environment provided by *Mathematica*.

Java is good for many things, and one of its more popular uses is for building platform-independent user interfaces. This makes it an ideal toolkit for a commonly sought capability—creating user interface elements for *Mathematica* programs. You might want something as simple as a progress bar or a dialog box with a few fields for entering input values, or something as complex as a wizard for leading users through the steps of a long computation. *J/Link* gives you the full power of Java to build these types of interfaces and more, using just *Mathematica* code, not Java.

This article presents a quick-start introduction to creating *Mathematica* user interfaces using *J/Link*. It can be read and worked through input by input in a single sitting. By the end, you will have been introduced to all the major techniques and you will be ready to create your own interfaces. *J/Link* includes an extensive User Guide that covers all the information in this article, and much more. You can find it in the Help Browser under the Add-ons category, and you should consult it for more detailed examples and explanations. There are also a number of example programs included with *J/Link* that demonstrate user interface techniques. These can be found in the **<Mathematica directory>/AddOns/.JLink/Examples/Part1** directory, and they are discussed in Section 1.3 of the User Guide.

Our focus here is on user interfaces designed to be used in conjunction with the notebook front end. That is, we will be creating *Mathematica* programs that, when run from a typical *Mathematica* session, cause dialog boxes or other windows to “pop up” on the screen for the user to interact with. It is also possible to use *J/Link* to write Java programs that call on *Mathematica* in the background for computational services. That involves going in the other direction—calling the *Mathematica* kernel from Java code. Such programs are the subject of the second half of the *J/Link* User Guide, and we will not be dealing with them in this article.

■ *J/Link* Basics

In this section we will examine the basic concepts of *J/Link* programming. My intent is to cover just enough for readers to become comfortable with the user interface examples developed later. I will make occasional references to specific sections in the *J/Link* User Guide. These references will be given as section numbers, and you can use these section numbers in the Go To field of the Help Browser. If a reference is to Section 1.2.1, for example, you can type “*J/Link* 1.2.1” (without the quotation marks) in the text field at the top and click the Go To button. You must make sure that the Add-ons button is depressed when you do this, as the User Guide is in the Add-ons section of the Help Browser. If you are reading this article as a notebook, then of course you can just click the links and be taken to the appropriate pages in the Help Browser automatically.

□ What is *J/Link*?

J/Link is a toolkit that integrates *Mathematica* and Java. It lets you call Java from *Mathematica* and *Mathematica* from Java. It sits on top of *MathLink*, Wolfram Research’s protocol for communicating with the *Mathematica* kernel. When you use *J/Link* to call *Mathematica* from a Java program, you can think of it as a *MathLink* Developer Kit for Java. When you use it to call Java from *Mathematica*, as in this article, it is just a *Mathematica* package that provides functionality for launching and controlling a Java runtime. You simply load the package and start writing programs that use it. The internals of *J/Link* are written in a combination

of *Mathematica*, Java, and C. All the source code is provided for those who might be interested, and can be found in <Mathematica/AddOns/JLink/Source and <Mathematica directory>/AddOns/JLink/Kernel (where the *Mathematica* code resides).

Beginning with *Mathematica* 4.2, *J/Link* is a standard part of the *Mathematica* system, not an add-on that you or your users need to download and install (although this is necessary in *Mathematica* 4.0 and 4.1). A Java Runtime Environment is also installed with *Mathematica* 4.2, so *J/Link* is ready to use right out of the box. This means that you can write *J/Link* programs for others and know that they will work without any setup on the user's part. They might be completely unaware that Java is even being used. Those with *Mathematica* 4.0 and 4.1 need to install *J/Link* and a Java Runtime Environment on their own, although many people will already have an appropriate Java system on their machines. *J/Link* includes installation instructions and information on obtaining a Java runtime for those with *Mathematica* 4.0 and 4.1. The current version of *J/Link* is 2.0, and that is the version assumed for this article. The home page for *J/Link* is www.wolfram.com/solutions/mathlink/jlink.

□ Starting the Java Runtime

The first step in using *J/Link* is to load the package.

```
In[1]:= Needs["JLink`"]
```

Java programs execute in a managed environment called a Java Runtime Environment, or just a "Java runtime." Before you can use *J/Link* from *Mathematica*, you must launch the Java runtime. All calls from *Mathematica* into Java are directed to this single Java runtime, which runs as a separate process from the *Mathematica* kernel. *MathLink* is used to communicate between the kernel and the Java runtime, but you do not have to be concerned with such details.

The function that launches Java is called **InstallJava**, named after the existing *Mathematica* function **Install**, which launches an external program and prepares it to be used via *MathLink*.

```
In[2]:= InstallJava[]
```

```
Out[2]= LinkObject[d:\math42\SystemFiles\Java\Windows\bin\javaw, 2, 2]
```

J/Link also has functions for shutting down the Java runtime (**UninstallJava**) and for restarting it (**ReinstallJava**). Do not use them. Developers need to remember that the Java runtime used by *J/Link* is a shared resource. Other *Mathematica* packages you have loaded might also be using *J/Link*, and shutting down Java will cause problems for them. Start the Java runtime when you need it and then just leave it running.

If Java is already running when you call **InstallJava**, the function does nothing. This means that it is safe to call it whenever you want, such as at the start of any programs that need *J/Link*.

□ Creating Objects and Calling Methods

Now let us begin calling into Java. An obvious first step is to create a Java object. In Java, you use the `new` operator for this. In *Mathematica*, you use the **JavaNew** function. Here we create an object of the `Date` class.

```
In[3]:= dateObject = JavaNew["java.util.Date"]
```

```
Out[3]= «JavaObject[java.util.Date] »
```

The first argument to **JavaNew** is the full name of the class with all the periods. Any arguments to the constructor would be given in a sequence after the class name, but the constructor being invoked here takes no arguments. It creates a `Date` object that represents the current time. Note that the return value is a funny expression surrounded by angle brackets. The angle brackets remind you that this is a special output form, not the true *Mathematica* representation of the expression. A **JavaObject** expression, as this is called, is a *Mathematica* representation of a Java object that lives in the Java runtime.

How did I know the name of the class and the arguments to the constructor I wanted to invoke? Obviously, I read the Java documentation. To use *J/Link* you are not going to have to learn the Java language, but you will have to learn what Java classes are available and what methods they contain. There are many books describing the standard Java class libraries, and for some uses you can get by with just the so-called “JavaDoc” help files. JavaDoc is a hyperlinked web page format for supplying help files for Java classes. A complete JavaDoc reference to the standard Java class libraries (including the user interface classes) can be found at java.sun.com/j2se/1.4/docs/api.

If you look at the documentation for the `java.util.Date` class, you will see that it has a method called `toString()` (all Java objects have this method). Here is how to call the `toString()` method.

```
In[4]:= dateObject@toString[]
```

```
Out[4]= Wed Feb 05 01:55:52 CST 2003
```

If you are familiar with Java, you would know that to call this method you would write in Java:

```
// Java code
dateObject.toString();
```

It looks almost the same in *Mathematica* except that we use the `@` symbol instead of the dot, and we use square brackets instead of parentheses. Throughout *J/Link*, we use the `@` like Java’s dot, to mean “call this method (or access this field) on the preceding object.” Below is a table taken from Section 1.1.6 of the User Guide that compares the *Mathematica* and Java syntaxes for all the different ways you use Java objects.

Constructors	
Java:	<code>MyClass obj = new MyClass(args);</code>
<i>Mathematica</i> :	<code>obj = JavaNew["MyClass", args];</code>
Methods	
Java:	<code>obj.methodName(args);</code>
<i>Mathematica</i> :	<code>obj@methodName[args];</code>
Fields	
Java:	<code>obj.fieldName = 1; value = obj.fieldName;</code>
<i>Mathematica</i> :	<code>obj@fieldName = 1; value = obj@fieldName;</code>
Static methods	
Java:	<code>MyClass.staticMethod(args);</code>
<i>Mathematica</i> :	<code>MyClass`staticMethod[args];</code>
Static fields	
Java:	<code>MyClass.staticField = 1; value = MyClass.staticField;</code>
<i>Mathematica</i> :	<code>MyClass`staticField = 1; value = MyClass`staticField;</code>

If this chart is hard to understand or the *Mathematica* syntax looks unfamiliar, do not worry. After just a few minutes of writing *J/Link* code yourself, it will become very natural.

We have already seen an example of how to call a method on an object (a so-called “instance” method). Classes can also have static methods and fields, which are associated with the class itself and not any specific object of that class. For example, Java has a class called `Math` that contains a number of mathematical functions like `abs()`, `sqrt()`, `sin()`, and so on. You do not ever create an object of type `Math` and call methods on it. It is just a container for a collection of static methods that operate on numbers.

To call a static method, the first thing you must do is load the class into *Mathematica*. Any Java class you use with *J/Link* must be loaded into *Mathematica*, but this happens automatically for you when you call `JavaNew` to create the first object of that class. When calling a static method, however, you are not using an object of that class, so the class might not have been loaded yet. You have to do it manually using `LoadJavaClass`.

```
In[5]:= LoadJavaClass["java.lang.Math"]
```

```
Out[5]= JavaClass[java.lang.Math, 2]
```

When you call a static method you use the class name as a prefix, separated from the method name by a context mark.

```
In[6]:= Math`cos[N[Pi]]
```

```
Out[6]= -1.
```

You will note that static methods and fields are the exception to the rule that dots in Java become @ in *Mathematica*. Although this sometimes causes confusion for novice *J/Link* programmers, it is actually quite natural. Java uses the dot in two very different ways—as the member access operator, as in `obj.method()`, and as the scope resolution operator, as in `package.Class.staticMethod()`. Some other object-oriented languages use different operators for these two meanings (C++ uses `::` as the scope resolution operator). Although it is not often described in this way, the context mark is *Mathematica*'s scope resolution operator. In other words, do not complain that *Mathematica* requires two different operators, complain that Java overloads the dot for two very different meanings.

J/Link maps Java types to their natural equivalents in *Mathematica*. For example Java integer types (`char`, `int`, `long`, and so on) map to *Mathematica* integers. Arrays in Java map to *Mathematica* lists of the appropriate depth. Java objects map to **JavaObject** expressions. When I say “map,” I mean that if a Java method takes an `int`, then you call it from *Mathematica* with an integer argument. If a Java method returns an `int`, you will get an integer in *Mathematica* as the return value. Section 1.1.4 of the User Guide has a table showing the complete set of type mappings. It should correspond with most people's intuition about which types are most logically equivalent (for example, `boolean` maps to **True** or **False** in *Mathematica*, and the `null` object maps to *Mathematica*'s symbol **Null**).

Underscores in Java Names

Some Java names have underscores (`_`) in them. These characters are especially common in the names of constants, for example `TOP_ALIGNMENT`. This causes a problem for *J/Link* because the underscore is not a valid character in a *Mathematica* symbol name. *J/Link* solves this problem by mapping the underscore into a capital U in symbol names. The rule is that when you use a Java name in a *Mathematica* symbol, you convert the `_` to a U, but when you use the name in a *Mathematica* string, you do no conversion (in other words, you convert only when necessary). For example, if you have a class named `My_Class` with a field named `SOME_CONSTANT`, you would get the value of that field like this.

```
val = MyUClass`SOMEUCONSTANT
```

But when you refer to the class name in a string, you keep it unmodified.

```
LoadJavaClass["My_Class"]
```

□ Cleaning up Unused Objects

J/Link keeps track of which Java objects have been sent to *Mathematica* so that it can prevent the Java garbage collector from trying to free them while they are still in use by *Mathematica*. Unless you explicitly tell *J/Link* that you are finished with a Java object in *Mathematica*, it will never allow the object to be freed and objects will pile up in your Java session. This is generally not a problem if you are just tinkering with *J/Link*, but, in code that you write for others to use, you should make sure that you clean up any Java objects that are no longer needed.

You use the function **ReleaseJavaObject** to tell *J/Link* that you no longer wish to use a Java object in *Mathematica*. Here we release the **Date** object we created earlier.

```
In[7]:= ReleaseJavaObject[dateObject]
```

It is an error to refer to an object in *Mathematica* after it has been released.

In actual practice, **ReleaseJavaObject** is not used very often because it is too much effort to keep track of every object sent to *Mathematica*. What you usually want is a way to create a block of code in which all newly-created Java objects will be marked as temporary and automatically released when the block finishes. This is especially the case with user interface programs—your program will create a window with some controls in it and display it to the user, but, when the user closes the window, the program is done and does not need to leave anything behind. In the same way that any local variables that were used get cleaned up, you want any Java objects sent to *Mathematica* to be released. The function that accomplishes this is **JavaBlock**.

Many *J/Link* programs will have the following structure.

```
MyFunc[args_]:=
  JavaBlock[
    Module[{obj, result, otherLocals},
      obj = JavaNew["SomeClass"];
      ... work with obj, perhaps create more objects ...
      result
    ]
  ]
```

You can be guaranteed that **MyFunc** will not “leak” any objects, meaning that any objects sent to *Mathematica* for the first time during the execution of **MyFunc** will be released when it finishes. Note that only objects sent to *Mathematica* for the *first time* will be released. Objects that have already been seen by *Mathematica* will not be affected. This means that you do not have to worry that **JavaBlock** will aggressively release an object that is not truly temporary to that evaluation.

Remember that Java objects are sent to *Mathematica* not just when you call **JavaNew**, but also when you call a Java method that returns an object. You might not even have assigned the object to a variable, such as when you chain method calls.

```
obj@methodThatReturnsObject[]@otherMethod[]
```

In this example, `methodThatReturnsObject()` will cause a **JavaObject** expression to be created in *Mathematica* but not be assigned to any variable. This is why you should get in the habit of wrapping **JavaBlock** around code that does not need to create long-lived Java objects in *Mathematica*. You cannot tell whether a program creates objects in *Mathematica* simply by looking for calls to **JavaNew**.

There is one useful exception to the rule that all novel objects encountered during a **JavaBlock** are released when it ends. If a **JavaBlock** returns a single object, that object will not be released. This allows another common programming pattern in which you create several objects and operate on them, but you need just one of them to live beyond the boundary of the **JavaBlock**. We will see examples of this later, such as when you create a `Frame` object and populate it with controls (which are themselves objects), but you need the `Frame` object itself to persist after the function that created it finishes.

