

The *Mathematica*[®] Journal

Volume 10, Issue 1

Contents

Departments

Staff and Contributors	2
Calendar	3
News Bulletins	4
New Products	6
New Resources	11
New Publications	13
Article Submissions to <i>The Mathematica Journal</i>	265
Classifieds	269

Column

Trott's Corner by <i>Michael Trott</i>	15
--	----

Feature Articles

Efficient, High-Quality Force-Directed Graph Drawing by <i>Yifan Hu</i>	37
Ordered and Unordered Factorizations of Integers by <i>Arnold Knopfmacher and Michael Mays</i>	72
Approximating Solutions of Linear Ordinary Differential Equations with Periodic Coefficients by Exact Picard Iterates by <i>Armando G. M. Neves</i>	90
A Flexible Implementation for Support Vector Machines by <i>Roland Nilsson, Johan Björkegren, and Jesper Tegnér</i>	114
Configuration Analysis and Design by Using Optimization Tools in <i>Mathematica</i> by <i>Frank J. Kampas and János D. Pintér</i>	128
The Xmath Calculator by <i>Odd Bringslid</i>	155
Elements—An Object-Oriented Approach to Industrial Software Development by <i>Gerd Baumann and Michal Mnuk</i>	161
<i>MathModelica</i> —An Object-Oriented Mathematical Modeling and Simulation Environment by <i>Peter Fritzson</i>	187

The electronic version of this issue is available at www.mathematica-journal.com/issue/v10i1.

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

On the Cover

Images from *Wolfram Tones* (tones.wolfram.com), an experiment in a new kind of music made possible by *Mathematica* and *A New Kind of Science*.

Staff and Contributors

Managing Editor: George Beck

Project Manager: Peggy Pope

Design Team: Jeremy Davis, Megan Gillette

Compositor: Glenn Scholebo

Archives and Web Production: Jean Buck

Web Development: Andy Hunt, Lisa Denlinger

Editorial Staff: George Beck, Marcia Krause, Larry Adelston, Lorri Coey

Columnist: Michael Trott

Contributors: Gerd Baumann, Johan Björckegren, Odd Bringslid, Peter Fritzon, Yifan Hu, Frank J. Kampas, Arnold Knopfmacher, Michael Mays, Michal Mruk, Armando G. M. Neves, Roland Nilsson, János D. Pintér, Jesper Tegnér

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); Daniel Lichtblau (Wolfram Research, Inc.); David Lovelock (University of Arizona); Jerry Marsden (Caltech); Robert Moore (University of Washington, Seattle); Flip Phillips (Skidmore College); Bill Press (Harvard University); Sam Savage (University of Chicago); Dana Scott (Carnegie Mellon University); Lawrence M. Seiford (University of Michigan, Ann Arbor); Lynn Steen (St. Olaf College); Francis Sullivan (Institute for Defense Analyses); Bill Thurston (University of California, Davis); Michael Trott (Wolfram Research); Alfred van der Poorten (Macquarie University); Hal Varian (University of California, Berkeley); Eric Weisstein (Wolfram Research); Rob Wolff (Apple Computer, Inc.); and Stephen Wolfram (Wolfram Research).

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 5* 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.

© 2006 Wolfram Media, Inc. 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. All other trademarks are the property of their respective owners. *Mathematica* is not associated with Mathematica Policy Research, Inc. or MathTech, Inc. 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

For the complete Wolfram Research calendar, see
www.wolfram.com/news/events.

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

2006 Highlighted Events

March

APS March Meeting 2006

American Physical Society

March 13–17, Baltimore, Maryland

Info: www.aps.org/meet/MAR06

Mathematica User Day 2006

CANdiensten

March 22, Eindhoven, the Netherlands

Info:

www.can.nl/english/cursussen/frameset.asp?boven=cursussendetail.asp?id=81

June

NKS 2006 Wolfram Science Conference

Wolfram Research

June 16–18, Washington, DC

Info: www.wolframscience.com/conference/2006

International Mathematica Symposium 2006

IMS'06

June 19–23, Avignon, France

Info: internationalmathematicasymposium.org/IMS2006

News Bulletins

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

World's Largest 3D Display Screen Created with *Mathematica*

X3D Technologies GmbH used *Mathematica* to create the world's largest-ever 3D projection wall display—with a diagonal measurement of over 15 feet. The display, featured at the 2005 World Expo in Japan, allows lifelike images to be seen by the naked eye from wide viewing angles at any perspective. For more information, go to www.wolfram.com/news/3dscreen.html.

Mathematica Used to Model Restructured Electric Power Markets

Researchers at Argonne National Laboratory in the United States are relying on *Mathematica* to explore electric power markets and prevent outages from occurring in major U.S. cities. *Mathematica* has allowed them to visualize and analyze data, model real-world systems, and predict the effects of deregulation. For more information, go to www.wolfram.com/news/anlpower.html.

Next-Generation Prosthetics Designed in *Mathematica*

The Eindhoven University of Technology and the Netherlands Organization for Applied Research have developed new algorithms and a complete workflow in *Mathematica* to build customized endplates for spinal disc prostheses. Analysis results are exported directly from *Mathematica* to a machine that creates the endplates that are later attached to the spine. About 15–20% of all people suffer from chronic back pain, and these advanced *Mathematica*-based methods are helping to treat them. For more information, go to www.wolfram.com/news/spinaldisc.html.

Machine Learning Optimizes Automated Assembly Lines

Mathematica and the *machine learning framework* (MLF) application package, are an integral part of the production process for several leading companies, including AMS Engineering. AMS relies on the data-mining and modeling capabilities of MLF to improve its overall equipment efficiency and discrete manufacturing processes. The company uses *Mathematica* throughout to create the right models

offline and ensure the efficiency of the shop floor management systems during production. For more information, go to www.wolfram.com/news/mlfsystems.html.

Archaeological Breakthrough Made Possible by *Mathematica*

Mathematica has helped lead to a breakthrough in the deciphering of Incan khipu, ancient string bundles thought to have been used for communication and administrative records. Harvard University researchers used *Mathematica* to analyze the knot patterns, uncovering many interesting mathematical similarities. For more information, go to www.newscientist.com/article.ns?id=dn7835.