□ Getting Information about Classes

J/Link provides a few utility functions that let you see what constructors, methods, and fields a class includes. Although this is no substitute for documentation on the classes you are using, it can be very useful as a quick help system. Here we inspect the constructors for the `java.awt.Button` class.

```
In[8]:= Constructors["java.awt.Button"]

Out[8]//TableForm=
  Button() throws java.awt.HeadlessException
  Button(String) throws java.awt.HeadlessException
```

Here are the fields.

```
In[9]:= Fields["java.awt.Button"]

Out[9]//TableForm=
  static final int ABORT
  static final int ALLBITS
  static final float BOTTOM_ALIGNMENT
  static final float CENTER_ALIGNMENT
  static final int ERROR
  static final int FRAMEBITS
  static final int HEIGHT
  static final float LEFT_ALIGNMENT
  static final int PROPERTIES
  static final float RIGHT_ALIGNMENT
  static final int SOMEBITS
  static final float TOP_ALIGNMENT
  static final int WIDTH
```

There are a lot of methods in the `Button` class, the vast majority of which are inherited from parent classes (especially the very large class `java.awt.Component`). If you are only interested in the methods unique to the `Button` class, you can use the option **Inherited→False**.

```
In[10]:= Methods["java.awt.Button", Inherited → False]
```

```
Out[10]//TableForm=
```

```
void addActionListener(java.awt.event.ActionListener)
void addNotify()
javax.accessibility.AccessibleContext getAccessibleContext()
String getActionCommand()
String getLabel()
java.util.EventListener[] getListeners(Class)
void removeActionListener(java.awt.event.ActionListener)
void setActionCommand(String)
void setLabel(String)
```

You can also use string patterns to restrict the members you want to see. Here we specify that we only want to see the “set” methods (because we have left out the **Inherited**→**False** option, we see many methods defined in parent classes).

```
In[11]:= Methods["java.awt.Button", "set*"]
```

```
Out[11]//TableForm=
```

```
void setActionCommand(String)
void setBackground(java.awt.Color)
void setBounds(int, int, int, int)
void setBounds(java.awt.Rectangle)
void setComponentOrientation(java.awt.ComponentOrientation)
void setCursor(java.awt.Cursor)
void setDropTarget(java.awt.dnd.DropTarget)
void setEnabled(boolean)
void setFocusable(boolean)
void setFocusTraversalKeysEnabled(boolean)
void setFocusTraversalKeys(int, java.util.Set)
void setFont(java.awt.Font)
void setForeground(java.awt.Color)
void setIgnoreRepaint(boolean)
void setLabel(String)
void setLocale(java.util.Locale)
void setLocation(int, int)
void setLocation(java.awt.Point)
void setName(String)
void setSize(int, int)
void setSize(java.awt.Dimension)
void setVisible(boolean)
```

The examples above used the class name as the first argument, but you can also use an object of the class.

```
In[12]:= button = JavaNew["java.awt.Button"];
Fields[button, "*_ALIGNMENT"]

Out[13]//TableForm=
static final float BOTTOM_ALIGNMENT
static final float CENTER_ALIGNMENT
static final float LEFT_ALIGNMENT
static final float RIGHT_ALIGNMENT
static final float TOP_ALIGNMENT
```

■ User Interfaces with *J/Link*

After these basics of *J/Link*, we are ready to begin writing user interfaces. As we have seen, when we write *J/Link* programs we are writing the same sort of code that we would write in Java, except using *Mathematica* syntax. This means that the user interface programs we write will look almost identical to programs of similar functionality written in Java. This is both a great strength of *J/Link* and a drawback. As a drawback, it means that there is no higher-level “*Mathematica*-like” layer on top of the raw Java calls. Such a layer would attempt to make it simpler to create interfaces and reduce the amount of code that needed to be written. But the fact that the only programming functions you need to learn are the Java user interface classes has some important advantages. First, if you already have experience using the Java classes, that knowledge carries over directly. If you do not know the Java classes, then the knowledge you gain from using them with *J/Link* will have advantages in the future if you ever do Java programming. In other words, you do not have to invest time learning a special *Mathematica*-only user interface toolkit. Another important benefit is that you can take advantage of the huge number of existing books on Java programming and the large amount of sample code out there. Finally, you have exactly the amount of power that Java’s libraries give you—no less. If you use a simplified toolkit, then you risk running into limitations of the toolkit if you have special needs or want to do something very fancy. In the future, Wolfram Research may very well make available another way of creating user interfaces that sits on top of *J/Link* and hides some of the raw Java calls (but still allows power users to get down to the metal if necessary). For now, though, writing user interface programs in *Mathematica* is essentially just like writing them in Java.

As mentioned earlier, an obvious consequence of the fact that you will be making calls to the Java user interface class libraries is that you will want documentation for these libraries. You can go a long way just by copying fragments out of the *J/Link* example programs below and the ones included with *J/Link*, but there is no substitute for a reference book. Good reference materials are online at Sun’s Java pages. The JavaDocs for all the standard Java classes can be found at java.sun.com/j2se/1.4/docs/api, and there are good tutorials on Java user interfaces at java.sun.com/docs/books/tutorial/ui, if you want to use the older but simpler Abstract Window Toolkit (AWT) classes, and at java.sun.com/docs/books/tutorial/uiswing/index.html, if you want to use the newer and more powerful Swing library.

As just mentioned, there are two user interface class libraries that are a standard part of Java. The original one is called AWT, and the newer one is called Swing. The AWT is easy to use but suffers from some limitations, most notably that it can be hard to get user interfaces to look very polished on every platform. It also is not suitable for very complicated interfaces, but that would not be a limitation for most of the interfaces *Mathematica* programmers are likely to create. In response to perceived problems with AWT, Sun created a new user interface library called Swing. Swing is a little more complicated to use but has more features and gives you a better chance of having your interface look good on every platform. We will use Swing for the examples in this article, primarily because that is what most programmers are using nowadays. The examples included with J/Link use mostly the older AWT because when they were created, some platforms did not have Java runtimes modern enough to run Swing.

□ Your First Window

Let us get started right away with our first window. The Swing class that represents a top-level window is called `JFrame`. Instead of using `JFrame`, we will use a subclass of it that is included in J/Link, called `MathJFrame`. `MathJFrame` adds two features to a standard `JFrame` that make it useful for J/Link programmers: the window goes away when its Close box is clicked, and it has special support for so-called “modal” J/Link interfaces, which will be discussed later.

```
In[14]:= myWindow = JavaNew["com.wolfram.jlink.MathJFrame"]
```

```
Out[14]= «JavaObject[com.wolfram.jlink.MathJFrame] »
```

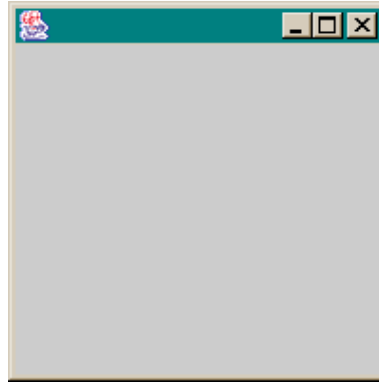
You may notice a slight delay the first time a class is loaded into *Mathematica*, especially if it (or one of its parent classes) has many methods, as is the case with the `JFrame` class. We have now created a window, but it is not visible. Before we make it visible, let us make it a reasonable size and place it somewhere other than the default location at the top left corner of the screen.

```
In[15]:= myWindow@setSize[200, 200]
myWindow@setLocation[300, 300]
```

The `JFrame` class has methods called `setVisible()` and `toFront()` that sound like they would be useful for making the window appear on top of all other windows. Unfortunately, these methods are not enough to force the window to the foreground on all platforms, so J/Link includes a special function called **JavaShow** that does this for you.

```
In[17]:= JavaShow[myWindow]
```

Here is what the window looks like.



Congratulations, you have now created your first *Mathematica* user interface! For a little fun, play around with resizing and repositioning it with *Mathematica*.

```
In[18]:= myWindow@setSize[400, 400]
```

You have just experienced one of the incredible advantages of *J/Link* programming versus writing programs in Java. In *Mathematica* you can build your interface a line at a time, *while it is running*. There is no need to write a complete program, compile it, and run it just to see what it does. This ability to interactively experiment with Java classes and methods makes *J/Link* an ideal tool for learning about a class library.

□ Adding a Component to the Window

An empty window is not the most useful interface, so let us add something to it: a button that changes the background color of the window every time it is clicked. The Swing class for buttons is `JButton`, and one of the constructors lets us specify the text in the button.

```
In[19]:= myButton = JavaNew["javax.swing.JButton", "Click Me!"]
```

```
Out[19]= «JavaObject[javax.swing.JButton] »
```

When you add components to a Swing `JFrame`, you do not add them directly to the `JFrame`—instead, you add them to a special “pane” called the content pane. The Swing documentation goes into detail about the motivation for having the content pane, but you do not really need to care. Just remember to call `getContentPane()`.

```
In[20]:= myWindow@getContentPane[]@add[myButton]
```

```
Out[20]= «JavaObject[javax.swing.JButton] »
```

Components that can contain other components support a special class called a layout manager, which can supply complex logic for controlling the sizes and positions of child components. We will use a layout manager in a later example, but for now we will specify the button’s size and position explicitly. To do this we need to get rid of the content pane’s default layout manager.

```
In[21]:= myWindow@getContentPane[]@setLayout[Null]
```

Next we position the button.

```
In[22]:= myButton@setSize[100, 30]  
myButton@setLocation[50, 70]
```

□ Handling Events

Now that we have the window and its button, we need to add some behavior. We want a click in the button to cause the background color of the window to change to a random color generated by *Mathematica*. Java user interface components fire events in response to user actions, and other components indicate their interest in these events by registering as event listeners. In our example, the button will fire an event when it is clicked, and something else will listen for the event and change the content pane's color in response. The "something else" that listens for the button-click event is not the content pane itself. To put that logic into the content pane would require creating a subclass of the pane's class. It would be very cumbersome to have to create a new subclass of each component class just to allow it to respond to events in a certain way. Instead, the programmer writes an "adapter" class that implements the desired event-listener interface and calls certain methods in the component in response to various events. The only specialty code goes into the adapter class, allowing the components that fire and respond to events to be generic.

Swing buttons fire an `ActionEvent` when clicked, and the corresponding event-listener interface is `ActionListener`. In our example, a Java programmer would create a tiny new class that implements the `ActionListener` interface. This interface has only one method, `actionPerformed()`, and in the body of the `actionPerformed()` method the programmer would set the content pane's color. The programmer would wire up this event behavior by calling the button's `addActionListener()` method, supplying an instance of this new class. When the button is clicked, it notifies all its registered listeners by calling their `actionPerformed()` methods.

We see from this discussion that Java programmers typically need to create a new listener class for every behavior they want to add to their program. *Mathematica* programmers using *J/Link* cannot write new Java classes, so how do we add behaviors to our programs? The answer is that *J/Link* comes with ready-made listener classes for the user interface events typically encountered in Java programs. These classes are referred to generically as "MathListener" classes because they listen for events and respond by calling user-specified code in *Mathematica*. To add a behavior to your program, you simply create an instance of the appropriate MathListener class and tell it what *Mathematica* function you want executed for each of the events that it is capable of responding to.

This is all very straightforward when you see it in action. Let us add the click behavior to our button. First we create an instance of the appropriate MathListener class, which for a button is `MathActionListener`.

```
In[24]:= actionListener = JavaNew["com.wolfram.jlink.MathActionListener"]
```

```
Out[24]= «JavaObject[com.wolfram.jlink.MathActionListener] »
```

Now we assign the *Mathematica* function we want called. To do this, we use the `setHandler()` method, which takes the name of the event-listener method (“`actionPerformed`”) and the name (as a string) of a *Mathematica* function to be called to implement the `actionPerformed()` method. Our *Mathematica* function will be called **ButtonClickHandler**.

```
In[25]:= actionListener@setHandler["actionPerformed", "ButtonClickHandler"]
```

```
Out[25]= True
```

Now we need to register our **actionListener** with the button. This is done by calling the button’s `addActionListener()` method.

```
In[26]:= myButton@addActionListener[actionListener]
```

All that remains is to write the **ButtonClickHandler** function. To see what arguments will be passed to your *Mathematica* handler function, consult the table in Section 1.2.7.4 of the User Guide. Your function will always be passed the event object itself, and possibly other information that might be useful (this extra information can always be obtained from the event object—it is provided separately simply as a convenience). In this example, we are not interested in any of the arguments.

```
In[27]:= ButtonClickHandler[___] :=
  JavaBlock[
    myWindow@getContentPane[]@setBackground[
      JavaNew["java.awt.Color", Random[], Random[], Random[]]
    ]
  ]
```

Note that we wrapped the body of the function in **JavaBlock** to release the `Color` object that we created. We used the `Color` constructor that takes RGB values as real numbers in the range 0 to 1.

It has only taken a few lines of *Mathematica* code to wire up a simple behavior for the button. At this point you might be wondering how to know what events a component fires and what `MathListener` to use. Events and event listeners are well documented in Java, but in practice you can use a very quick shortcut. Simply look in the JavaDocs for the component you are interested in (`JButton`, `JPanel`, `JComboBox`, `JComponent`, and so on) for one or more methods with names of the form `addXXXListener`. Do not forget to look in the methods inherited from parent classes as well. Once you find such a method, you know that the component fires the events listed in the `XXXListener` interface. The corresponding `MathListener` will be named `MathXXXListener`.

In some cases you will find that there is no `MathListener` class provided for the event-listener interface that you need to use. For example, at the time of this writing there is no `MathMenuListener` in *J/Link* for handling events from `JMenu` components. If there is no existing `MathListener`, you can use the *Mathematica*

function **ImplementJavaInterface** to create an appropriate class “on the fly.” We will see an example of using **ImplementJavaInterface** later, and Section 1.2.17 of the User Guide describes **ImplementJavaInterface** in detail and gives an example of using it to handle events from a JMenu.

Did you click the button to see if it changed the color of the window? If you have not done this yet, do not, because it will not work! One crucial step remains, which is the subject of the next section.

□ Modal and Modeless Interfaces

Consider what will happen if you click on the button in your running example program. Java will detect the click, and the button will notify the *MathActionListener* by calling its `actionPerformed()` method. The *MathActionListener* will send commands to *Mathematica* to execute the **ButtonClickHandler** function. These commands will arrive on the *MathLink* that connects the *Mathematica* kernel and Java. What will *Mathematica* do when these commands arrive on the link? Absolutely nothing! That is because the *Mathematica* kernel is not paying any attention to the link to Java. The kernel spends its life waiting for commands to arrive on the *MathLink* that connects to the front end. Before any Java user interface program can interact with *Mathematica*, *Mathematica* needs to be put into a state where it is receptive to commands arriving from Java.

There are two main ways for making *Mathematica* receptive to commands initiated in Java (a third, less commonly used method is described in Section 1.2.7.11 of the User Guide). The method you choose will depend on the requirements of your user interface program. Consider a Java dialog box that is used to control some initial values for a computation. The dialog box is created and displayed; the user enters some values and then dismisses it by clicking the OK button. The entire lifetime of the dialog box falls within the bounds of a single computation in *Mathematica*. In contrast, consider a Java interface that is intended to remain visible and active for a long stretch of time in a user’s session. This could be some kind of tool or utility that the user would use occasionally, similar to a palette in the front end.

The first type of interface, which is created and destroyed within the span of a single computation, is called a “modal” interface. The term “modal” is used here with a slightly different meaning than the usual one. In standard computing terminology, a modal window is one that must be dismissed before anything else can be done in the application. A typical example is a Save File dialog box. In *J/Link*, what we call a modal interface is not necessarily modal with respect to other Java windows—it is modal with respect to the *Mathematica* kernel. That is, the kernel is kept busy dealing with commands arriving from Java until the Java window is dismissed.

The second type of interface, which remains active for longer than a single kernel computation, is called “modeless.” It is modeless with respect to the *Mathematica* kernel, meaning that the kernel is not kept busy servicing commands from Java. The user is free to do computations using the front end while the Java interface is still active.

Modal interfaces are more common than modeless ones, and they are conceptually simpler. They also offer better performance if the Java program needs to call *Mathematica* many times in quick succession, such as when a slider is being dragged. Modal interfaces are run using the **DoModal** function, which does not return until the modal interface is dismissed. Modeless interfaces are enabled with the **ShareKernel** function, which puts the kernel into a state where it is receptive to commands arriving from either the front end or Java.

Let us now run our button example program in a modal way. Before we do this, go back and click the Click Me! button if you have not done so already. From the discussion in the previous few paragraphs we know it will not work yet, but it is interesting to see exactly what happens. Not only does the color of the window not change, but the Java user interface thread hangs. You can see this by clicking the button again and noting that it does not even give any visual feedback of being pressed. In fact, if you bring another window in front of the Java window and then click the Java window to bring it back in front, it will not repaint itself, a sure sign that the Java user interface thread is hanging. This makes sense because the thread is waiting for *Mathematica* to respond to its attempt to call the **ButtonClickHandler** function.

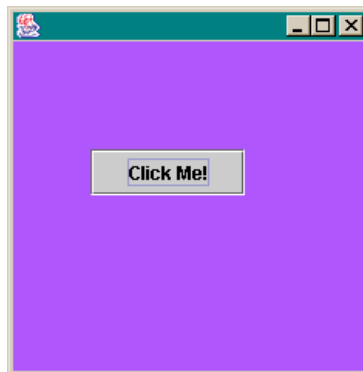
The user interface thread will hang until we tell *Mathematica* to start paying attention to commands arriving from Java, which we can do by calling **DoModal[]** (or **ShareKernel[]** if we want a modeless interface). The **DoModal** function will not return until the Java side calls **EndModal[]**. There are many ways to arrange for this to happen, but by far the easiest and most common way is to use a special facility of the *MathJFrame* class. If you call a *MathJFrame*'s **setModal()** method, it will call **EndModal[]** when it is closed, thus causing **DoModal** to return.

```
In[28]:= myWindow@setModal[]
```

Now we are finally ready to run the button example as an interactive program.

```
In[29]:= DoModal[]
```

The call to **DoModal** will not return until you close the Java window. Click the Click Me! button a few times and see how the window color changes. Here is what it looks like in action.



□ The Complete ButtonClick Program

We have now completed a trivial *J/Link* user interface program. Note that it only took a few lines of code, and all of it is written in *Mathematica*. The code is almost line-for-line comparable to a pure Java version of the same program. Now let us collect all the code together and package it as a simple program called **ButtonClick**.

```
In[30]:= ButtonClick[] :=
  JavaBlock[
    Module[{window, button, actionListener, buttonClickHandler},
      InstallJava[];
      window = JavaNew["com.wolfram.jlink.MathJFrame"];
      window@setSize[400, 400];
      window@setLocation[300, 300];
      button = JavaNew["javax.swing.JButton", "Click Me!"];
      window@getContentPane[]@add[button];
      window@getContentPane[]@setLayout[Null];
      button@setSize[100, 30];
      button@setLocation[50, 70];
      actionListener =
        JavaNew["com.wolfram.jlink.MathActionListener"];
      actionListener@setHandler["actionPerformed",
        ToString[buttonClickHandler]];
      button@addActionListener[actionListener];
      buttonClickHandler[___] :=
        JavaBlock[
          window@getContentPane[]@setBackground[
            JavaNew["java.awt.Color",
              Random[], Random[], Random[]
            ]
          ]
        ];
      window@setModal[];
      JavaShow[window];
      DoModal[]
    ]
  ]
```

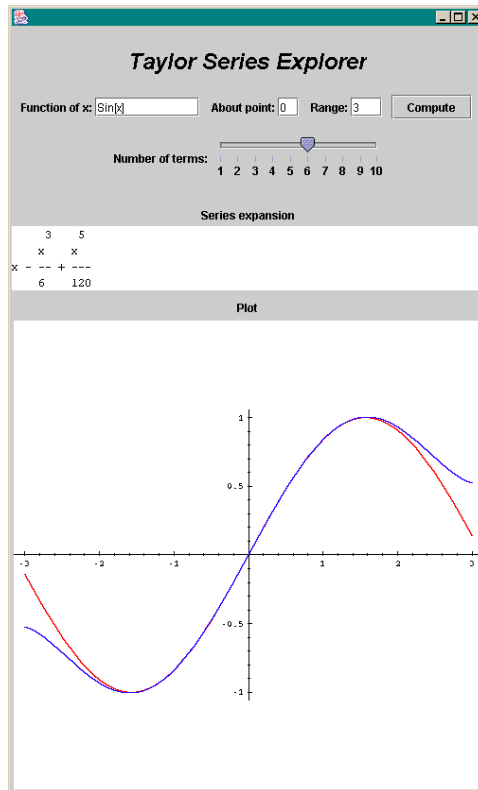
The **ButtonClick** function is a very typical *J/Link* program. The whole function is wrapped in **JavaBlock**, as we have no need for any of the Java objects created to persist in *Mathematica* after the function ends. Another common idiom is to call **InstallJava** at the start, in case Java is not already running. One trick to note is in the line that calls **setHandler()**. We want the **buttonClickHandler** function to be local to the **Module**, and this means that its true name is something like **buttonClickHandler\$123**. We have to pass the name of this function as a string in **setHandler()**, and we can be sure to capture its true name by calling **ToString[buttonClickHandler]**, instead of incorrectly hard-coding **"buttonClickHandler"**.

□ A More Advanced Example

We have now worked our way through a trivial *J/Link* user interface program. Now let us look at something considerably more advanced, and examine some new techniques along the way. In particular, we will see the following.

- How to display *Mathematica* graphics in a Java window.
- How to use **ShareKernel** to run an interface modelessly.
- How to use **ImplementJavaInterface** to allow event callbacks to *Mathematica* when there is no appropriate existing *MathListener* class.

The program we will create allows the user to experiment with Taylor series approximations to functions. They can enter a function and a range over which to plot the function and its Taylor series approximation. They then use a slider to modify the number of terms in the series expansion. The series expansion is displayed along with a plot showing the function and the Taylor series approximation. The plot and series expansion are updated on the fly as the slider is dragged. Here is a screen shot of the program in action.



The general flow of the **Taylor** program is as follows.

```
Taylor[] :=  
  JavaBlock[  
    Module[{...},  
      ... prepare the window and its components ...  
      ... establish the behaviors via callbacks to Mathematica ...  
      ... show the window and run it via DoModal ...  
    ]  
  ]
```

Laying out the Window

As a first step in creating the **Taylor** program, we will write a preliminary version, **Taylor1**, that only builds the interface and displays it. Behavior will be added in subsequent revisions. The code to create the window and all its components is somewhat lengthy. One reason for this is that we will do the work necessary to ensure that the components in the window lay out nicely on all platforms and reposition correctly if the window is resized. In the **ButtonClick** example, we specified the button's size and location with explicit numerical values. This works fine for simple windows that cannot be resized and have just a few components, or if you are not concerned about platforms beyond the one on which you are developing the program. But if you hardcode component positions and sizes for a more complex window, it is practically guaranteed that on some platforms (perhaps Mac OS X, or Linux with some unusual window manager and fonts) there will be layout problems. Typical examples are buttons that are too small to hold all their text, labels that are cut off by other components, and text fields that are the wrong height for a line of text.

The best way to avoid layout problems like these is to use a layout manager, which is a special Java class that knows how to lay out child components within a container component. Proper use of a layout manager helps ensure that components have their “natural” sizes on each platform and reposition sensibly if their container is resized. Using layout managers can be a tricky part of Java programming, but it is worth the effort if you want to create high-quality interfaces that look good on every machine. In the **Taylor** program, we will use a **BoxLayout**, which is a new layout manager provided in Java 1.3. **BoxLayout** is useful when you have several rows with different numbers of components in each. A useful tutorial on layout managers can be found at java.sun.com/docs/books/tutorial/uiswing/mini/layout.html.

The section of the program that creates the window and lays out the components is too long to try to work through and explain. Frankly, this part of any program is somewhat tedious and uninteresting, and it is beyond the scope of this article to try to teach readers how to write this type of code. The *Mathematica* code you write using *J/Link* is essentially identical to the code a Java programmer would write, and for guidance you can refer to any of the copious books, articles, and other resources on user interface programming in Java.

The code in **Taylor1** is presented as a complete function, but of course this is not how it was created. Naturally, it was built up bit by bit with much experimentation. This interactive programming style is the huge advantage of using *J/Link*

versus programming directly in Java. It is especially valuable when trying to get the layout of components just right. In fact, I had never used a `BoxLayout` and its helper `Box` class before, and being able to play with them a line at a time was extremely useful for learning.

Two things that are temporarily left out of **Taylor1** are a **JavaBlock** wrapping the entire function and local variables specified in the **Module**. It is very important that these are left out because this **Taylor1** program is not self-contained. What we want for now is a program that simply displays the window with its final appearance. We will then interactively add behavior via event callbacks to *Mathematica*. To do this we need access to the various Java objects we have created (the frame, the plot panel, the text fields, and so on), hence they cannot yet be made local to the **Module** or released by a **JavaBlock**.

```
In[31]:= Taylor1[] := Module[{}, (* No local declarations yet. *)
(* Create the window and position it. *)
  window = JavaNew["com.wolfram.jlink.MathJFrame"];
  window@setLocation[50, 50];
  window@setTitle["Taylor"];

(* Create the two panels we will use. topPanel holds all except the plot. *)
  topPanel = JavaNew["javax.swing.JPanel"];
  topPanel@setLayout[JavaNew["javax.swing.BoxLayout", topPanel,
BoxLayout`YUAXIS]];
  plotPanel = JavaNew["com.wolfram.jlink.MathGraphicsJPanel"];
  LoadJavaClass["java.awt.Color"];
  plotPanel@setBackground[Color`white];
  plotPanel@setPreferredSize[JavaNew["java.awt.Dimension", 500, 500]];

(* Add the two panels to the top and bottom of the window. *)
  LoadJavaClass["java.awt.BorderLayout"];
  window@getContentPane[]@add[topPanel,
ReturnAsJavaObject[BorderLayout`NORTH]];
  window@getContentPane[]@add[plotPanel,
ReturnAsJavaObject[BorderLayout`SOUTH]];

(* Now create and configure the various components. There are two main rows:
row1 has all the text fields and Compute button, and row2 has the slider. Use
Box, Border and Glue components to ensure a nice layout on all platforms and at
varying window sizes.
*)
  LoadJavaClass["javax.swing.Box"];
  row1 = Box`createHorizontalBox[];
  row2 = Box`createHorizontalBox[];
  functionField = JavaNew["javax.swing.JTextField", "Sin[x]", 10];
  functionField@setMaximumSize[JavaNew["java.awt.Dimension", 20, 20]];
  aboutPointField = JavaNew["javax.swing.JTextField", "0", 2];
  aboutPointField@setMaximumSize[JavaNew["java.awt.Dimension", 8, 20]];
  aboutPointLabel = JavaNew["javax.swing.JLabel", "About point: "];
  functionLabel = JavaNew["javax.swing.JLabel", "Function of x: "];
  rangeLabel = JavaNew["javax.swing.JLabel", "Range: "];
  rangeField = JavaNew["javax.swing.JTextField", "3", 3];
```

```

rangeField@setMaximumSize[JavaNew["java.awt.Dimension", 8, 20]];
computeButton = JavaNew["javax.swing.JButton", "Compute"];
row1@add[Box`createHorizontalGlue[]];
row1@add[functionLabel]; row1@add[functionField];
row1@add[Box`createHorizontalGlue[]];
row1@add[aboutPointLabel]; row1@add[aboutPointField];
row1@add[Box`createHorizontalGlue[]];
row1@add[rangeLabel]; row1@add[rangeField];
row1@add[Box`createHorizontalGlue[]];
row1@add[computeButton];
row1@add[Box`createHorizontalGlue[]];
title = JavaNew["javax.swing.JLabel", "Taylor Series Explorer"];
title@setFont[JavaNew["java.awt.Font", "Dialog",
    BitOr[Font`BOLD, Font`ITALIC], 24]];
title@setAlignmentX[Component`CENTERUALIGNMENT];
LoadJavaClass["javax.swing.BorderFactory"];
title@setBorder[BorderFactory`createEmptyBorder[20, 0, 20, 0]];
slider = JavaNew["javax.swing.JSlider", 1, 10, 1];
slider@setSnapToTicks[True]; slider@setPaintLabels[True];
slider@setPaintTicks[True]; slider@setMajorTickSpacing[1];
slider@setMaximumSize[JavaNew["java.awt.Dimension", 180, 50]];
slider@setBorder[BorderFactory`createEmptyBorder[20, 0, 20, 0]];
row2@add[JavaNew["javax.swing.JLabel", " Number of terms: "]];
row2@add[slider];
seriesLabel = JavaNew["javax.swing.JLabel", "Series expansion"];
seriesLabel@setAlignmentX[Component`CENTERUALIGNMENT];
seriesLabel@setBorder[BorderFactory`createEmptyBorder[10, 0, 4, 0]];
seriesTextArea = JavaNew["javax.swing.JTextArea"];
seriesTextArea@setRows[4];
seriesTextArea@setFont[
    JavaNew["java.awt.Font", "Monospaced", Font`PLAIN, 12]];
plotLabel = JavaNew["javax.swing.JLabel", "Plot"];
plotLabel@setAlignmentX[Component`CENTERUALIGNMENT];
plotLabel@setBorder[BorderFactory`createEmptyBorder[10, 0, 4, 0]];

(* Add everything to the top panel. *)
topPanel@add[title]; topPanel@add[row1]; topPanel@add[row2];
topPanel@add[seriesLabel]; topPanel@add[seriesTextArea];
topPanel@add[plotLabel];
window@pack[];

JavaShow[window]
]

```

Yikes, that is a lot of code, but for the most part it is straightforward: create the components, set some properties, and put them where they belong. Scattered throughout are calls to manage properties of the layout, such as borders and spacing. This will pay off in allowing the window to be resized and still keep a nice appearance. One *J/Link* programming issue is worth noting in this code—the line that reads:

```

window@getContentPanel[]@add[topPanel,
ReturnAsJavaObject[BorderLayout`NORTH]];

```

You might be wondering why this is not the more obvious code:

```
window@getContentPane[]@add[topPanel, BorderLayout`NORTH];
```

The answer is that the `add()` method takes an `Object`, not a `String`, as its second argument. The constant `BorderLayout.NORTH` is a Java string, and so it arrives in *Mathematica* as a literal string. The problem is that *J/Link* does not allow you to pass a *Mathematica* string to an argument slot that is typed to take a Java object. What you need to pass as the second argument to `add()` is a **JavaObject**: `ject` that is a reference to a Java string with the appropriate value. An easy way to do this is to use the **ReturnAsJavaObject** function to ask *J/Link* to return `BorderLayout.NORTH` to *Mathematica* as a **JavaObject** instead of its usual behavior of converting it into a *Mathematica* string.

Adding Behavior

Now that we have a function that builds the interface, let us start it running so that we can experiment with adding behavior.

```
In[32]:= Taylor1[]
```

The **Taylor1** program shows the window and then returns, leaving the window on the screen. Now we want to add behavior—specifically, we want the two computed components (the series expansion text field and the plot panel) to recompute when either the slider is dragged or the Compute button is clicked. As we have seen, we will do this by registering event listeners that call back to *Mathematica* when the user manipulates either of these two controls.

The first thing we need is a function that extracts the necessary values from the input fields in the window, computes the necessary results, and updates the series expansion text box and the plot panel. We will call this function **computeTaylor**, and here is what it looks like.

```
In[33]:= computeTaylor[___] :=
Module[{func, about, range, sliderVal, series, plot},
  func = ToExpression[functionField@getText[]];
  about = ToExpression[aboutPointField@getText[]];
  range = ToExpression[rangeField@getText[]];
  sliderVal = slider@getValue[];
  series = Collect[Normal[Series[func, {x, about, sliderVal}]], x];
  seriesTextArea@setText[ToString[series, OutputForm]];
  Block[{$DisplayFunction = Identity},
    plot = Plot[{func, series}, {x, about - range, about + range},
      PlotStyle -> {Hue[0], Hue[0.7]}]
  ];
  plotPanel@setMathCommand[ToString[plot, InputForm]]
]
```

We have made **computeTaylor** accept (and ignore) any sequence of arguments because it will eventually be called from Java in two different circumstances with different arguments, none of which are we interested in. The first four lines of **computeTaylor** extract the values from the input fields and slider. The function then computes the series expansion, puts the **OutputForm** of this result in the

series expansion text area, and creates the plot. Because we are only interested in the **Graphics** object returned by **Plot**, we use a **Block** to control the value of **\$DisplayFunction** so that no PostScript output is created.

The last line of **computeTaylor** causes the plot to appear in the plot panel. This panel is an instance of a special class provided with *J/Link* called **MathGraphics**: **JPanel**, which is a subclass of the Swing **JPanel** class with some special capabilities for displaying *Mathematica* graphics and typeset expressions. The **MathGraphicsJPanel** class and its counterpart for AWT-style programs, **MathCanvas**, are discussed in more detail in Section 1.2.8 of the User Guide. For many programs, using either of these classes is as simple as calling the **setMathCommand()** method, which takes a string giving a *Mathematica* expression that evaluates to a graphics expression. The panel evaluates the expression and displays the resulting image, automatically centered and scaled to fit the panel's current dimensions.

We can now call **computeTaylor** and see its effects in the window. Make sure the input fields in the window have reasonable values and then execute this line.

```
In[34]:= computeTaylor[]
```

The only thing left is to wire up the button and slider so that they call the **computeTaylor** function. Before we can allow calls to *Mathematica* from the Java user interface thread, we need to ensure that *Mathematica* is in a state where it is receptive to calls originating in Java. In the **ButtonClick** example, we achieved this by using **DoModal** to put the kernel into a loop where it was continuously reading from the link to Java. We will use **DoModal** again with the **Taylor** program, but only when we have fully packaged it in its final form. It is inconvenient to use **DoModal** during development because we want to be able to tinker with the program while it is running. The **DoModal** function does not return until the window is closed, so we would need to set up all the behavior in advance. You can use the **ShareKernel** function to put the kernel into a state where it is receptive to calls from either the front end or Java. This is convenient during development even for programs, like **Taylor**, that will be run modally when deployed.

This starts sharing.

```
In[35]:= sharingToken = ShareKernel[];
```

We save the result from **ShareKernel** to pass to **UnshareKernel** later on, when we turn off the sharing state.

When a **JButton** is clicked, it notifies all registered **ActionListeners** by calling their **actionPerformed()** methods. Therefore the **MathListener** we need to use is **MathActionListener**.

```
(sharing) In[36]:= buttonListener = JavaNew["com.wolfram.jlink.MathActionListener"];
buttonListener@setHandler["actionPerformed", "computeTaylor"];
computeButton@addActionListener[buttonListener];
```

The **computeTaylor** function will now be called when the button is clicked. It will be passed some arguments when it is called, but we are not interested in any

of them. Try altering the input fields in the window and clicking the Compute button to see the program in action.