Key Legal Precedents Analyzed in *Mathematica*

Seth Chandler, a professor of law at the University of Houston, was featured in *The Economist* for his innovative use of computer modeling to assess the importance of past United States Supreme Court decisions based on the frequency of their subsequent citations. Chandler also presented a paper on this topic at the 2005 International *Mathematica* Symposium. For more information, go to economist.com/science/displayStory.cfm?story_id=4316174.

Financial Startup Venture Turns to *Mathematica* for Modeling

Billionaire Gordon Getty's ReFlow Management startup company uses *Mathematica* for its sophisticated financial modeling needs. For more information, go to www.businessweek.com/bwdaily/dnflash/sep2005/nf20050929_5281_db049.htm.

New Products

Mathematica Personal Grid Edition

Personal Supercomputing Has Arrived

With *Mathematica Personal Grid Edition* and new cost-effective quad-core computers, personal supercomputing is now a reality. *Mathematica Personal Grid Edition* eliminates the barriers to using parallelism as part of your daily workflow—with no administrative overhead and no contending for shared resources—and opens the door to new possibilities in high-performance computing. You can easily tackle larger problems and investigate parallel approaches at any stage of the problem-solving process, right at your desk and at your own convenience.

Mathematica Personal Grid Edition combines the computational capabilities of *Mathematica* with high-level parallel language extensions to create an optimal computing framework for quad-core machines, instantly changing the supercomputing landscape. *Mathematica*'s high-level language and state-of-the-art numeric and symbolic solvers make it ideal for doing exploratory technical computing in a wide range of fields, resulting in increased productivity and letting you push the limits further than ever before.

The *Mathematica* 5 series (5.0, 5.1, 5.2) provided numerous performance and scalability enhancements to the software of choice for demanding technical computing applications. All-platform 64-bit computing has broken the memory barrier, while fast machine-optimized dense and sparse linear algebra gives *Mathematica* unparalleled numerical speed. These versions also incorporate major new functionality for connectivity—including built-in universal database link, web services, and numerous file format converters—as well as hundreds of new numerical and symbolic algorithms.

grid $\mathit{Mathematica}$ extends this high-level framework to enterprise parallel computing. Now *Mathematica Personal Grid Edition* is the only solution to bring high-performance supercomputing to the individual desk, continuing the Wolfram Research commitment to cutting-edge innovation.

“Traditionally, parallel computing has been the realm of specialist programmers with limited understanding of different application fields. Researchers had to cross multiple barriers—including prohibitive cost, administrative overhead, remote locations, and hard-to-use APIs—to perform parallel computations, often putting this resource beyond their reach,” said Roger Germundsson, director of research and development at Wolfram Research. “Now parallel computing can directly be used by individual researchers as an integral problem-solving method. With *Mathematica Personal Grid Edition* and a quad-core machine, you have an immediately usable environment for exploring parallelism at any level.”

Mathematica Personal Grid Edition offers performance improvements of up to 300% over standard *Mathematica* with only minimal code modifications. For more-intensive parallel applications, programs can run unchanged on grids or clusters of any size using *gridMathematica*. This also makes *Mathematica Personal Grid Edition* the perfect prototyping environment for large-scale parallelism, and adds another dimension to Wolfram Research's ability to conveniently handle all of your supercomputing needs within the same basic framework.

Mathematica Personal Grid Edition is available for Windows, Mac OS X, Linux, and Unix, for both 32-bit and 64-bit systems.

More info

www.wolfram.com/products/personalgrid

gridMathematica 2.1

gridMathematica 2.1 Capitalizes on Large-Scale Improvements in Mathematica 5.2

gridMathematica has been updated for high-performance computing to include *Mathematica* 5.2, the 64-bit and multicore release that vastly increases memory and CPU availability.

Highlights of *gridMathematica* 2.1 for large-scale computation include:

- All-platform support for 64-bit addressing
- Multicore support on major platforms
- Multithreaded numerical linear algebra
- 64-bit-enhanced arbitrary-precision numerics
- Vector-based performance enhancements
- *MathematicaMark* 5.2 benchmark now covering grids and clusters

With these added features, *gridMathematica* is now an even better solution for handling the most demanding large-scale problems and simulations. Universal 64-bit technology means that there's effectively no memory barrier—theoretically addressing up to 16 exabytes—so *gridMathematica* is limited only by the operating system and hardware. Version 2.1 also supports automatically threaded numerical linear algebra, enabling linear algebra operations to automatically run in parallel on all available processor cores.

For further system optimization, the new *MathematicaMark* 5.2 benchmark tool provides for scalability of *Mathematica* benchmark scores, allowing benchmarking of SMP machines, clusters, and grids. Quick benchmark reports show how a grid or cluster compares for overall *Mathematica* performance as well as specific computations, providing an accurate, real-world estimation of how well a system performs and whether or not investing in different hardware would be advantageous.

grid $\textit{Mathematica}$ 2.1 is available for all common Linux and Unix systems, Windows, and Mac OS X. Computers in a cluster do not have to run the same operating system.

More info

www.wolfram.com/products/gridmathematica

Mathematica CalcCenter for Students: Semester and Annual Editions

Students Save with Affordable New Licensing Options

New semester and annual editions make *Mathematica CalcCenter for Students* available for as little as half the price of an average textbook.

Characterized as having “the power of *Mathematica*, the ease of a calculator,” *Mathematica CalcCenter* gives students the same core *Mathematica* technology used every day by science and technology leaders around the world. With the award-winning interface, new users can get up and running in less than 10 minutes.

Unique features such as `SmartPlot` and `InstantCalculators` help students accurately set up and solve tough symbolic and numeric problems for almost any math or science class. Results can then be saved and combined into rich reports and professional presentations—all within one integrated system.

“While it’s probably the simplest technical software on the market to use, it handles some of the most sophisticated tasks,” said Jon McLoone, lead developer of *Mathematica CalcCenter*. “All the key *Mathematica* features a student needs are right there—just pick, click, and solve.”

Mathematica CalcCenter for Students is available for Windows and Mac OS X.

More info

www.wolfram.com/products/calccenter/student.html

Mathematica Applications

Digital Image Processing 2

Digital Image Processing 2 introduces over 60 new functions, options, and major modules for fast and powerful image processing on the desktop. The package integrates with *Mathematica* to automate custom analyses, run routine procedures, and develop new algorithms.

More info

www.wolfram.com/products/applications/digitalimage

Geometrica05

Geometrica05 extends *Mathematica*'s symbolic engine to a wide range of geometrical analysis, exact drawing, and CAD applications.