The Compute button is intended for when the user has edited the text fields and wants to recompute. We also want the slider to be “live,” so that dragging it automatically recomputes. To see what events a JSlider fires as it is dragged, we examine the JavaDocs for JSlider looking for methods of the form `addXXXListener()`. We find `addChangeListener()` and see that we need a class that implements the `javax.swing.event.ChangeListener` interface. Unfortunately, *J/Link* does not include a `MathChangeListener` class, so what do we do?

J/Link provides a function called **ImplementJavaInterface** that allows you to create a new Java class “on the fly” that implements any interface by calling into *Mathematica*. When you call **ImplementJavaInterface**, you specify a list of rules that associates each method in the interface with a *Mathematica* function to be called that provides the implementation of that method. The most common use of **ImplementJavaInterface** is to create `MathListener`-type classes for interfaces that *J/Link* does not have built-in `MathListener` classes for. Below we create the listener we need, specifying that the `stateChanged()` method of the `ChangeListener` interface should be implemented by calling the `computeTaylor` function.

```
(sharing) In[39]:= sliderListener =  
    ImplementJavaInterface["javax.swing.event.ChangeListener",  
        {"stateChanged" -> "computeTaylor"}];
```

Now register our new listener with the slider.

```
(sharing) In[40]:= slider@addChangeListener[sliderListener]
```

Play around with the slider and other fields to see the interface in action. When you are finished with the Taylor window, close it by clicking in its Close box. We have now interactively created the code we need for the final version of **Taylor**, and are satisfied that it looks and works as desired. We were able to create the code in stages, interactively modifying the window and its behavior while it was running.

The final version of the **Taylor** program will be run modally, and because we are done with the interactive development phase, we call **UnshareKernel** to turn off kernel sharing. We pass the value that was returned from **ShareKernel** as the argument to **UnshareKernel**.

```
(sharing) In[41]:= UnshareKernel[sharingToken];
```

The Completed Program

Here is the complete code for the final **Taylor** program. We essentially just collect all the code we have created so far into a single function.

```
In[42]:= Taylor[f_:Sin[x], aboutPoint_:0, range_:3, nTerms_Integer:3] :=  
    JavaBlock[  
        Module[{window, topPanel, plotPanel, row1, row2, functionField,  
            aboutPointField, aboutPointLabel, functionLabel,  
            rangeLabel, rangeField, computeButton, title, slider,
```

```

        seriesLabel, seriesTextArea, plotLabel, computeTaylor,
        buttonListener, sliderListener},
InstallJava[];

(* Create the window and position it. *)
window = JavaNew["com.wolfram.jlink.MathJFrame"];
window@setLocation[50, 50];
window@setTitle["Taylor"];

(* Create the two panels we will use. topPanel holds all except the plot.
*)
topPanel = JavaNew["javax.swing.JPanel"];
topPanel@setLayout[JavaNew["javax.swing.BoxLayout", topPanel,
BoxLayout`YUAXIS]];
plotPanel = JavaNew["com.wolfram.jlink.MathGraphicsJPanel"];
LoadJavaClass["java.awt.Color"];
plotPanel@setBackground[Color`white];
plotPanel@setPreferredSize[JavaNew["java.awt.Dimension", 500, 500]];

(* Add the two panels to the top and bottom of the window. *)
LoadJavaClass["java.awt.BorderLayout"];
window@getContentPane[]@add[topPanel,
ReturnAsJavaObject[BorderLayout`NORTH]];
window@getContentPane[]@add[plotPanel,
ReturnAsJavaObject[BorderLayout`SOUTH]];

(* Now create and configure the various components. There are
two main rows: row1 has all the text fields and Compute button,
and row2 has the slider. Use Box, Border and Glue components to
ensure a nice layout on all platforms and at varying window sizes.
*)
LoadJavaClass["javax.swing.Box"];
row1 = Box`createHorizontalBox[];
row2 = Box`createHorizontalBox[];
functionField = JavaNew["javax.swing.JTextField", ToString[f], 10];
functionField@setMaximumSize[JavaNew["java.awt.Dimension", 20, 20]];
aboutPointField = JavaNew["javax.swing.JTextField",
ToString[aboutPoint], 2];

aboutPointField@setMaximumSize[JavaNew["java.awt.Dimension", 8, 20]];
aboutPointLabel = JavaNew["javax.swing.JLabel", " About point: "];
functionLabel = JavaNew["javax.swing.JLabel", "Function of x: "];
rangeLabel = JavaNew["javax.swing.JLabel", " Range: "];
rangeField = JavaNew["javax.swing.JTextField", ToString[range], 3];
rangeField@setMaximumSize[JavaNew["java.awt.Dimension", 8, 20]];
computeButton = JavaNew["javax.swing.JButton", "Compute"];
row1@add[Box`createHorizontalGlue[]];
row1@add[functionLabel]; row1@add[functionField];
row1@add[Box`createHorizontalGlue[]];
row1@add[aboutPointLabel]; row1@add[aboutPointField];
row1@add[Box`createHorizontalGlue[]];
row1@add[rangeLabel]; row1@add[rangeField];
row1@add[Box`createHorizontalGlue[]];

```

```

row1@add[computeButton];
row1@add[Box`createHorizontalGlue[]];
title = JavaNew["javax.swing.JLabel", "Taylor Series Explorer"];
title@setFont[JavaNew["java.awt.Font", "Dialog",
    BitOr[Font`BOLD, Font`ITALIC], 24]];
title@setAlignmentX[Component`CENTERALIGNMENT];
LoadJavaClass["javax.swing.BorderFactory"];
title@setBorder[BorderFactory`createEmptyBorder[20, 0, 20, 0]];
slider = JavaNew["javax.swing.JSlider", 1, 10, 1];
slider@setValue[nTerms];
slider@setSnapToTicks[True]; slider@setPaintLabels[True];
slider@setPaintTicks[True]; slider@setMajorTickSpacing[1];
slider@setMaximumSize[JavaNew["java.awt.Dimension", 180, 50]];
slider@setBorder[BorderFactory`createEmptyBorder[20, 0, 20, 0]];
row2@add[JavaNew["javax.swing.JLabel", " Number of terms: "]];
row2@add[slider];
seriesLabel = JavaNew["javax.swing.JLabel", "Series expansion"];
seriesLabel@setAlignmentX[Component`CENTERALIGNMENT];
seriesLabel@setBorder[BorderFactory`createEmptyBorder[10, 0, 4, 0]];
seriesTextArea = JavaNew["javax.swing.JTextArea"];
seriesTextArea@setRows[4];
seriesTextArea@setFont[
    JavaNew["java.awt.Font", "Monospaced", Font`PLAIN, 12]];
plotLabel = JavaNew["javax.swing.JLabel", "Plot"];
plotLabel@setAlignmentX[Component`CENTERALIGNMENT];
plotLabel@setBorder[BorderFactory`createEmptyBorder[10, 0, 4, 0]];

(* Add everything to the top panel. *)
topPanel@add[title]; topPanel@add[row1]; topPanel@add[row2];
topPanel@add[seriesLabel]; topPanel@add[seriesTextArea];
topPanel@add[plotLabel];
window@pack[];

(* Define the function to be called when the button is clicked
or slider dragged.
*)
computeTaylor[___] :=
Module[{func, about, rng, sliderVal, series, plot},
    func = ToExpression[functionField@getText[]];
    about = ToExpression[aboutPointField@getText[]];
    rng = ToExpression[rangeField@getText[]];
    sliderVal = slider@getValue[];
    series = Collect[Normal[Series[func, {x, about, sliderVal}]], x];
    seriesTextArea@setText[ToString[series, OutputForm]];
    Block[{$DisplayFunction = Identity},
        plot = Plot[{func, series}, {x, about-rng, about+rng},
            PlotStyle -> {Hue[0], Hue[0.7]}]
    ];
    plotPanel@setMathCommand[ToString[plot, InputForm]]
];

(* Add behavior. *)
buttonListener = JavaNew["com.wolfram.jlink.MathActionListener"];

```

```

        buttonListener@setHandler["actionPerformed",
ToString[computeTaylor]];
computeButton@addActionListener[buttonListener];
sliderListener =
    ImplementJavaInterface["javax.swing.event.ChangeListener",
        {"stateChanged" -> ToString[computeTaylor]}};
slider@addChangeListener[sliderListener];

window@setModal[];
JavaShow[window];
(* Call computeTaylor to get the plot shown for the initial values. *)
computeTaylor[];
DoModal[]
] ]

```

Now run this final **Taylor** program. Quit the program by clicking the window's Close box.

```
In[43]:= Taylor[]
```

You can also specify the initial function, point, range, and number of terms.

```
In[44]:= Taylor[Cos[x], 0, 3, 8]
```

■ Where to Go from Here

This article has been a quick introduction to the subject of creating user interfaces for *Mathematica* programs using *J/Link*. There are many techniques and tips that were not covered. For more information, consult the *J/Link* User Guide. In particular, look at Section 1.2.7, “Creating Windows and Other User Interface Elements.” There are also several examples provided with *J/Link*. They can be found in the <Mathematica directory>/AddOns/JLink/Examples/.Part1 directory, and most are discussed in detail in Section 1.3 of the User Guide. Get yourself one of the many books on the Java user interface APIs, put a link to the Java Platform JavaDocs in your browser favorites, and get programming!

About the Author

Todd Gayley is the creator of *J/Link*. He is currently working to bring the same capabilities to users of Microsoft's .NET framework.

Todd Gayley

Director of Java Technology
Wolfram Research, Inc.
tgayley@wolfram.com

Mathematica 4.2: A Technical Review

Lars Hohmuth

This article reviews the major new features and enhancements in *Mathematica* 4.2. These include improved connectivity to Java[™], XML, and the web, new packages for statistical analysis and technical publishing, a new slide show environment for presentations, as well as many additions and improvements to the kernel functions.

■ Introduction

Mathematica 4.2 introduced many new features including enhanced connectivity to Java[™], Extensible Markup Language (XML), and the web. Programmers can now take advantage of the included Java Runtime Environment (JRE) to create Java-based user interfaces or make use of standard Java libraries in *Mathematica*. The new XML capabilities allow users to import and manipulate XML data and to export notebooks or cells in XML format.

There were a large number of enhancements to both the front end and kernel. The new *AuthorTools* package and the new slide show environment make it easy to create great-looking technical documents and presentations. In addition, there were many new functions for linear programming, statistics, optimization, combinatorics, and graph theory. *Mathematica* 4.2 also provides easier management and restriction of licenses than was possible with previous versions.

The list below summarizes the major improvements. The rest of this article discusses some of these features in more detail.

- Transparent Java integration with *J/Link* 2.0 and built-in JRE
- New XML functions for symbolic XML manipulation
- Support for exporting *Mathematica* notebooks and expressions as XML
- Support for Extensible Hypertext Markup Language (XHTML) export including style sheets
- Extended MathML 2.0 support
- Improved linear programming and optimization
- Speed and robustness improvements for many numerical functions
- Improved simplifier

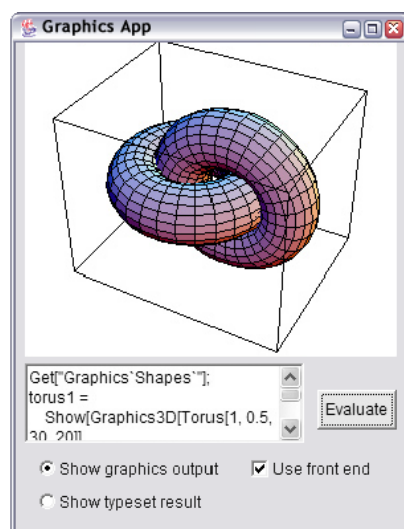
- New *AuthorTools* package for technical publishing
- New packages for statistical analysis of variance (ANOVA) and global optimization
- Enhanced *Combinatorica* package for combinatorics and graph theory
- New import and export formats including Flexible Image Transport System (FITS) and Spatial Data Transfer Standard (SDTS)
- A new tutorial giving a ten-minute overview of *Mathematica* for new users
- Slide show environment for presentations
- Enhanced management and monitoring of licenses for multiple copies on a network

■ J/Link 2.0

Mathematica 4.2 shipped with a new release of *J/Link*, the toolkit that integrates *Mathematica* and has Java pre-installed, thereby avoiding the installation step for all platforms. Using *J/Link*, you can call Java from *Mathematica* in a completely transparent way and also control the *Mathematica* kernel from any Java program.

For *Mathematica* users, *J/Link* makes the whole universe of existing and future Java classes an automatic extension to the *Mathematica* environment. For example, users can use standard Java classes to provide new interfaces to *Mathematica* or create graphical user interface (GUI) elements using Abstract Window Toolkit (AWT) or Swing classes.

For Java programmers, *J/Link* turns *Mathematica* into a scripting shell that allows them to experiment with, build, and test Java classes one line at a time. It also makes Java a powerful language for writing programs that use the computational services of *Mathematica*.



Mathematica 4.2 includes *J/Link* and JRE Version 1.4 on Windows, Linux, and Solaris and the JRE Version 1.3.1 on most other platforms. This means that you can start using *J/Link* right out of the box. *Mathematica* developers can now also rely on a JRE for applications that mix *Mathematica* and Java functionality. Let us look at a simple example.

This loads the *J/Link* package.

```
In[1]:= << JLink`
```

This launches the Java runtime and prepares it for use.

```
In[2]:= InstallJava[]
```

```
Out[2]= LinkObject[C:\Program, 2, 2]
```

This constructs a new object of the specified class.

```
In[3]:= newrandom = JavaNew["java.util.Random"]
```

```
Out[3]= «JavaObject[java.util.Random] »
```

Functions in this Java class can then be called from *Mathematica*.

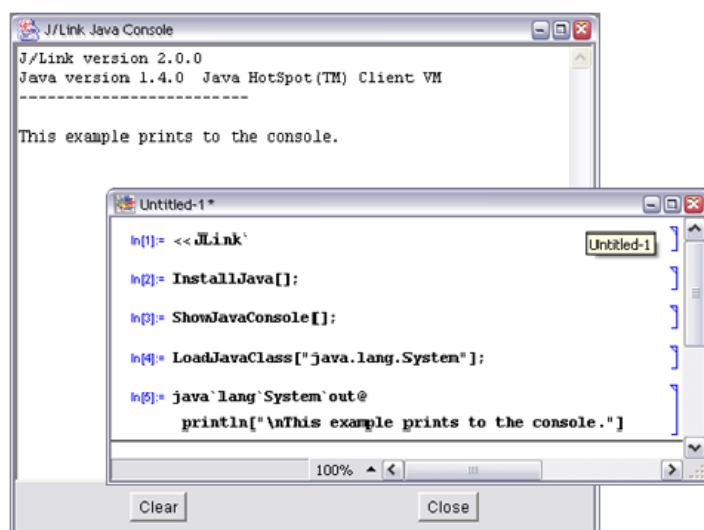
```
In[4]:= {newrandom@nextInt[], newrandom@nextLong[],  
        newrandom@nextFloat[], newrandom@nextDouble[]}
```

```
Out[4]= {-1886226219, 4989899970133314885, 0.620182, 0.796959}
```

J/Link 2.0 featured a number of enhancements over previous versions of *J/Link*, including the following.

- *JLink.jar* was moved to the *JLink* directory, from *JLink/SystemAdditions*.
- There were dramatic speed improvements for calling Java methods.
- A new component called `com.wolfram.jlink.ui.MathSessionPane` implements a complete In/Out session window, but with many more features.
- A new component called `MathGraphicsJPanel` displays *Mathematica* graphics and typeset expressions.
- The `JavaBlock` function is improved to give it more control over what objects are released when it ends. These improvements make some common programming idioms easier, without having to resort to manual use of `ReleaseObject`.
- A new `KeepObjects` option added to `JavaBlock` allows you to specify a list of objects that will not be released at the end.
- The new `ImplementInterface` function lets you implement any Java interface entirely with *Mathematica* code.
- A new `SetInternetProxy` function allows users to specify proxy port and host information if necessary to reach the internet.

- A new `MakeJavaExpr` function lets you construct, from *Mathematica*, a Java `expr` object from an arbitrary *Mathematica* expression.
- A new `JLink`Information`` context defines symbols giving version information.
- For Java programmers, the `MLGetType` and `MLGetNext` methods, when applied to a `KernelLink` object, can now return `KernelLink.MLTKOBJ`JECT`. This means that the expression on the link is a Java object reference and can be read with `getObject`.
- The *MathLink* `connect` method takes an optional `timeout` parameter giving the number of milliseconds to wait for the connection before throwing an exception.
- Information can be sent to the Java Console.



See the article by Todd Gayley in this issue for information on building user interfaces using *J/Link*. Additional examples and information about *J/Link* are available on the Wolfram Research website at www.wolfram.com/solutions/mathlink/jlink.

■ XML Support

Mathematica 4.2 introduced powerful new functionality for importing, manipulating, and exporting XML. You can import any XML document and then use *Mathematica*'s built-in functions to analyze the contents of the document or transform its structure. You can also export notebooks in XML format using the NotebookML DTD defined for this purpose. *Mathematica* 4.2 also includes support

for specific XML formats such as MathML, SVG, and XHTML for exporting mathematical formulas, graphics, and complete notebooks, respectively.

□ Manipulating XML Using SymbolicXML

When you import an XML document, it is automatically converted into Symbolic XML, a special type of *Mathematica* expression that preserves the structure of the original XML document. Since both XML and *Mathematica* expressions have a tree structure, it is easy to define a mapping between them. The advantage of converting an XML document to SymbolicXML is that you can then use all the standard techniques of *Mathematica* programming, including functional programming and pattern matching to extract or analyze specific pieces of data from the XML document. Let us look at some examples.

Example 1

This imports an XML file containing data on baseball players. You can download this file from the TMJ website at www.mathematica-journal.com/issue/v9i1.

```
In[5]:= mydata = Import["ALHitters.xml"];
```

With some knowledge of how the particular data set is structured, you can construct parsing functions to extract specific pieces of data from the file. This example extracts the entries corresponding to players for the New York Yankees.

```
In[6]:= yankees = Cases[mydata, XMLElement["PlayerRecord", _,
    {___, XMLElement["TEAM", _, {"NYY"}], ___}], Infinity];
```

The data set contains the names of 21 players who are hitters for the Yankees.

```
In[7]:= Length[yankees]
```

```
Out[7]= 21
```

The following function can be applied to the data in order to obtain a list of player names that fit the criteria specified earlier.

```
In[8]:= GetName[playerrecord_] := First[
    Cases[playerrecord, XMLElement["PLAYER", _, {name_}] → name, 2]]
```

```
In[9]:= Map[GetName, yankees]
```

```
Out[9]= {Bernie Williams, Derek Jeter, David Justice, Jorge Posada, Paul Oneill,
    Tino Martinez, Chuck Knoblauch, Jose Canseco, Scott Brosius,
    Luis Polonia, Shane Spencer, Glenallen Hill, Clay Bellinger,
    Jose Vizcaino, Luis Sojo, Chris Turner, Ryan Thompson,
    Alfonso Soriano, Lance Johnson, Felix Jose, Roberto Kelly}
```

Example 2

You can fetch XML data over the web using the `XML`Parser`XMLGet` function. Here is a function that extracts data from the Amazon.com website.

```
In[10]:= AmazonSearch[str_String] :=
    XML`Parser`XMLGet[
        "http://rcm.amazon.com/e/cm?t=subcommittee&l=st1&search=" <>
        str <> "&mode=books&p=102&o=1&f=xm1",
        "ValidateAgainstDTD" → False];
```

This searches for all books listed on Amazon.com that contain the word “mathematica” in the title.

```
In[11]:= books = AmazonSearch["mathematica"];
```

The following command sorts the search results in order of their Amazon sales rank and displays them in a table.

```
In[12]:= Cases[books,
  XMLElement["product", __,
    {__, XMLElement["ranking", __, {rank_}],
     __, XMLElement["title", __, {title_}],
     __, XMLElement["author", __, {author_}],
     __, XMLElement["release_date", __, {releasedate_}], ___}] ->
  {StringTake[title, 20], author, rank, releasedate},
  Infinity] // TableForm
```

```
Out[12]//TableForm=
```

On Formally Undecida	Godel, Kurt	1	19920200
The Mathematica Book	Wolfram, Stephen	2	19990315
Mathematical Statist	Rose, Colin	3	20020308
Mathematica for Micr	Stinespring, John Robert	4	20011215
Practical Optimizati	Bhatti, M. Asghar	5	20000700
Mastering Mathematic	Gray, John W.	6	19971000
Schaum's Outline of	Don, Eugene	7	20000930
Modeling Nature : Ce	Gaylord, Richard J.	8	19960700
Mathematica Graphics	Wickham-Jones, Tom	9	19941200
Fantasia Mathematica	Fadiman, Clifton	10	19970400
Option Valuation und	Lewis, Alan L.	11	20000201
Modeling Financial D	Shaw, William T.	12	19990200
Calculus & Mathemati	Bill Davis	13	19990101
Principia Mathematic	Whitehead, Alfred North	14	19970600
Mathematica Navigato	Ruskeepaa, Heikki	15	19990115

□ Exporting Notebooks as XML

You can save any notebook in XML format by choosing the File ▸ Save As ▸ XML (NotebookML) menu command. NotebookML is an XML format developed by Wolfram Research for representing notebooks. Here is a simple notebook expression.

```
In[13]:= nbexp = Notebook[{Cell["First Section", "Section"],
  Cell["Here is some text", "Text"]}];
```

This exports the notebook expression as NotebookML. Notice the correspondence between the different parts of the notebook expression and the element names in the NotebookML representation.

```
In[14]:= ExportString[nbexp, "NotebookML"]
```

```
Out[14]= <?xml version='1.0'?>
<!DOCTYPE Notebook SYSTEM '
  http://www.wolfram.com/XML/notebookml1.dtd'>
```

```

<Notebook CreatedBy='Mathematica 4.2'
  xmlns:xhtml='http://www.w3.org/1999/xhtml'
  xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
<List>
  <Cell class='Section'>
    <String>First Section</String>
    <Style>
      <String>Section</String>
    </Style>
  </Cell>
  <Cell class='Text'>
    <String>Here is some text</String>
    <Style>
      <String>Text</String>
    </Style>
  </Cell>
</List>
</Notebook>

```

When you import the NotebookML file back into *Mathematica*, it is automatically converted into a notebook.

```
In[15]:= ImportString[%, "NotebookML"]
```

```
Out[15]= Notebook[{Cell[First Section, Section], Cell[Here is some text, Text]}]
```

You can also export specific parts of a notebook in XML format, using another XML format called ExpressionML. As you would expect, ExpressionML is a subset of NotebookML. Here is the ExpressionML for a simple mathematical formula.

```
In[16]:= ExportString[x2 + Sin[x], "ExpressionML"]
```

```

Out[16]= <?xml version='1.0'?>
<!DOCTYPE Expression SYSTEM '
  http://www.wolfram.com/XML/notebookml1.dtd'>
<Expression xmlns:mathematica='http://www.wolfram.com/XML/'
  xmlns='http://www.wolfram.com/XML/'>
  <Function>
    <Symbol>Plus</Symbol>
    <Function>
      <Symbol>Power</Symbol>
      <Symbol>x</Symbol>
      <Number>2</Number>
    </Function>
    <Function>
      <Symbol>Sin</Symbol>
      <Symbol>x</Symbol>
    </Function>
  </Function>
</Expression>

```

□ Working with MathML

Mathematica 4.2 included comprehensive support for the MathML 2.0 standard for describing mathematics in machine-to-machine communication. Endorsed and recommended by the World Wide Web Consortium (W3C) and developed with strong support from Wolfram Research, MathML makes it possible to embed mathematical expressions in web pages and XML documents.

You can export a notebook as an XHTML document with all formulas in the notebook automatically converted into MathML by choosing File ▸ Save As ▸ HTML+MathML. The resulting document can then be displayed in any browser that supports MathML, such as Mozilla 1.0, Netscape 7.0, or Internet Explorer 5.5 (using the freely distributed MathPlayer plug-in).

You can also select a formula and choose Edit ▸ Copy As ▸ MathML to copy its MathML representation to the Clipboard. Conversely, if you paste a MathML string into a notebook, it can be automatically interpreted and converted into a *Mathematica* expression. This makes it possible, for example, to copy a mathematical formula from a web page, paste it into a notebook, and evaluate it as standard *Mathematica* input.

There are also several kernel functions for converting between MathML and the boxes and expressions used by *Mathematica* to represent mathematical formulas. For example, `XML`MathML`ExpressionToMathML` yields the MathML representation of its argument.

```
In[17]:= XML`MathML`ExpressionToMathML[x2 + y]
```

```
Out[17]= <math xmlns='http://www.w3.org/1998/Math/MathML'>
  <semantics>
    <mrow>
      <msup>
        <mi>x</mi>
        <mn>2</mn>
      </msup>
      <mo>+</mo>
      <mi>y</mi>
    </mrow>
    <annotation-xml encoding='MathML-Content'>
      <apply>
        <plus/>
        <apply>
          <power/>
          <ci>x</ci>
          <cn type='integer'>2</cn>
        </apply>
        <ci>y</ci>
      </apply>
    </annotation-xml>
  </semantics>
</math>
```

There are several conversion options for customizing the output. For example, by default, the above function produces parallel markup containing both content and presentation MathML. You can get only presentation markup by explicitly specifying the "Formats" conversion option.

```
In[18]:= XML`MathML`ExpressionToMathML[
      x2 + y, "Formats" -> "PresentationMathML"]

Out[18]= <math xmlns='http://www.w3.org/1998/Math/MathML'>
      <mrow>
        <msup>
          <mi>x</mi>
          <mn>2</mn>
        </msup>
        <mo>+</mo>
        <mi>y</mi>
      </mrow>
    </math>
```

You can use the function `XML`MathML`MathMLToExpression` to do the reverse transformation, that is, from MathML to a *Mathematica* expression.

```
In[19]:= XML`MathML`MathMLToExpression[%]

Out[19]= x2 + y
```

■ New and Enhanced Packages

□ Global Optimization

Mathematica 4.2 introduced two new functions, **NMinimize** and **NMaximize**, for solving global optimization problems. These functions are flexible enough to handle functions that are not differentiable or continuous. You can specify one of four different methods for doing the optimization:

- differential evolution (genetic programming)
- Nelder–Mead (nonlinear simplex algorithm)
- simulated annealing
- random search

This loads the package.

```
In[20]:= << NumericalMath`NMinimize`
```

This minimizes a quadratic function over a polyhedron.

```
In[21]:= NMinimize[{x2 + (y - .5)2, y ≥ 0 && y ≤ x + 1}, {{x, 0, 10}, {y, 0, 10}}]

Out[21]= {0.125, {x → -0.249991, y → 0.750009}}
```

This is the corresponding exact minimization.

```
In[22]:= << Experimental'; Minimize[x^2 + (y - 1/2)^2, y ≥ 0 && y ≥ x + 1, {x, y}]
```

```
Out[22]:= {1/8, {x → -1/4, y → 3/4}}
```

Below are four examples that demonstrate how **NMinimize** handles some problems better than previous versions of *Mathematica*.

FindMinimum Example

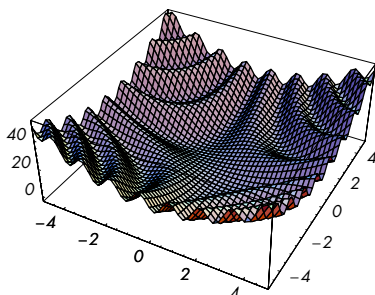
This defines a function with a lot of local minima.

```
In[23]:= Clear[f]; f[x_, y_] := 5 Sin[x*y] + x^2 + y^2
```

Here is a plot of the function.

```
In[24]:= Plot3D[f[x, y], {x, -5, 5}, {y, -5, 5}, PlotPoints → 50];
```

From In[24]:=



This function cannot be easily handled using **FindMinimum**. If we start **FindMinimum** away from the global minimum, it gets trapped by a local minimum.

```
In[25]:= FindMinimum[f[x, y], {x, 5}, {y, 5}]
```

```
Out[25]:= {41.7183, {x → 4.81149, y → 4.81149}}
```

However, **NMinimize** finds the global minimum, with similar starting points.

```
In[26]:= NMinimize[f[x, y], {{x, 5, 6}, {y, 5, 6}}]
```

```
Out[26]:= {-2.26402, {x → 1.0767, y → -1.0767}}
```

Constrained Minimization Example

NMinimize can solve problems including constraints in the case $x + y \geq 1$.

```
In[27]:= NMinimize[{x^2 + y^2, x + y ≥ 1}, {x, y}]
```

```
Out[27]:= {0.5, {x → 0.500021, y → 0.499979}}
```

Solve Example

Mathematica's standard **Solve** cannot find a solution for this system of equations.

```
In[28]:= Solve[Floor[x^2] == 196 && Mod[x, 5] == 4, {x}]
```

InverseFunction::ifun :
Inverse functions are being used. Values may be lost for multivalued inverses.

Solve::ifun :
Inverse functions are being used by Solve, so some solutions may not be found.

```
Out[28]= {{x -> -sqrt(Floor[196]), {x -> sqrt(Floor[196])}}
```

NMinimize can solve this problem easily if given a constant objective function and the equations to be solved as constraints.

```
In[29]:= NMinimize[{1, Floor[x^2] == 196 && Mod[x, 5] == 4}, {x}]
```

```
Out[29]= {1., {x -> 14.}}
```

NonlinearFit Example

In this example, we generate a data set with random perturbations.

```
In[30]:= Clear[a, b, c, d, e];
SeedRandom[13];
model = a + Sin[b 2 Pi t + c] + Sin[d 2 Pi t + e];
parameters = {a, b, c, d, e};
values = Table[Random[], {5}];
data = Table[Evaluate[{t, model /. Thread[parameters -> values]}] +
  Random[Real, {-0.1, 0.1}], {t, 0, 2 Pi, .05 * 2 Pi}]

Out[35]= {{-0.0118779, 0.693023}, {0.253306, 1.76439}, {0.610906, 0.829856},
{1.00905, 0.190497}, {1.28726, 1.26628}, {1.6569, 1.86055},
{1.98229, 0.338248}, {2.17786, -0.882349}, {2.54513, 0.0269258},
{2.92582, 0.469951}, {3.16257, -1.07563}, {3.523, -1.78343},
{3.76718, -0.529739}, {4.13474, 0.357163}, {4.43687, -0.629511},
{4.72606, -0.891516}, {5.10437, 0.846205}, {5.34844, 1.68502},
{5.68204, 0.636237}, {5.96208, 0.169645}, {6.20064, 1.38103}}
```

We first use **NonlinearFit** to try to fit a model to this data set.

```
In[36]:= << Statistics`NonlinearFit`;
```

```
In[37]:= nlf = NonlinearFit[data, model, {t}, {#, 0, 1} & /@ parameters]
```

FindMinimum::fmlim : The minimum could not be bracketed in 30 iterations.

```
Out[37]= 0.169377 - Sin[0.373292 - 2.64238 t] + Sin[1.19996 + 4.08387 t]
```

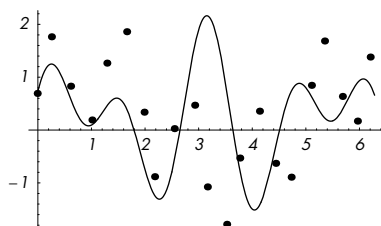
As we see below, **NonlinearFit** gets trapped by a local minimum.

```

In[38]:= lp = ListPlot[data, PlotJoined → False,
    DisplayFunction → Identity, PlotStyle → {PointSize[0.02]}];
nlfplot = Plot[nlf, {t, 0, 2 Pi}, DisplayFunction → Identity];
Show[lp, nlfplot, DisplayFunction → $DisplayFunction];

```

From In[38]:=



We can get a more accurate model by creating a sum of squares by hand and using **NMinimize** to find the minimum with the **DifferentialEvolution** method.

```

In[41]:= sos =
    Sum[(model /. {t → data[[i, 1]]} - data[[i, 2]]^2, {i, Length[data]}];
nmin = NMinimize[sos, {#, 0, 1} & /@ parameters,
    Method -> "DifferentialEvolution"]

Out[42]:= {0.550636,
    {a → 0.201929, b → 0.191519, c → 0.387394, d → 0.802503, e → 0.0205679}}

```

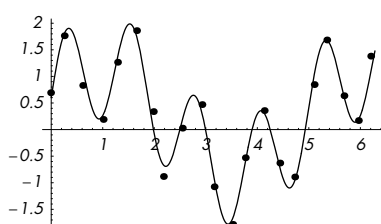
This approach creates a much better fit.

```

In[43]:= nminplot =
    Plot[model /. Last[nmin], {t, 0, 2 Pi}, DisplayFunction → Identity];
Show[lp, nminplot, DisplayFunction → $DisplayFunction];

```

From In[43]:=



□ Statistical Analysis

ANOVA is a statistical test for heterogeneity of means by analysis of group variances. The **Statistics'ANOVA'** package extends *Mathematica's* already rich statistical functionality to include support for analysis of variance. ANOVA was developed as a way of optimizing crop yields, but is widely used for quality assurance and biomedical experiments and in many other fields.