More info

www.wolfram.com/products/applications/geometrica

Global Optimization 5.1

Global Optimization is now five times faster at global nonlinear optimization with *Mathematica* and can solve problems with over 10,000 variables. Any *Mathematica* function can be used as input, including the degree of fit of a model against data, black-box functions, and simulation models.

More info

www.wolfram.com/products/applications/globalopt

LinkageDesigner

The new *LinkageDesigner* package is a time-saving engineering solution for analyzing, synthesizing, and simulating linkages with serial chain, tree, and graph structures. It simplifies the machine design process, allowing engineers to perform more extensive analyses than possible with other software.

More info

www.wolfram.com/products/applications/linkagedesigner

machine learning framework 1.3

machine learning framework 1.3 includes a whole new range of functions for real-life data-mining projects. It is a complete solution for business and financial engineers, process and manufacturing engineers, quality-assurance professionals, and all experts who want to extract computational models from data.

More info

www.wolfram.com/products/applications/mlf

MathGL3d

MathGL3d is an interactive, OpenGL-based 3D-graphics package for advanced scientific visualization with *Mathematica*. It allows real-time interaction with *Mathematica* graphics, using features such as rotation, animation, visual view-point, and light-source editing.

More info

www.wolfram.com/products/applications/mathgl3d

MechanicalSystems 2

MechanicalSystems 2 speeds up prototyping and simulation tasks for complex models of rigid body systems. Over forty 2D and 3D geometric constraints enable users to instantly visualize changes. New documentation fully integrates into the searchable *Mathematica* Help Browser.

More info

www.wolfram.com/products/applications/mechsystems

PathwayLab Research Edition

PathwayLab Research Edition is a new *in silico* systems biology application for modeling and analyzing biochemical pathways and networks. It leverages the capabilities of *Mathematica* to help predict disease pathways and reduce the need for physical experiments.

More info

www.wolfram.com/products/applications/pathwaylab

UnRisk PRICING ENGINE 2.2

UnRisk PRICING ENGINE 2.2 significantly boosts speed without sacrificing precision for financial derivatives analysis.

More info

www.wolfram.com/products/applications/unrisk

New Resources

The Wolfram Technology Guide

The Wolfram Technology Guide reveals the technology inside *Mathematica* that makes it such a unique and powerful technical computing system. The easily navigated interactive interface provides essential overviews of many key technologies, complete with visual illustrations and animated examples. Links on each page lead to related *Mathematica* features and helpful documentation elsewhere on the Wolfram Research website. Explore the guide at www.wolfram.com/technology/guide.

Wolfram Technology Conference 2005 Proceedings

The 2005 Wolfram Technology Conference offered nearly 100 presentations, classes, and workshops on Wolfram technologies and their applications. Notebooks of these presentations are now available online at www.wolfram.com/techconf2005.

Mathematica Shortcuts

A printable reference sheet of all *Mathematica* 5.2 keyboard shortcuts for Windows, Macintosh, Linux, and Unix is now available in PDF format. Download this handy resource from media.wolfram.com/brochures/mathematicashortcuts.pdf.

WolframTones

WolframTones is an innovative online music and ringtone generator built on core discoveries from Stephen Wolfram's *A New Kind of Science*. It features a fully interactive user interface that lets anyone instantly compose original music based on abstract cellular automata patterns and download it to their cellular phones. Compositions can be enhanced by adding instruments and percussion, adjusting scales and tempo, and selecting from a wide variety of musical styles. Create your unique tones at tones.wolfram.com.

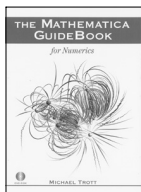
Scientific Visualization and Graphics with *Mathematica*

A new website, Scientific Visualization and Graphics with *Mathematica*, showcases practical 2D and 3D graphics and animations. In addition to being a home for these often artful images, it is a site for learning the basics of *Mathematica* visualization and how to use those methods to individual applications. Each graphic or animation is accompanied by its *Mathematica* notebook, so visitors can see all the code and learn how to create similar visualizations on their own. Browse this site at members.wolfram.com/jeffb/visualization.

Mathematica in Education and Research

Volume 10, Number 4 of *Mathematica in Education and Research* (*MiER*) is now online. The new issue has articles on front-end programming, Sudoku puzzle solving, mathematical graphics, and more. *MiER* is online at www.ijournals.net.

New Publications



The *Mathematica* GuideBook for Numerics

Michael Trott, 2006, Springer-Verlag, 1208 pp., hardcover, includes DVD-ROM

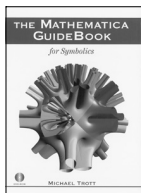
The Mathematica GuideBook for Numerics introduces and explains the available types of arithmetic (machine, high-precision, and interval) in *Mathematica*. Fundamental numerical operations, such as compiling programs, fast Fourier transforms, minimization, numerical solutions of equations, and ordinary/partial differential equations, are analyzed in detail and are applied to a large number of examples in the main text and in the solutions to the exercises.

Contents

Numerical Computations | Computations with Exact Numbers

Available:

store.wolfram.com/view/book/ISBN0387950117.str



The *Mathematica* GuideBook for Symbolics

Michael Trott, 2006, Springer-Verlag, 1453 pp., hardcover, includes DVD-ROM

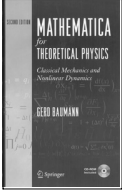
Structural and mathematical operations on single and systems of polynomials are fundamental to many symbolic calculations, and they are covered here in considerable detail. The solution of equations and differential equations, as well as the classical calculus operations (differentiation, integration, summation, series expansion, limits) are exhaustively treated. Generalized functions and their uses are discussed. In addition, this volume discusses and employs the classical orthogonal polynomials and special functions of mathematical physics. To demonstrate the symbolic mathematics power, a large variety of problems from mathematics and physics are presented.

Contents

Symbolic Computations | Classical Orthogonal Polynomials | Classical Special Functions

Available:

store.wolfram.com/view/book/ISBN0387950206.str



***Mathematica* for Theoretical Physics: Classical Mechanics and Nonlinear Dynamics, Second Edition**

Gerd Baumann, 2005, Springer-Verlag, 544 pp., hardcover, includes CD-ROM

This book demonstrates how to solve physical problems and deal with their underlying theoretical concepts while using *Mathematica* to derive numeric and symbolic solutions. The accompanying CD-ROM provides *Mathematica* programs and notebooks of the entire text, making the examples interactive and flexible. The second edition includes a full course in classical mechanics, new examples in quantum mechanics, and measurement methods for fractals.

Contents

Introduction | Classical Mechanics | Nonlinear Dynamics | Electrodynamics | Quantum Mechanics | General Relativity | Fractals | Appendices

Available:

store.wolfram.com/view/book/ISBN0387016740.str

Scattering States in a 2D Quantum Dot Embedded in a Waveguide

Michael Trott

The eigenfunctions belonging to the continuous spectrum of a circular quantum dot connected to two infinite strip-like wires are calculated using the mode-matching technique. The corresponding quantum-mechanical probability densities, currents, and transmission and reflection coefficients are calculated and visualized.

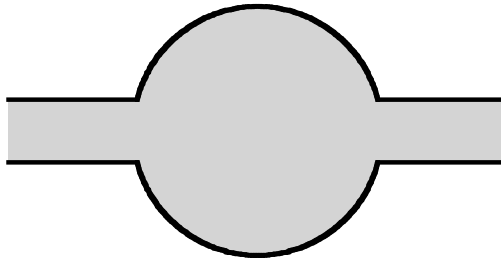
■ Introduction

In the 9:3 Corner, I discussed the sound of a fractal drum. More technically speaking, I calculated selected point spectrum eigenvalues and eigenfunctions of the Helmholtz equation of a complicated, bounded 2D domain with homogeneous Dirichlet boundary conditions. In this Corner, I will treat a problem similar in spirit—I will calculate some eigenfunctions of the continuous spectrum of an unbounded 2D domain with homogeneous Dirichlet boundary conditions. More concretely, we will study the transmission of a wave through a 2D quantum dot in the stationary regime. Here is a sketch of the domain Ω under consideration: (the x -direction runs to the left and the y -direction runs upwards).

```

In[1]:= Module[{R = 10, L = 5,  $\phi$ },  $\phi$  = ArcSin[(L/2)/R];
  Show[Graphics[(* left and right arms and center disk *)
    {GrayLevel[0.8], Disk[{0, 0}, R],
      Rectangle[{-2 R, -L/2}, {2 R, L/2}]},
    (* boundaries with homogeneous Dirichlet boundary conditions *)
    {GrayLevel[0], Thickness[0.01], Circle[{0, 0}, R, { $\phi$ , Pi -  $\phi$ ]},
      Circle[{0, 0}, R, {Pi +  $\phi$ , 2 Pi -  $\phi$ ]},
      Line[{{-2 R, +L/2}, {-R Cos[ $\phi$ ], +L/2}}],
      Line[{{2 R, +L/2}, {R Cos[ $\phi$ ], +L/2}}],
      Line[{{-2 R, -L/2}, {-R Cos[ $\phi$ ], -L/2}}],
      Line[{{2 R, -L/2}, {R Cos[ $\phi$ ], -L/2}}]}],
    PlotRange -> All, AspectRatio -> Automatic]]

```



The arms have lateral width L and infinite horizontal extension, and the center disk has radius R . To make calculations simpler, we take the arms to extend up to the circular boundary of the center disk.

We will solve the elliptic eigenvalue problem $-\Delta \psi(\mathbf{r}) = \mathcal{E} \psi(\mathbf{r})$ with the boundary condition $\psi(\mathbf{r})|_{\mathbf{r} \in \partial\Omega} = 0$. We will assume an incoming wave from the left with unit strength of the form $\exp(i k x) \chi(y)$. We will use the mode-matching technique to solve this Helmholtz equation. That is, we will construct within neighboring regions locally parametrized complete families of solutions and join them smoothly along certain curves by fixing the values of the parameters. We will naturally choose the left arm, the center disk, and the right arm for the regions.

A practical realization of such structures are quantum dots [1–8]. The Helmholtz equation in this case is identical to the time-independent Schrödinger equation with energy $\mathcal{E}$. We will use this model as the physical interpretation in our calculations.

Because the domain Ω is symmetrical, we can construct eigenfunctions with a definite parity with respect to reflections in the x -axis. We will always consider symmetric states. The construction of antisymmetric states is completely analogous.

Inside the left arm, we can write the general solution of the Helmholtz equation in the form

$$\psi(x, y) = e^{i k_1 x} + \sum_{m=1}^{\infty} \alpha_m e^{-i k_m x} \cos\left(\frac{(2m-1)\pi y}{L}\right) \approx e^{i k_1 x} + \sum_{m=1}^p \alpha_m e^{-i k_m x} \cos\left(\frac{(2m-1)\pi y}{L}\right) \xrightarrow{x \rightarrow -\infty} (e^{i k_1 x} + \mathcal{R} e^{-i k_1 x}) \cos\left(\frac{\pi y}{L}\right).$$

Here the inhomogeneous term has strength 1. We truncate the infinite sum because the contributions of the higher states decay exponentially with m for energies in the range $(\pi/L)^2 < \mathcal{E} < (3\pi/L)^2$ (the lowest sub-band in the y -direction is occupied). The lateral wave number k_m is, for a given energy $\mathcal{E}$, defined through $\mathcal{E} = k_m^2 + ((2m-1)\pi/L)^2$. The first term in the expansion contains the reflection coefficient $\mathcal{R}$. The form of the y -dependent part automatically ensures the fulfillment of the Dirichlet boundary conditions in the lower and upper boundaries of the left arm.

Similarly, in the right arm, we use the expansion

$$\psi(x, y) = \sum_{m=1}^{\infty} \gamma_m e^{i k_m x} \cos\left(\frac{(2m-1)\pi y}{L}\right) \approx \sum_{m=1}^q \gamma_m e^{i k_m x} \cos\left(\frac{(2m-1)\pi y}{L}\right) \xrightarrow{x \rightarrow \infty} \mathcal{T} e^{i k_1 x} \cos\left(\frac{\pi y}{L}\right).$$

Again, we truncate the infinite sum. This time, the first term in the expansion contains the transmission coefficient $\mathcal{T}$ and k_m is defined as in the left arm.

Inside the center disk, we use a polar coordinate system. In polar coordinates, any solution of the Helmholtz equation can be written in the form

$$\psi(r, \varphi) = \sum_{m=0}^{\infty} \beta_m \cos(m\varphi) \mathcal{J}_m(\sqrt{\mathcal{E}} r) \approx \sum_{m=0}^o \beta_m \cos(m\varphi) \mathcal{J}_m(\sqrt{\mathcal{E}} r).$$

Here the $\mathcal{J}_m(\zeta)$ are Bessel functions.