This loads the package.

```

In[45]:= << Statistics'ANOVA'

```

This creates a simple univariate data set.

```
In[46]:= data = {{1, 7.0}, {1, 5.3}, {1, 5.9}, {1, 6.6}, {1, 4.9}, {2, 4.4},
               {2, 6.8}, {2, 7.7}, {2, 8.3}, {2, 6.6}, {3, 8.1}, {3, 10.4}, {3, 8.0},
               {3, 6.8}, {3, 9.2}, {4, 5.7}, {4, 3.9}, {4, 6.2}, {4, 5.5}, {4, 6.2}};
```

This computes the ANOVA table for the above data.

```
In[47]:= ANOVA[data]

Out[47]= {ANOVA →
          DF      SumOfSq      MeanSq      FRatio      PValue
Model      3      26.2935      8.7645      6.12795      0.0056161
Error      16      22.884      1.43025
Total      19      49.1775

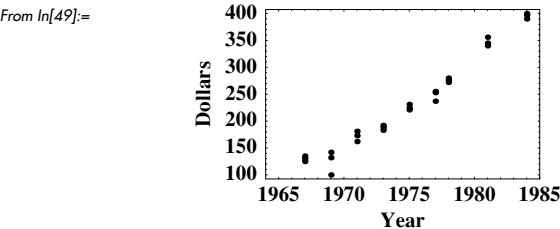
          All      6.675
          Model[1]  5.94
CellMeans → Model[2]  6.76 }
          Model[3]  8.5
          Model[4]  5.5
```

Here is another example data set.

```
In[48]:= mydata = {{1967, 125}, {1967, 130}, {1967, 135}, {1969, 142},
                  {1969, 100}, {1969, 132}, {1971, 162}, {1971, 181},
                  {1971, 173}, {1973, 188}, {1973, 192}, {1973, 183}, {1975, 221},
                  {1975, 225}, {1975, 231}, {1977, 253}, {1977, 255}, {1977, 237},
                  {1978, 272}, {1978, 280}, {1978, 276}, {1981, 345}, {1981, 340},
                  {1981, 356}, {1984, 400}, {1984, 390}, {1984, 397}};
```

Here is a plot for visualizing the data.

```
In[49]:= ListPlot[mydata, PlotStyle → {PointSize[.02]},
                  Frame → True, PlotRange → {{1964, 1985}, Automatic},
                  FrameLabel → {"Year", "Dollars"},
                  TextStyle → {FontFamily → "Times", FontWeight → "Bold"}];
```



Here is the ANOVA table for the data.

```
In[50]:= ANOVA[mydata]
```

```
Out[50]= {ANOVA →
```

	DF	SumOfSq	MeanSq	FRatio	PValue
Model	8	209113.	26139.2	276.876	0.
Error	18	1699.33	94.4074		
Total	26	210813.			

```

CellMeans →
      All      234.111
      Model[1] 130.
      Model[2] 124.667
      Model[3] 172.
      Model[4] 187.667
      Model[5] 225.667
      Model[6] 248.333
      Model[7] 276.
      Model[8] 347.
      Model[9] 395.667

```

□ Combinatorics

Mathematica 4.2 includes a completely new and expanded implementation of `DiscreteMath`Combinatorica``. Apart from new functions—covering topics such as Polya enumeration theory, used for enumerating isomers in chemistry—the package offers substantial speed and graphics enhancements. The new version of *Combinatorica* is described exhaustively in *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*, by Steven Skiena and Sriram Pemmaraju, to be published by Cambridge University Press. Let us look at some examples of the new functionality.

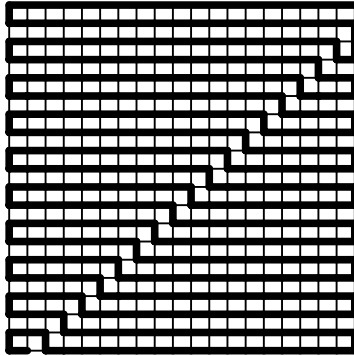
This loads the package.

```
In[51]:= << DiscreteMath`Combinatorica`
```

You can use the package to generate a variety of graphs that are of interest in graph theory. For example, `GridGraph[n,m]` constructs an $n \times m$ grid graph, the product of paths on n and m vertices.

```
In[52]:= s = GridGraph[20, 20];
ShowGraph[Highlight[s, {Map[Sort, DepthFirstTraversal[s, 2, Edge]]}],
VertexStyle -> Disk[0]];
```

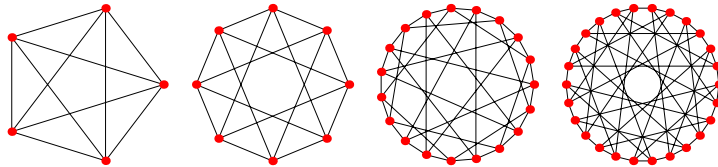
From In[52]:=



The girth of a graph is the length of its shortest cycle. A cage is the smallest possible regular graph that has a prescribed girth. Here, we plot several cage graphs of degree 4. **ShowGraphArray** permits the display of multiple graphs in one window.

```
In[54]:= ShowGraphArray[Table[CageGraph[4, n], {n, 3, 6}],
VertexStyle -> Disk[.05], VertexColor -> Red];
```

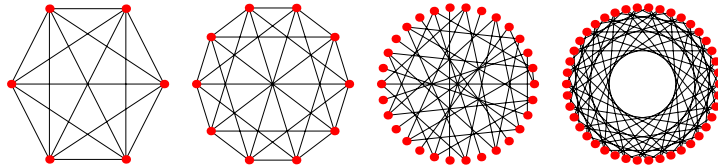
From In[54]:=



Here are some cage graphs of degree 5.

```
In[55]:= ShowGraphArray[Table[CageGraph[5, n], {n, 3, 6}],
VertexStyle -> Disk[.05], VertexColor -> Red];
```

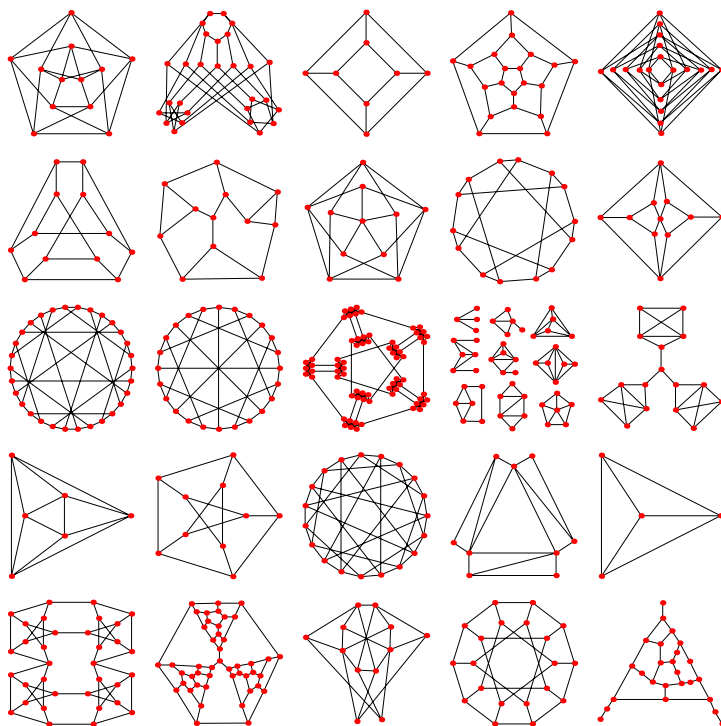
From In[55]:=



FiniteGraphs produces a list of all the interesting, finite, parameterless graphs built into *Combinatorica*.

```
In[56]:= ShowGraphArray[Partition[FiniteGraphs, 5],
  VertexStyle -> Disk[.04], VertexColor -> Red];
```

From In[56]:=



□ **AuthorTools**

The new *AuthorTools* package makes it easy to process notebooks for print or online publication. Using *AuthorTools* you can quickly create a table of contents or index from one or more notebooks. You can also set a variety of printing options, extract all cells of a specific type from a notebook, or create bilaterally formatted cells.

AuthorTools consists of about sixty functions and ten palettes. The palettes provide a point-and-click interface for most common tasks. You can access the various palettes using the OpenAuthorTools palette listed under the File ▸ Palettes menu.

For more advanced functions and greater customizability, you can also evaluate the package functions directly. For more details on using the package, see “*AuthorTools*: A Package for Document Processing,” in our next issue.

■ Kernel Enhancements

□ LinearProgramming

Mathematica 4.2 provides a completely new implementation of the **LinearProgramming** solver, bringing significant enhancements and increased performance. **LinearProgramming** also provides new functionality. Users can now directly specify equality constraints and bound constraints making typical problems easier to specify (e.g., as one equality instead of two inequalities).

Robustness

In *Mathematica* 4.2, a scaling technique is introduced so that **LinearProgramming** is able to solve very badly scaled problems. For example, consider the following Klee–Minty problem of dimension 5.

Maximize $10000 y[1] + 1000 y[2] + 100 y[3] + 10 y[4] + y[5]$
 subject to $y[1] \leq 1,$
 $20 y[1] + y[2] \leq 100,$
 $200 y[1] + 20 y[2] + y[3] \leq 100^2,$
 $2000 y[1] + 200 y[2] + 20 y[3] + y[4] \leq 100^3,$
 $20000 y[1] + 2000 y[2] + 200 y[3] + 20 y[4] + y[5] \leq 100^4,$
 $y[1] \geq 0, y[2] \geq 0, y[3] \geq 0, y[4] \geq 0, y[5] \geq 0$

In general, the Klee–Minty problem of dimension n , expressed in the **LinearProgramming** syntax, is:

```
In[57]:= KleeMinty[n_] :=
Module[{a, b, c, m, rhs}, c = Table[-10^(n - i), {i, n}];
a[i_] := Join[Table[2*10^(i - j), {j, i - 1}],
{1}, Table[0, {n - i}]];
rhs[i_] := {100^(i - 1), -1};
m = Table[a[i], {i, n}];
b = Table[rhs[i], {i, n}];
{c, m, b};
```

It is known that the use of the simplex algorithm with a naive pivot rule on this example would make the number of iterations increase exponentially with the dimension of the problem. However, due to the scaling technique introduced, this can be solved quite fast. For example, in *Mathematica* 4.1:

```
{c, m, b} = KleeMinty[16];
b = -Transpose[b][[1]];
m = -m;
{LinearProgramming[c, m, b], $Version} // Timing
{45.416 Second, {{0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 1000000000000000000000000000000},
4.1 for Microsoft Windows (September 19, 2000)}}
```

Mathematica 4.2 applies scaling by default. This means that the simplex algorithm converges very quickly.


```

In[64]:= s = NSolve[p == 0, {b, c, d, e}, WorkingPrecision -> 200]; // Timing
Out[64]= {2.724 Second, Null}

In[65]:= SolutionAccuracy /@ (p /. s)
Out[65]= {136, 136, 143, 143, 139, 137, 146, 148, 148, 143}

```

The solution accuracy described here is the worst residual error in p . **NSolve** gives a set of solutions that, when put back into the original equations, should all give zero. **SolutionAccuracy** checks for the value that is farthest from zero. This results in the number of digits to the right of the decimal point that are zero.

The corresponding list of accuracies in Version 4.1 is {54, 54, 50, 52, 71, 71, 64, 65, 70, 70}.

This is a 6×6 system of polynomial equations.

```

In[66]:= e =
{
  x0^2 + x1^2 + x2^2 - 1, x0*x3 + x1*x4 + x2*x5, x3^2 + x4^2 + x5^2 - 0.3,
  x3^2 + x4^2 - 2*x2*x4 + x2^2 + x5^2 + 2*x1*x5 + x1^2 - 1/4,
  x3^2 + Sqrt[3]*x2*x3 + 3/4*x2^2 + x4^2 - x2*x4 +
  1/4*x2^2 + x5^2 - Sqrt[3]*x0*x5 + x1*x5 + 3/4*x0^2 -
  Sqrt[3]/2*x0*x1 + 1/4*x1^2 - 1/4, x3^2 - 2/3*Sqrt[6]*x1*x3 +
  Sqrt[3]/3*x2*x3 + 2/3*x1^2 - Sqrt[18]/9*x1*x2 +
  1/12*x2^2 + x4^2 + 2/3*Sqrt[6]*x0*x4 - x2*x4 + 2/3*x0^2 -
  Sqrt[6]/3*x0*x2 + 1/4*x2^2 + x5^2 - Sqrt[3]/3*x0*x5 +
  x1*x5 + 1/12*x0^2 - Sqrt[3]/6*x0*x1 + 1/4*x1^2 - 1/4};

In[67]:= s = NSolve[e, {x0, x1, x2, x3, x4, x5}]; // Timing
Out[67]= {2.433 Second, Null}

```

All of the above solutions are equal to 0, at least up to the given accuracy, as shown below. The following code generates the residual errors for the previous calculation.

```

In[68]:= Chop[e /. s]
Out[68]= {{0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0},
{0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0}}

```

More precisely, this is the accuracy with which the polynomials are satisfied for each solution.

```

In[69]:= Map[-Log[10, Max[Abs[#]]] &, e /. s]
Out[69]= {15.0515, 13.7552, 13.4231, 13.7241, 14.1476, 13.6428, 14.5074, 13.6102,
13.7319, 14.3136, 13.5413, 13.6827, 14.3923, 14.1613, 11.7836, 12.3149,
12.6615, 12.6079, 13.9661, 14.1383, 13.1714, 13.4868, 14.0795, 13.6816}

```

□ Simplify

New Simplifier: ProductLog

Mathematica 4.2 allows you to simplify the output of the **ProductLog** function. This capability is useful when solving certain classes of exponential polynomial equations.

```
In[70]:= ProductLog[528309 / 168976 Log[9 / 4] 3^(7451 / 10561) 2^(3110 / 10561)] //
FullSimplify
```

```
Out[70]= 
$$\frac{39134 \operatorname{Log}\left[\frac{3}{2}\right]}{10561}$$

```

```
In[71]:= FullSimplify[-((2 * ProductLog[-(Log[2] / 2)]) / Log[2])]

```

```
Out[71]= 2
```

```
In[72]:= FullSimplify[-((2 * ProductLog[-1, -(Log[2] / 2)]) / Log[2])]

```

```
Out[72]= 4
```

```
In[73]:= Solve[2^x == x^2, x]
```

InverseFunction::ifun :
Inverse functions are being used. Values may be lost for multivalued inverses.

InverseFunction::ifun :
Inverse functions are being used. Values may be lost for multivalued inverses.