The $p + q + (o + 1)$ unknown coefficients α_k , β_k , and γ_k are determined through the continuity of $\psi(x, y)$, the first derivative of $\psi(x, y)$ along curves joining the left and right arms with the center disk, and the fulfillment of the Dirichlet boundary condition along the arcs between the arms and the center disk. We will unite the three sets of coefficients α_k , β_k , and γ_k into the $p + q + (o + 1)$ coefficients c_k and use the following convention:

$$\begin{aligned} c_1, \dots, c_p &\equiv \alpha_1, \dots, \alpha_p \text{ (left arm)} \\ c_{p+1}, \dots, c_{p+q} &\equiv \gamma_1, \dots, \gamma_p \text{ (right arm)} \\ c_{p+q+1}, \dots, c_{p+q+o+1} &\equiv \beta_0, \dots, \beta_{o+1} \text{ (center disk).} \end{aligned}$$

■ Calculate Fourier Coefficients

To determine the coefficients c_k , we must construct equations encoding the continuity of the solution and its first derivative. As the joining curves, we will use the parts of the boundary of the center disk that intersects both arms. Along these arcs, we demand that the first d Fourier coefficients of the difference between the solution ansatz of the arms and of the disk vanish.

```
In[2]:= Off[General::spell]; Off[General::spell1];
```

We denote by ϕ the angle from the positive x -axis to the intersection of the center disk with the upper boundary of the right arm.

```
In[3]:=  $\phi = \text{ArcSin}[(L/2)/R];$ 
```

We use these global parameters: the energy $\mathcal{E}$, the number of states in the three regions $p, o + 1$, and q , the radius and width R and L , and ϕ .

To abbreviate the following inputs, we define these solution ingredients: a left running mode, a right running mode, and a mode for the center disk. Though we restrict ourselves to solutions symmetric with respect to the x -axis, the antisymmetric states can be obtained by replacing $\cos$ with $\sin$ in the fundamental modes.

Here are the basis functions for the expansion in the three regions.

```
In[4]:= k[m_] = Sqrt[E - (m Pi / L)^2];
 $\chi_L[m_, \{x_, y_ \}] := \text{Exp}[-I k[2 m - 1] x] \text{Cos}[(2 m - 1) \text{Pi} y / L];$ 
 $\chi_R[m_, \{x_, y_ \}] := \text{Exp}[+I k[2 m - 1] x] \text{Cos}[(2 m - 1) \text{Pi} y / L];$ 
 $\omega[m_, \{r_, \phi_ \}] := \text{Cos}[\mu \phi] \text{BesselJ}[\mu, \text{Sqrt}[E] r]$ 
```

On the left arc, we multiply with the first of the functions χ_L, χ_R, ω that is of the form $\frac{\cos}{\sin}(n(\varphi - \pi)/\phi \frac{\pi}{2})$ (they form an overcomplete system but cover the arc in a homogeneous manner). Later, we will use some of the Fourier coefficients (the cos-modes with odd n that are complete for the symmetric states we are looking for) and demand that they vanish.

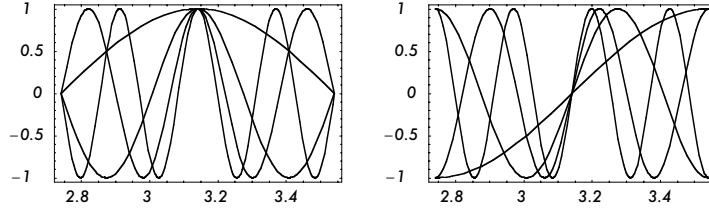
```
In[8]:= tLeft[sc_, n_,  $\phi_$ ] := sc[n ( $\phi - \text{Pi}$ ) /  $\phi \text{Pi} / 2$ ]
```

Here are these Fourier modes. Because the solution of the problem has to fulfill homogeneous Dirichlet boundary conditions along the boundaries of the center disk, we will later use the cos-modes that vanish at the endpoints of these arcs.

```

In[9]:= Block[{ $\phi = 0.4$ },
  Show[GraphicsArray[Block[{$DisplayFunction = Identity},
    Plot[Evaluate[Table[ $\mathfrak{tLeft}[\#, n, \phi]$ , {n, 1, 7, 2}]],
      { $\phi$ ,  $\text{Pi} - \phi$ ,  $\text{Pi} + \phi$ }, Frame  $\rightarrow$  True, Axes  $\rightarrow$  False] & /@ {Cos, Sin}]]]]

```



The function `cond[leftRight,  $\alpha$ ,  $\mu$ , {cs, n}]` represents the continuity of the μ th mode on the left or right arm arc of the α th derivative ($\alpha = 0$ or $\alpha = 1$ in the following) arising from the n th function of the Fourier series with trigonometric function *cs*. When the Fourier coefficients can be calculated in closed form symbolically, we will do so. Some Fourier coefficients have to be calculated through numerical integration. The function `condI` represents the inhomogeneous part of the linear equation arising from the incoming wave.

```

In[10]:= Do[{cs = {Cos, Sin}][[ $\beta$ ]];
  (* center disk-- symbolically calculatable *)
  cond[left,  $\alpha$ ,  $\mu$ ?NonNegative, {cs, n_}] = Integrate[
    (D[ $\omega[\mu, \{r, \phi\}]$ , {r,  $\alpha$ }] /. r  $\rightarrow$  R)  $\mathfrak{tLeft}[cs, n, \phi]$ , { $\phi$ ,  $\text{Pi} - \phi$ ,  $\text{Pi}$ ]];
  (* center disk  $\mu == 0$  *) cond[left,  $\alpha$ , 0, {cs, 0}] =
    Limit[cond[left,  $\alpha$ , 0, {cs, n}], n  $\rightarrow$  0];
  (* left arm-- numerically calculatable *)
  cond[left,  $\alpha$ , m_?((-1  $\geq$  #  $\geq$  -p) &), {cs_, n_}] := NIntegrate[
    Evaluate[(D[- $\chi_L[-m, \{r \text{ Cos}[\phi], r \text{ Sin}[\phi]\}]$ , {r,  $\alpha$ }] /. r  $\rightarrow$  R)
       $\mathfrak{tLeft}[cs, n, \phi]$ ], { $\phi$ ,  $\text{Pi} - \phi$ ,  $\text{Pi}$ ]];
  (* right arm coefficients are absent *)
  cond[left,  $\alpha$ , m_?((-p - 1  $\geq$  #  $\geq$  -p - q) &), {cs_, n_}] := 0;
  (* inhomogeneous part-- numerically calculatable *)
  condI[left,  $\alpha$ , {cs_, n_}] := NIntegrate[Evaluate[
    (D[ $\chi_R[1, \{r \text{ Cos}[\phi], r \text{ Sin}[\phi]\}]$ , {r,  $\alpha$ }] /. r  $\rightarrow$  R)  $\mathfrak{tLeft}[cs, n, \phi]$ ,
      { $\phi$ ,  $\text{Pi} - \phi$ ,  $\text{Pi}$ }], { $\alpha$ , 0, 1}, { $\beta$ , 1, 2}]

```

We proceed in a completely analogous manner for the right arc (where there is no inhomogeneous part).

```

In[11]:=  $\mathfrak{tRight}[sc_, n_, \phi_] := sc[n \phi / \phi \text{ Pi} / 2]$ 

```

```

In[12]:= Do[cs = {Cos, Sin}][[β]];
(* center disk *)
cond[right, α, μ_?NonNegative, {cs, n_}] = Integrate[
  (D[ω[μ, {r, φ}], {r, α}] /. r → R) tRight[cs, n, φ], {φ, 0, φ}];
(* center disk μ == 0 *) cond[right, α, 0, {cs, 0}] =
  Limit[cond[right, α, 0, {cs, n}], n → 0];
(* left arm coefficients are absent *)
cond[right, α_, m_?((-1 ≥ # ≥ -p) &), {cs_, n_}] := 0;
(* right arm *) cond[right, α_, m_?((-p - 1 ≥ # ≥ -p - q) &), {cs_, n_}] :=
  NIntegrate[
    Evaluate[(D[-χR[-m - p, {r Cos[φ], r Sin[φ]}], {r, α}] /. r → R)
      tRight[cs, n, φ]], {φ, 0, φ}], {α, 0, 1}, {β, 1, 2}]

```

The homogeneous Dirichlet boundary conditions along the horizontal boundaries of the left and right arms are fulfilled automatically through the form of the ansatz. To fulfill the Dirichlet boundary conditions along the upper and lower arcs bounding the center disk, we again multiply with functions from an overcomplete set of Fourier-like functions. Later, we will use the sin-modes for all n , which form a complete system along the upper and lower arcs of the center disk.

```

In[13]:= tCenter[sc_, n_, φ_] := sc[n (φ - φ) / (Pi - 2 φ) Pi]

In[14]:= Do[cs = {Cos, Sin}][[β]];
(* center disk *)
cond[center, α, μ_?NonNegative, {cs, n_}] = Integrate[
  (D[ω[μ, {r, φ}], {r, α}] /. r → R) tCenter[cs, n, φ], {φ, φ, Pi - φ}];
(* center disk μ == 0 *) cond[center, α, 0, {cs, 0}] =
  Limit[cond[center, α, 0, {cs, n}], n → 0];
(* left and right arms *) cond[center, α, μ_?Negative, n_] := 0,
  {α, 0, 1}, {β, 1, 2}]

```

■ Matrix Construction

We have to determine $w = p + (o + 1) + q$ coefficients c_k . To find them, we will have to use more equations than variables. (If we take the same number of equations as unknown coefficients, we would get a unique solution that makes the specified Fourier coefficients vanish but which could oscillate dramatically. See the following explicit numerical results.) Here, we will use d Fourier modes along the circular arcs and e modes along the longer Dirichlet parts of the center disk.

As mentioned earlier, we will use the cos-modes along the arcs joining the center disk with the left and right arm and the sin-modes along the Dirichlet part of the center disk. We could, of course, also use some of the sin modes along the joining arcs.

The function `makeLHSAndRHS[prec, {d, e}]` will construct the coefficient matrix and the right-hand side of the system of linear equations for the c_k with $4d + e$ equations.

```

In[15]:= makeLHSAndRHS[prec_, {d_, e_}] := Module[{lhs, rhs},
  (* set options for NIntegrate to obtain precise matrix elements *)
  SetOptions[NIntegrate, WorkingPrecision -> prec + 20,
    PrecisionGoal -> prec, MinRecursion -> 6, Method -> GaussKronrod];
  (* coefficient matrix *)
  (* use cos Fourier states for function and derivative
  along left-center, center-right,
  and sin Fourier states for center Dirichlet region boundaries *)
  lhs = Join[Table[cond[left, 0, m, {Cos, n}], {n, 1, 2 (d - 1) + 1, 2},
    {m, -p - q, o}], Table[cond[left, 1, m, {Cos, n}],
    {n, 1, 2 (d - 1) + 1, 2}, {m, -p - q, o}], Table[
    cond[right, 0, m, {Cos, n}], {n, 1, 2 (d - 1) + 1, 2}, {m, -p - q, o}],
    Table[cond[right, 1, m, {Cos, n}], {n, 1, 2 (d - 1) + 1, 2},
    {m, -p - q, o}], Table[cond[center, 0, m, {Sin, n}],
    {n, 1, e}, {m, -p - q, o}] // N[#, prec {1, 1}] &;
  (* right-hand side inhomogeneous vector;
  nontrivial values only for incoming wave in the left arm *) rhs =
  Join[Table[condI[left, 0, {Cos, n}], {n, 1, 2 (d - 1) + 1, 2}],
    Table[condI[left, 1, {Cos, n}], {n, 1, 2 (d - 1) + 1, 2}],
    Table[0, {n, 1, 2 (d - 1) + 1, 2}],
    Table[0, {n, 1, 2 (d - 1) + 1, 2}], Table[0, {n, 1, e}]];
  (* return coefficient matrix and right-hand side vector *) {lhs, rhs}]