Solve::ifun :
Inverse functions are being used by Solve, so some solutions may not be found.

```
Out[73]= {{x -> - $\frac{2 \operatorname{ProductLog}\left[-\frac{\operatorname{Log}[2]}{2}\right]}{\operatorname{Log}[2]}\}},$ 

$$\{x \rightarrow -\frac{2 \operatorname{ProductLog}\left[\frac{\operatorname{Log}[2]}{2}\right]}{\operatorname{Log}[2]}\}, \{x \rightarrow -\frac{2 \operatorname{ProductLog}[-1, -\frac{\operatorname{Log}[2]}{2}]}{\operatorname{Log}[2]}\}}$$

```

```
In[74]:= FullSimplify[%]
```

```
Out[74]= {{x -> 2}, {x -> - $\frac{2 \operatorname{ProductLog}\left[\frac{\operatorname{Log}[2]}{2}\right]}{\operatorname{Log}[2]}\}}, \{x \rightarrow 4\}}$ 
```

```
In[75]:= (2 * ProductLog[Sqrt[2] * Log[8]]) / Log[2] // FullSimplify
```

```
Out[75]= 3
```

```
In[76]:= ProductLog[24 Log[2]] / Log[2] // FullSimplify
```

```
Out[76]= 3
```

Improved Simplifier: Mod

We can now simplify modular expressions, leading also to improvements for functions such as **Floor** and **Ceiling**.

```
In[77]:= FullSimplify[Mod[a + k Mod[b, m], m], Element[k, Integers]]
```

```
Out[77]= Mod[a + b k, m]
```

```
In[78]:= FullSimplify[Mod[2 Mod[2 x, 1], 1], 0 < x < 1/4]
```

```
Out[78]= 4 x
```

Improved Simplifier: Log

FullSimplify now knows when it can use the distribution rule in logarithmic expressions.

```
In[79]:= FullSimplify[Log[a x] == Log[a] + Log[x], -Pi < Arg[a] + Arg[x] ≤ Pi]
```

```
Out[79]= True
```

```
In[80]:= FullSimplify[Log[I x] == Log[I] + Log[x], -Pi/2 < Arg[x] < Pi/2]
```

```
Out[80]= True
```

Improved Simplifiers: Erf, InverseErf, Erfc, InverseErfc

Mathematica 4.2 also improves the capabilities for simplifying expressions involving the error function and related functions.

```
In[81]:= FullSimplify[Erf[InverseErf[x]], -1 < x < 1]
```

```
Out[81]= x
```

```
In[82]:= FullSimplify[InverseErfc[Erfc[x]], Element[x, Reals]]
```

```
Out[82]= x
```

□ Miscellaneous

Continuing the tradition of better performance and functionality with each new version, *Mathematica* 4.2 featured many other improvements to previously existing technology. The following are some examples of the additional improvements in the kernel.

Improved PackedArray Implementation

Mathematica 4.2 introduces improved packed array implementation over earlier versions. This improvement leads to speedups for functions such as **Dot** and **Transpose**. The speedups affect operations such as separating the colors in a two-dimensional image with three color planes.

FactorInteger

The speed of **FactorInteger** has been improved by a new implementation.

```
In[83]:= n = 2^128 + 1;
          {$Version, FactorInteger[n];} // Timing
```

```
Out[84]= {4.847 Second, {4.2 for Microsoft Windows (May 29, 2002), Null}}
```

For comparison, *Mathematica* 4.1 took almost 50 times as long to do the same computation.

```
{221.268 Second, {4.1 for Microsoft Windows (September 19, 2000), Null}}
```

The **FactorInteger** function now implements the quadratic sieve algorithm for large integers. This results in significant speed improvements for certain calculations. For example, the following command factors the 7th Fermat number, a notoriously large calculation that was done for the first time in 1970.

```
In[85]:= FactorInteger[227 + 1] // Timing
```

```
Out[85]:= {4.898 Second, {{59649589127497217, 1}, {5704689200685129054721, 1}}}
```

NotElement and InvisibleTimes

For simplicity, *Mathematica* 4.2 introduced a new Boolean test function that makes it easier to determine if an expression is not a member of a family of expressions or numbers.

```
In[86]:= NotElement[2^Sqrt[2], Algebraics]
```

```
Out[86]:= True
```

```
In[87]:= 123456789012345678901 ∈ Primes
```

```
Out[87]:= True
```

InvisibleTimes is also a new operator. The expression below is actually multiplying 2 and 3 together due to the presence of the **InvisibleTimes** operator between them.

```
In[88]:= 23
```

```
Out[88]:= 6
```

□ New Cellular Automata Function

Cellular automata are useful computational tools for describing and modeling natural systems. They have wide applications in science, engineering, and computer science. The new **CellularAutomaton**[*num*, *init*, *t*] function generates a list representing the evolution of cellular automaton rule *num* from initial condition *init* for *t* steps. This function makes it easy to experiment with cellular automata, such as the ones covered in Stephen Wolfram's *A New Kind of Science*. For more information on the book, visit www.wolframscience.com.

This generates the first 100 steps in the evolution of the rule 30 cellular automaton.

```
In[89]:= mydata = CellularAutomaton[30, {{1}, 0}, 100];
```

This generates a plot of the data so you can visualize the cellular automaton.

```
In[90]:= myGridPlot[data_, opts___] :=  
Module[{size}, size = Length[First[data]];  
Show[Graphics[Raster[1 - Reverse[data]], opts,  
AspectRatio → Automatic]]]; myGridPlot[mydata];
```

From In[90]:=

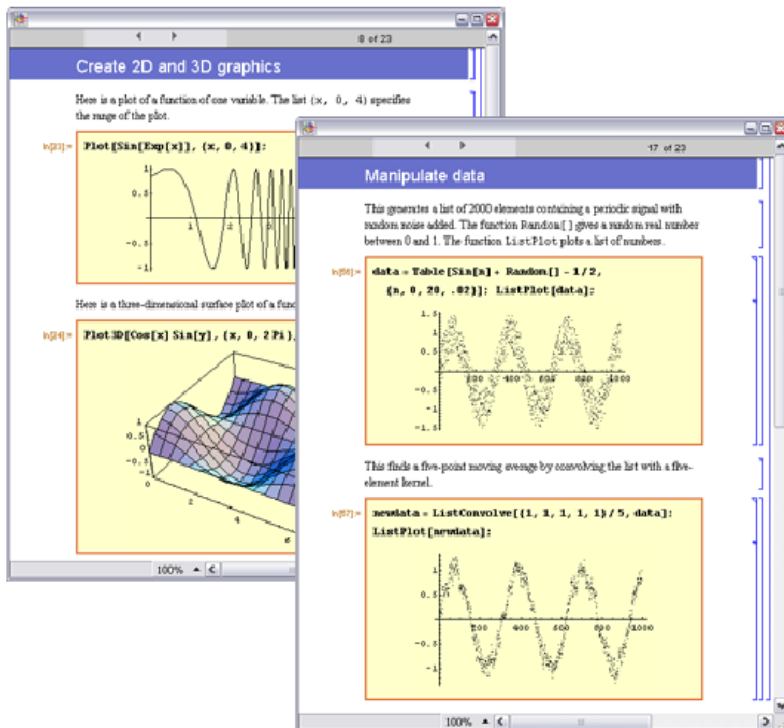


■ Front End Enhancements

□ Slide Show Environment and Ten-Minute Tutorial

A new screen style environment, introduced in *Mathematica* 4.2, allows notebooks to be presented in a slide show format. This new feature makes it simple to turn multiple notebooks into a single, easy-to-use presentation. It is no longer necessary to use third-party slide show viewers to present material that you have developed in *Mathematica*.

A good example of the possibilities of the new slide show environment is the ten-minute interactive tutorial that gives new users a gentle introduction to *Mathematica*. Because the tutorial is an ordinary notebook, new users can modify, change, and re-evaluate all *Mathematica* input directly in the tutorial.



□ **New *Mathematica* Fonts**

The *Mathematica* fonts now include 49 additional characters for representing planetary symbols, musical notation, and double-struck numbers.

Mercury	Venus	Earth	Mars	Jupiter	Saturn	Uranus	Neptune	Pluto
♿	♀	♁	♂	♃	♄	♅	♆	♇

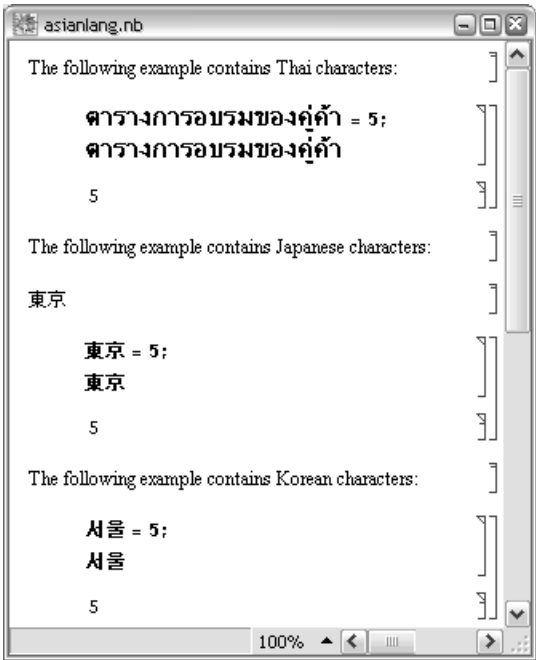
QuarterNote	EighthNote	BeamedEighthNote	BeamedSixteenthNote
♪	♪	♪	♪

DoubleStruckZero	0	DoubleStruckFive	5
DoubleStruckOne	1	DoubleStruckSix	6
DoubleStruckTwo	2	DoubleStruckSeven	7
DoubleStruckThree	3	DoubleStruckEight	8
DoubleStruckFour	4	DoubleStruckNine	9

In Version 4.2, the fonts were renamed Mathematica1, ... , Mathematica7.

□ **Internationalization Support**

With Windows XP and Windows 2000, full support for internationalization is provided. This feature allows *Mathematica* 4.2 running under the English version of the OS to display any language that can be read from left to right, including most Asian languages.

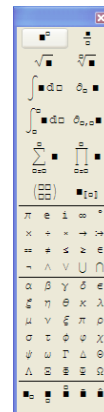
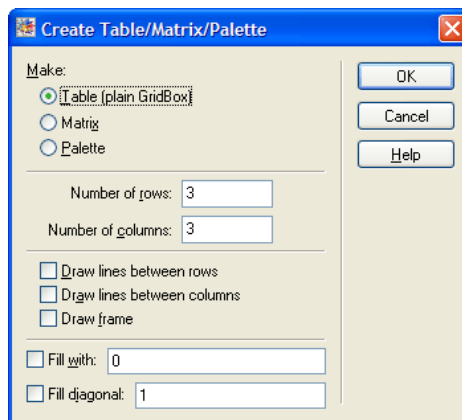
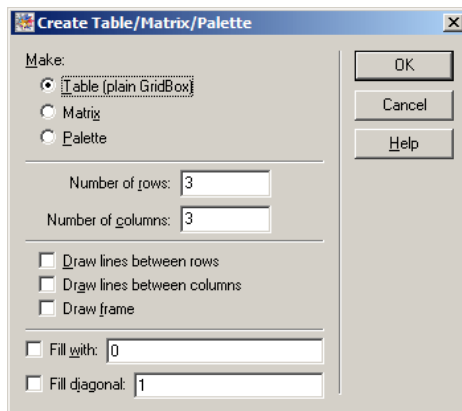


The Japanese language is supported under Linux with a language input server such as KInput2.

□ New Theme Support for Windows XP and Mac OS X

Newer operating systems such as Windows XP and Mac OS X now have window systems that support certain themes (or color schemes). The *Mathematica* 4.2 user interface was extended to support such themes. This new feature affects background colors for buttons and palettes. The colors will be automatically picked up when using options such as `Color→Automatic`.

Following are screen shots of *Mathematica* 4.2 running under two different themes.



■ Additional Import and Export Formats

Mathematica 4.2 added support for a number of new import and export formats. In addition to the NotebookML and MathML formats for notebooks and mathematical formulas mentioned earlier in this article, other formats supported include FITS, SDTS, SVG, and XHTML.

□ FITS

FITS is a data format designed by the astronomy community for the worldwide interchange of astronomical data. The standard is controlled by the International Astronomical Union. NASA has mandated the use of the FITS format for all data sets it produces, and the FITS standard is described in several NASA Office of Standards and Technology documents.

You can import a FITS file using the standard **Import** command. Here a conversion option is used to specify that the header information can also be imported.

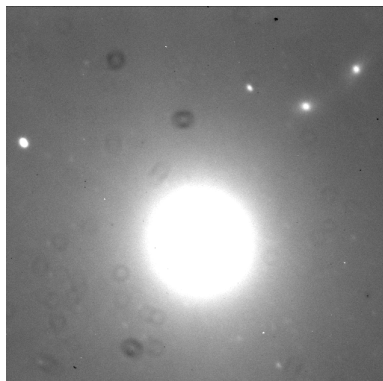
```
In[91]:= mypic =  
         Import["M87Galaxy.fits", ConversionOptions → {"Verbose" → True}];
```

You can download this file from the TMJ website at www.mathematica-journal.com/issue/v9i1.

In the case of image data, you can plot the data portion of the file by using **ListDensityPlot**.

```
In[92]:= ListDensityPlot[mypic[[1, 2]], Mesh → False,  
                        Frame → False, PlotRange → {{1, 512}, {1, 512}, Automatic}];
```

From In[92]:=



You can view the header information in the file by simply requesting the corresponding part of the imported data.

```

In[93]:= mypic[[1, 1]]

Out[93]:= {SIMPLE → {Value → True, Comment → Normal FITS file}, BITPIX →
  {Value → 16., Comment → Bits/pixel, negative is floating point},
  NAXIS → {Value → 2., Comment → Dimensionality of data},
  NAXIS1 → {Value → 552.}, NAXIS2 → {Value → 512.},
  EXTENDED → {Value → True, Comment → Extensions may be present},
  PCOUNT → {Value → 0., Comment → Deprecated},
  GCOUNT → {Value → 1., Comment → Deprecated},
  BSCALE → {Value → 1., Comment → Scale factor},
  BZERO → {Value → 0., Comment → Zero factor real=tape*bscale+bzero},
  CREATOR → {Value → 'Mathematica 4.0 for Linux (December 6, 1999)'},
  CREATOR2 → {Value → 'Release 2'},
  CREATOR3 → {Value → 'Created by ExportFITS package'}, END → {}}

```

□ SDTS

The SDTS format was designed by the United States government for transfer of spatial data such as digital elevation maps (DEMs) and satellite images. Each data archive contains a set of files. The specific file corresponding to the elevation data can be imported for analysis.

```

In[94]:= data = Import["9288CEL0.ddf",
  ConversionOptions → {"FillValue" → -50, "ElementSkip" → 2}];

```

You can download this file from the TMJ website at www.mathematica-journal.com/issue/v9i1.

Once imported, the data can be prepared for display. Here a **ColorFunction** is defined to make the data easier to visualize.

```

In[95]:= InterpolatingColorMap[c_] := Function[x, (n = Length[c] - 1;
  low = Floor[x n];
  If[low == n, low = n - 1,];
  high = low + 1;
  y = x n - low;
  Hue[Mod[c[[low + 1, 1]] + y (c[[low + 2, 1]] - c[[low + 1, 1]]), 1],
    c[[low + 1, 2]] + y (c[[low + 2, 2]] - c[[low + 1, 2]]),
    c[[low + 1, 3]] + y (c[[low + 2, 3]] - c[[low + 1, 3]])]];

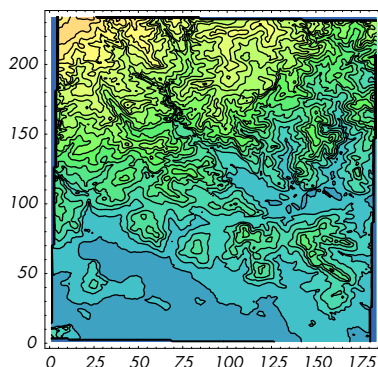
In[96]:= BlueGreenOrange =
  InterpolatingColorMap[{{0.6, 0.7, 0.7}, {0.3, 0.6, 1}, {0.1, 0.5, 1}}];

```

You can construct a contour map with **ListContourPlot** using the **ColorFunction** defined above.

```
In[97]:= ListContourPlot[data, Contours → 20, ColorFunction → BlueGreenOrange];
```

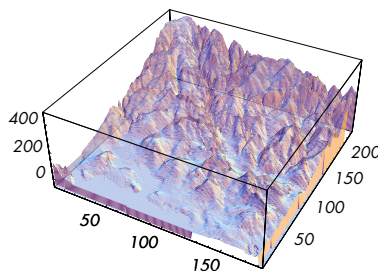
From In[97]:=



You can also visualize the data by creating a three-dimensional plot.

```
In[98]:= ListPlot3D[data, Mesh → False];
```

From In[98]:=



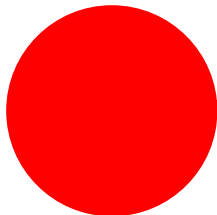
□ SVG

Scalable Vector Graphics (SVG) is an XML format developed by the W3C for describing two-dimensional graphics. It provides a convenient format for including images in web pages because it is more compact than other image formats and also allows for various types of interactivity. *Mathematica* 4.2 supports the export of graphics in SVG format.