```

■ Function and Gradient Values

A solution for the coefficients c_k gives us access to the solution $\psi(x, y)$ in the three regions. Using the transformation rules for changing from derivatives in Cartesian to derivatives in polar coordinates

$$\begin{aligned}\frac{\partial}{\partial x} &= \frac{\partial \varphi}{\partial x} \frac{\partial}{\partial \varphi} + \frac{\partial r}{\partial x} \frac{\partial}{\partial r} = -\frac{\sin(\varphi)}{r} \frac{\partial}{\partial \varphi} + \cos(\varphi) \frac{\partial}{\partial r} \\ \frac{\partial}{\partial y} &= \frac{\partial \varphi}{\partial y} \frac{\partial}{\partial \varphi} + \frac{\partial r}{\partial y} \frac{\partial}{\partial r} = +\frac{\cos(\varphi)}{r} \frac{\partial}{\partial \varphi} + \sin(\varphi) \frac{\partial}{\partial r},\end{aligned}$$

we can implement a function, `makeWaveFunctionAndGradientDefinition`, that sets up definitions for the solution $\Psi = \psi(x, y)$ and the gradient vector $\text{nabla}\Psi = \{\partial_x \psi(x, y), \partial_y \psi(x, y)\}$. (We will use the gradient vector to calculate currents associated with the complex-valued wave function $\psi(x, y)$.)

```

In[16]:= (* derivatives of solutions in the left and right arms *)
χLDx[m_, {x_, y_}] :=
  -I k [2 m - 1] Exp[-I k [2 m - 1] x] Cos[(2 m - 1) Pi y / L];
χLDy[m_, {x_, y_}] := -(2 m - 1) Pi / L
  Exp[-I k [2 m - 1] x] Sin[(2 m - 1) Pi y / L];
χRDx[m_, {x_, y_}] := I k [2 m - 1] Exp[+I k [2 m - 1] x]
  Cos[(2 m - 1) Pi y / L];
χRDy[m_, {x_, y_}] := -(2 m - 1) Pi / L
  Exp[+I k [2 m - 1] x] Sin[(2 m - 1) Pi y / L];

```

```

In[20]:= makeWaveFunctionAndGradientDefinition[
  {Ψ_, nablāΨ_}, sol_, prec_, opts___] :=
  ((* export the following function for global access *)Clear[
    "ψDot*", "ψLeft*", "ψRight*", "nablāψDot*",
    "nablāψLeft*", "nablāψRight*"]);
Module[{αs, γs, βs, ψLeft, ψRight, besselValues, cosValues,
  (*dataDot,*)dataIpoDot, ψDot, nablāψLeft, nablāψRight,
  besselDValues, cosDValues, dataDxDot, dataDyDot,
  dataDxIpoDot, dataDyIpoDot, nablāψDot, ppDot = 120, ε = 10^-6},
  (* extract coefficients for left arm, right arm,
    and center disk *)αs = Reverse@Take[sol, {q + 1, q + p}];
  γs = Reverse@Take[sol, q];
  βs = Take[sol, -o - 1];
  (* form solutions in left and right arms *)ψLeft[x_, y_] :=
    χR[1, {x, y}] + Table[χL[m, {x, y}], {m, p}].αs;
  ψRight[x_, y_] := Table[χR[m, {x, y}], {m, q}].γs;
  (* form solution in center disk;
    minimize number of relatively time-
    consuming Bessel function evaluations *)besselValues = Table[
    {r, Table[N[BesselJ[μ, Sqrt[ε] r], prec], {μ, 0, Length[βs] - 1}]},
    {r, ε, R, (R - ε) / ppDot}];
  cosValues = Table[{φ, Table[N[Cos[μ φ], prec], {μ, 0, Length[βs] - 1}]},
    {φ, 0, Pi, Pi / ppDot}];
  dataDot = Table[{besselValues[[i, 1]], cosValues[[j, 1]],
    (besselValues[[i, 2]] cosValues[[j, 2]]).βs}, {i, Length[
    besselValues]}, {j, Length[cosValues]}] // Flatten[#, 1] &;
  (* interpolate values *)dataIpoDot = Interpolation[
    dataDot, InterpolationOrder → 5];
  (* Cartesian coordinate system values inside center disk *)
  ψDot[x_, y_] :=
    dataIpoDot[Sqrt[x^2 + y^2], If[x == y == 0., 0, ArcTan[x, y]]];
  (* globally visible solution *)Ψ[x_, y_] :=
    Which[Re[N[x^2 + y^2]] ≤ R^2, ψDot[x, y],
    Re[N[x]] ≤ N[-R Cos[φ]], ψLeft[x, y], Re[N[x]] ≥ N[R Cos[φ]],
    ψRight[x, y], True, ψDot[x, y]];
  If[Not[TrueQ[MakeGradientDefinition /. {opts}]],
    (* form solutions gradients in left and right arms *)nablāψLeft[x_,
      y_] := {χRDx[1, {x, y}] + Table[χLDx[m, {x, y}], {m, p}].αs,
      χRDy[1, {x, y}] + Table[χLDy[m, {x, y}], {m, p}].αs};
    nablāψRight[x_, y_] := {Table[χRDx[m, {x, y}], {m, q}].γs,
      Table[χRDy[m, {x, y}], {m, q}].γs};
  (* form solution in center disk;
    minimize number of relatively time-
    consuming Bessel function evaluations *)besselDValues =
    Table[{r, Table[N[Sqrt[ε] / 2 (BesselJ[μ - 1, r Sqrt[ε]] -
      BesselJ[μ + 1, r Sqrt[ε]])], prec],
      {μ, 0, Length[βs] - 1]}, {r, ε, R, (R - ε) / ppDot}];
  cosDValues = Table[{φ, Table[N[-μ Sin[μ φ], prec],
    {μ, 0, Length[βs] - 1}]}, {φ, 0, Pi, Pi / ppDot}];
  (* make list of data of the form {r, φ, ψ[r, φ] *)
    dataDxDot = Table[{besselValues[[i, 1]], cosValues[[j, 1]],