Here is a simple graphic generated by *Mathematica*.

```
In[99]:= g = Show[Graphics[
  {RGBColor[1, 0, 0], Disk[{1, 1}, 1]}, AspectRatio → Automatic];
```

From In[99]:=



Here is the SVG markup describing the graphic.

```
In[100]:= ExportString[g, "SVG"]
Out[100]= <?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN" 'http://
www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd'>
<svg xmlns='http://www.w3.org/2000/svg'
width='288.0pt'
height='288.0pt'
viewBox='-0.05 -0.05 2.1 2.1'
preserveAspectRatio='xMidYMid meet'>
<g transform='scale(1, -1) translate(0, -2.0)'
style='fill:black; font-family:&apos;
Courier New&apos;;, Courier, Symbol, monospace; font-size:
0.04375pt; stroke:black; stroke-width:0.00182292pt'>
<ellipse fill='#f00'
cx='1.0'
cy='1.0'
rx='1.0'
ry='1.0'
stroke='none' />
</g>
</svg>
```

You can export the graphic as an SVG file using the standard **Export** function.

```
In[101]:= Export["C:\\Temp\\circle.svg", g, "SVG"];
```

□ EPS

It is now possible to import arbitrary Encapsulated PostScript (EPS) code into a notebook by using the File ▸ Import menu. The imported EPS code will not be rendered until the notebook is printed to a PostScript printer.

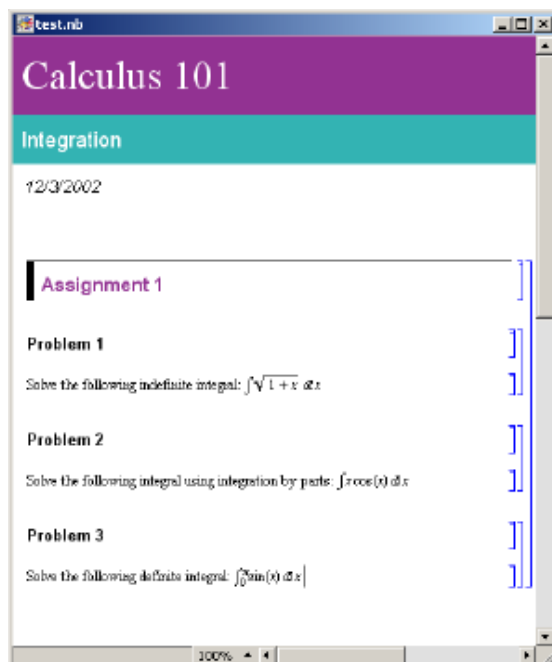
□ XHTML and CSS

The **Export** and **HTMLSave** functions now support XHTML by default. When exporting a notebook as HTML, you can now use conversion options to include style information in Cascading Style Sheets (CSS) syntax. Below are two examples that show how CSS style rules can be applied to HTML documents created by *Mathematica*.

The first example exports a single cell as HTML. The conversion option specified embeds a CSS style rule in the head element of the resulting HTML document.

```
In[102]:= Export["example1.html",
Cell["Hi", "Text"], "HTML", ConversionOptions →
{"HeadElements" → {"<style>.Text{color:blue}</style>"}]}
Out[102]= example1.html
```

You can also insert a reference to an external style sheet in the form of an HTML link element. Here is a notebook formatted using the built-in *Mathematica* style sheet Classroom.nb.



Here is the style sheet Classroom.css that contains definitions for some of the styles defined in the style sheet Classroom.nb.

```
.Title {color:white; background-color:purple; font-size:26pt; font-weight:normal; margin:0pt; padding:9pt; border:0pt;}

.Subtitle {color:white; background-color:teal; font-size:14pt; font-weight:normal; font-family:helvetica; margin:0pt; padding:7pt; border:0pt;}

.Subsubtitle {font-size:11pt; font-style:italic; font-weight:normal; font-family:helvetica; margin:0pt; padding:5pt; border:0pt;}

.Section {color:purple; font-size:12pt; font-family:helvetica; font-weight:bold; padding:5pt; border-top:solid thin black; border-left:solid medium black;}

.Subsection {font-size:10pt; font-family:helvetica; font-weight:bold;}

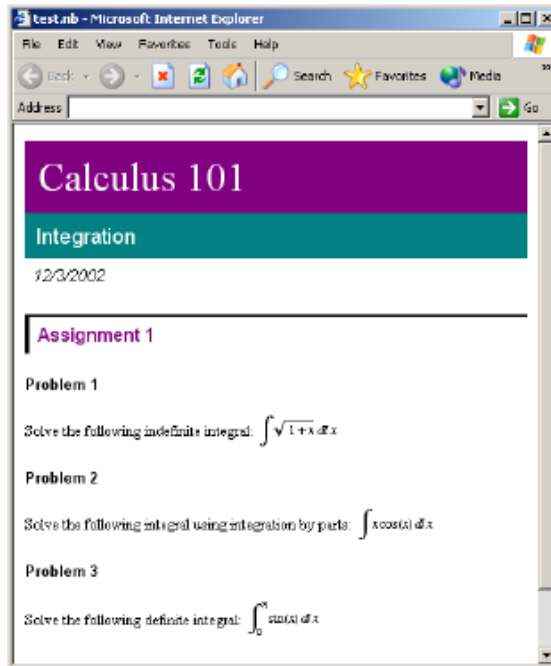
.Text {font-size:10pt;}
```

This exports the notebook to an HTML file that references the CSS style sheet defined above.

```
In[103]:= Export["test.html", "test.nb",
  ConversionOptions -> {"HeadElements" -> {"<link href=\"Classroom.
    css\" rel=\"stylesheet\" type=\"text/css\"/>"}]}]

Out[103]= example2.html
```

Here is the exported HTML document as seen in a web browser. You can see the close correspondence between the appearance of the HTML document and the original notebook. Using a CSS style sheet thus provides a convenient way to control the appearance of HTML documents created using *Mathematica*.



■ License Management

Starting with the release of *Mathematica* 4.1, *Network Mathematica* has allowed system administrators to easily deploy and administer *Mathematica* licenses in a central location, greatly simplifying system administration tasks. *Network Mathematica* also allows for concurrent use of *Mathematica* using *MathLM*, the *Mathematica* license manager. The new version of *MathLM* that ships with *Mathematica* 4.2 offers some major improvements and additional functionality.

□ Logging

The new *MathLM* logging facilities are extensive. The command syntax now defaults to W3C's common logfile format but is entirely customizable. The level of detail can be configured to report anything from only critical failures to every incoming and outgoing packet. In addition, *MathLM* sends critical messages to the system event Log.

□ Language Support

A translation facility allows users to define their own translation files to support either a new language or their own customized logging syntax. Translation files for French, German, and Japanese are included by default.

□ Policy Settings

The new policy support functions in *MathLM* are modeled on `.htaccess`. This permits system administrators to allow or deny certain domains or computers, or certain groups or individual users. For easy administration, various groups of users, machines, or domains can be given names and then used.

□ Dynamic Updating of *MathLM* Activity with New *MonitorLM*

MonitorLM is an application that allows system administrators to monitor the activity of the license manager. *MonitorLM* has been changed from a Unix shell script to a stand-alone application available on all platforms except Mac OS 9 and earlier. *MonitorLM* now also has customizable output and built-in support for plain text, HTML, and CGI output.

□ *MathLM* Now Available on All *Mathematica* Platforms

With the release of *MathLM* for Mac OS X, *MathLM* is now available for all operating systems supported by *Mathematica* except Mac OS 9 and earlier. In particular this means that you can have *Network Mathematica* running for a Macintosh-only lab if you run Mac OS X on at least one lab machine. In addition, *MathLM* for all platforms (Windows, Mac OS X, Linux [PC, Alpha, PowerPC], Solaris, HP/UX, IRIX, AIX, and HP Tru64 Unix) will come on its own CD.

■ Additional Materials

The following data files, used in some of the examples, are available at www.mathematica-journal.com.

ALHitters.xml. An XML file containing data on baseball players.

M87Galaxy.fits. An image of the M87 galaxy in FITS format.

9288.CELO.DDF. Sample geological data in the SDTS format.

About the Author

Lars Hohmuth is the Manager of the Strategic Marketing Group at Wolfram Research and Product Manager for the *Mathematica* product line. He holds a physics degree from the University of Ulm, Germany, and has been with Wolfram Research since 1996.

Lars Hohmuth

Product Manager for *Mathematica*
Wolfram Research, Inc.
larsh@wolfram.com

Article Submissions to *The Mathematica* *Journal*

The Mathematica Journal publishes articles and notes on all aspects of *Mathematica* and related subjects. Its goal is to inform and challenge the community of *Mathematica* users and to enable readers at all levels of proficiency to use *Mathematica* more effectively. A high standard of *Mathematica* programming, using clarity and conciseness in writing, is valued. *The Mathematica Journal* reserves the right to edit contributions, with changes subject to the authors' approval.

The Mathematica Journal also publishes *Mathematica* notebooks and packages, as well as programs in other languages that can be used to complement *Mathematica*. All programs are accompanied by a note or article. Programs distributed through the journal can be used freely and duplicated for noncommercial purposes.

Some of the categories of material published in *The Mathematica Journal* are listed below.

- Reports of applications of *Mathematica* to particular fields in education, research, or business. The level of exposition should be such that readers from other fields can grasp the essence of the application, the reasons why *Mathematica* is being used, and the possible relevance to other fields of the techniques being employed.
- Descriptions of programs—*Mathematica* packages and notebooks, or *Mathematica*-related programs in other languages—whether or not accompanied by the code itself. The programs should be exemplary in their implementation and thoroughly tested.
- *Mathematica* programming advice. This can range from notes about little-known programming tricks to tutorials focusing on a general problem or technique. The main test of applicability here is how many readers are likely to benefit from the ideas, but even a relatively esoteric application can aid in the teaching of good programming principles.
- Discussion of issues beyond *Mathematica*. Essays, columns, and surveys on subjects that do not involve *Mathematica* directly may be considered, if they are likely to be of general interest to the *Mathematica* community. Possible subjects include symbolic computation, numerical computation, computer-based education, computer graphics, computer art, and so forth. Articles of this type should be particularly well written and well

argued, and their relevance to the *Mathematica* community should be clear.

- General informational articles, profiles of exemplary *Mathematica* users or applications, interviews, and news related to the *Mathematica* community will also be sought for publication online on a frequent basis.

To submit a contribution, send the material—together with a cover letter stating that it is being submitted for publication in *The Mathematica Journal*—in one of the following formats. Use filenames that suggest their contents.

Web

Articles, graphics, and other files may be uploaded on the Internet at www.mathematica-journal.com/upload.

Email

Articles and other material may be sent by email to submissions@mathematica-journal.com. Do not email articles or attachments that exceed 1MB.

Hard Copy and/or Physical Media by Mail

Send to:

Editor
The Mathematica Journal
PO Box 6359
Champaign, IL 61826-6359 USA

Articles, packages, graphics, notebooks, and programs submitted for publication can be sent on cdrom, zip, or other disk. Submissions will not be returned. Do not send proprietary or confidential information.

■ Manuscript Submission Guidelines

Manuscripts must be in English. Manuscripts should be submitted as properly formatted notebooks made with *Mathematica* version 3 or later. Notebooks should be well structured, making good use of sections, text, and inline and displayed equations (if required), and should use the *Mathematica* style sheet “TMJv4Style.nb”. This style sheet along with an authoring palette can be downloaded from www.mathematica-journal.com/submissions.

Manuscripts should present a significant application of *Mathematica*. Manuscripts under consideration by or submitted simultaneously to other journals will not be considered.

Remember to include the following:

1. the title, if the title has more than 50 characters, submit an alternative short title;
2. your name(s), affiliation, postal, and electronic mail address;
3. an abstract of at most 150 words.

The remaining material should be included in the following order:

1. text of the article or note;
2. references (see below);
3. a list of additional electronic material, including, if appropriate, information on the necessary environment (machine, operating system, and so on);
4. a biography of each author, limited to 60 words per author, and 180 words overall.

References should be numbered in the text in order of appearance. In the listing at the end of the article, each reference should include full information: author or institution, full title, publisher, city, and year (for books, manuals, etc.); or full journal name, volume, year, and page range (for papers). References to software and hardware should contain complete manufacturer's or distributor's names and addresses. See format in a current issue.

All references in the bibliography should be cited in the text, or accompanied by comments stating their relevance.

Avoid footnotes if possible. If used, they should be numbered sequentially and clearly marked in the text, with the footnotes as a section at the end of the article.

Please observe the following guidelines for text and embedded code.

- Use of the word *Mathematica* should be kept to a minimum, since the reference is almost always clear from the context. It is especially discouraged in titles.
- Technical terms unlikely to be known to nonspecialists should be italicized the first time they are used, and, unless the first use is self-explanatory, they should be defined or explained then. Italics should not be used for emphasis, except very occasionally.
- *Mathematica* words, symbols, or expressions should appear in a font distinct from that used for text. The use of a fixed-pitch font, in which all characters have the same width, is strongly encouraged. Use the author's palette and *The Mathematica Journal* style sheet.
- References to *Mathematica* functions should not be followed by **[]**, unless what is meant is that the function is being invoked with no arguments.
- Lines of displayed code, whether from packages or interactive sessions, should be no longer than 60 characters.
- Do not use *Mathematica* comments in interactive sessions; instead, alternate explanatory text using complete sentences with input/output pairs. It is especially important to explain the commands in a long interactive session as you go along, rather than all at once at the beginning or at the end of the session. As a rule, notebooks should be cast into this same format: explanatory text alternating with code.
- Always mention which version of *Mathematica* you are running.

■ Guidelines for *Mathematica* Packages

Packages are programs written in *Mathematica*, to be used on their own or integrated with the user's code.

The single most important consideration in designing a *Mathematica* package for distribution is that it must be readily usable by others. Clarity is the goal and a good test of the clarity of your design is the ease with which it can be documented. Ideally, the purpose and use of each symbol visible to the user should be simple enough to explain in its usage string.

The more simply your design can be described, the more likely people are to be able to understand and use what you have built. In addition, such designs are often more general and therefore usable in a wider range of circumstances.

The book *Programming in Mathematica*, Third Edition, by Roman Maeder (Addison-Wesley, 1996) provides good pointers on programming style.

RiskQ 4.2

Software for Probability and Statistics

RiskQ provides tools for developing empirical and theoretical distributions and using these for hypothesis testing. Applications include:

○ **Experimental Data Analysis**

- Perform ANOVA, MANOVA, ANOCOVA, and GLM analyses
- Adjust p -values from multiple tests
- Apply common parametric and nonparametric statistical tests
- Compare correlation matrices

○ **Toxicology and Environmental Chemistry**

- Conduct statistical tests on data
- Characterize data variation graphically and analytically

○ **Epidemiology and Clinical Trials**

- Analyze associations exhibited by epidemiological data
- Perform adjusted chi-square tests of homogeneity and association
- Obtain the approximate or exact likelihood of observed contingency table data

○ **Risk and Uncertainty Analysis**

- Characterize distributions reflecting uncertainty
- Perform parametric and nonparametric sensitivity analysis
- Characterize Monte-Carlo estimation error of simulated output

Visit us at:

www.wolfram.com/products/applications/riskq

Loehle Enterprises

www.loehleenterprises.com

info@loehleenterprises.com

Classifieds

Classified advertising is provided for *Mathematica* users. Rates are available at www.mathematica-journal.com/advertising.

RiskQ 4.2: Software for Probability and Statistics—RiskQ provides tools for developing empirical and theoretical distributions and uses these tools for hypothesis testing. Applications include experimental data analysis, toxicology, epidemiology, and risk analysis. To learn more about RiskQ, visit the website at www.wolfram.com/products/applications/riskq. The developer of RiskQ can be contacted at:

Loehle Enterprises
1258 Windemere Avenue
Naperville, IL 60564
phone: 630-527-8402
fax: 630-527-8502
email: info@loehleenterprises.com
web: www.loehleenterprises.com

Mathematica Training and Consulting—Whether you are an individual or a company seeking *Mathematica* training, Wolfram Education Group provides a learning solution with **Certified Training**. Register for classes and review the Education Benefits Program for corporate account training incentives and discounts. If you are looking for **Independent Training** for additional courses, workshops, or tutorials, the Wolfram Education Group can provide a variety of learning opportunities from independent trainers and companies with years of *Mathematica* experience. Wolfram Research also offers direct **Consulting Services**, such as reviewing and debugging your *Mathematica* code, isolating problems in large *Mathematica* programs, implementing custom features, and writing complete programs based on users' specifications.

More info: www.wolfram.com/services/training

Custom Calculus Solutions Instantly—On this site you get automatic step-by-step derivatives and integrals with each step explained. Long multiplication and division of polynomials are also provided.

More info: www.calc101.com