```

```

      (Cos[cosValues[[j, 1]]] besselDValues[[i, 2]] cosValues[[j,
        2]] + besselValues[[i, 2]] (-Sin[cosValues[[j, 1]]] /
        besselValues[[i, 1]]) * cosDValues[[j, 2]]) .  $\beta$ s},
    {i, Length[besselValues]}, {j, Length[cosValues]}] //
  Flatten[#, 1] &;
dataDyDot = Table[{besselValues[[i, 1]], cosValues[[j, 1]],
  (Sin[cosValues[[j, 1]]] besselDValues[[i, 2]] cosValues[[j,
    2]] + besselValues[[i, 2]] (+Cos[cosValues[[j, 1]]] /
    besselValues[[i, 1]]) * cosDValues[[j, 2]]) .  $\beta$ s},
  {i, Length[besselValues]}, {j, Length[cosValues]}] //
  Flatten[#, 1] &;
(* interpolate values of gradient components *){dataDxIpoDot,
  dataDyIpoDot} = Interpolation /@ {dataDxDot, dataDyDot};
(* Cartesian coordinate system values inside center disk *)
nablaψDot[x_, y_] := {dataDxIpoDot[##], dataDyIpoDot[##]} &[
  Sqrt[x^2 + y^2], If[x == y == 0., 0, ArcTan[x, y]]];
(* globally visible solution *)
nablaψ[x_, y_] := Which[Re[N[x^2 + y^2]] ≤ R^2, nablaψDot[x, y],
  Re[N[x]] ≤ N[-R Cos[φ]], nablaψLeft[x, y], Re[N[x]] ≥ N[R Cos[φ]],
  nablaψRight[x, y], True, nablaψDot[x, y]]];

```

We also define a function, `insideQ`, that returns `True` if the point $\{x, y\}$ is inside the left arm, right arm, or center disk.

```

In[21]:= insideQ[x_, y_] := (x^2 + y^2 < R^2) ||
  x < -R Cos[φ] && Abs[y] < L/2 || x > +R Cos[φ] && Abs[y] < L/2

insideQ[{x_, y_}] := insideQ[x, y]

```

■ Measuring the Quality of the Solution

Before visualizing the eigenfunctions, we should determine values of the parameters p , q , o , d , e , and working precisions appropriate for the energy values under consideration (say $\mathcal{E} \approx 1 \dots 5$).

```

In[23]:= (* set global parameter values *)
E = 3; R = 10; L = 5;
Clear[p, q, o, b, e, prec];

```

The function `solutionQuality` carries out numerical integrations along the domain boundaries to quantify the quality of the solutions.

```

In[24]:= solutionQuality[{Ψ1_, nablāΨ1_}] :=
  With[{ε = 10^-6, opts = Sequence[PrecisionGoal → 3]},
    (* deviations from Dirichlet boundary conditions *)
    (NIntegrate[Abs[Ψ1[R Cos[φ], R Sin[φ]]]^2, {φ, 0, Pi - φ}, opts] +
      (* deviations from continuity along arcs connecting center disk
        and arms *) NIntegrate[Abs[Ψ1[(R - ε) Cos[φ], (R - ε) Sin[φ]] -
        Ψ1[(R + ε) Cos[φ], (R + ε) Sin[φ]]]^2, {φ, 0, φ}, opts] +
      NIntegrate[Abs[Ψ1[(R - ε) Cos[φ], (R - ε) Sin[φ]] -
        Ψ1[(R + ε) Cos[φ], (R + ε) Sin[φ]]]^2, {φ, Pi - φ, Pi}, opts] +
      (* deviations from differentiability along arcs connecting center
        disk and arms *) NIntegrate[Abs[(*) use directional derivatives *)
        (nablāΨ1[(R - ε) Cos[φ], (R - ε) Sin[φ]] - nablāΨ1[(R + ε) Cos[φ],
          (R + ε) Sin[φ]]).{Cos[φ], Sin[φ]}]^2, {φ, 0, φ}, opts] +
      NIntegrate[Abs[(nablāΨ1[(R - ε) Cos[φ], (R - ε) Sin[φ]] -
        nablāΨ1[(R + ε) Cos[φ], (R + ε) Sin[φ]]).{Cos[φ], Sin[φ]}]^2,
        {φ, Pi - φ, Pi}, opts]) / (* boundary length *) (R (2 Pi + 4 φ))])

```

The function `multiParameterSolutionQuality` constructs and solves the matrices and then measures the quality of the solutions for a set of parameters. The actual solution of the linear system of overdetermined equations is achieved using the function `PseudoInverse`. The function `multiParameterSolutionQuality` returns lists of the form:

$$\{\text{parameterValue}, \{\text{linearEquationResidue}, \text{integratedBoundaryValuesDifferences}, \text{reflectionAndTransmissionCoefficientMinus1}\}\}$$

The value `integratedBoundaryValuesDifferences` is the result of applying `solutionQuality` to the solution and `reflectionAndTransmissionCoefficientMinus1` is $||\mathcal{R}|^2 + |\mathcal{T}|^2 - 1|$. The last quantity is expected to vanish for the exact solution.

```

In[25]:= multiParameterSolutionQuality[{{pp_, qq_, oo_}, prec_, {dd_, ee_}},
      {iVar_, iVar1_, iVar2_, iVarStep_}, opts___] :=
Module[{lhs, rhs, psi, sol}, Table[
  (* set parameter values *) {p, o, q} = {pp, oo, qq}; {d, e} = {dd, ee};
  (* construct matrices *) {lhs, rhs} = makeLHSAndRHS[prec, {d, e}];
  (* solve for expansion coefficients *) psi = PseudoInverse[lhs];
  (* analyze vector solution *) {iVar,
    If[MatrixQ[psi], (* form solution *) sol = psi.rhs;
  (* form solution and gradient *) makeWaveFunctionAndGradientDefinition[
    {Ψ1, nablaΨ1}, sol, prec];
  (* potentially show scaled solution *)
  If[Make3DPlot /. {opts},
    make3DPlot[ArcTan[Abs[#]^2/4] / (Pi/2) &, Ψ1]];
  (* measure solution quality *)
  Off[NIntegrate::ploss];
  N[{(* quality of solution of the linear system of equations *)
    Norm[lhs.sol - rhs] / Length[rhs],
    (* quality of boundary condition fulfillment of the
      eigenfunction *) solutionQuality[{Ψ1, nablaΨ1}],
    (* sum of absolute value of reflection and transmission
      coefficient *) Abs[sol[[q]]]^2 + Abs[sol[[p+q]]]^2}, 3],
    {}], {iVar, iVar1, iVar2, iVarStep}]

```

We now carry out some experiments to determine appropriate parameters. We start by determining the number of Fourier modes versus the number of basis states. The following inputs show that 5 to 10 more Fourier modes give good results.

```

In[26]:= (* change number of Fourier modes in the arms *)
multiParameterSolutionQuality[{{4, 4, 30}, 50, {d, 30}}, {d, 1, 6, 1}]

Out[26]:= {{1, {0, 2.36, 1.00}}, {2, {4.4 × 10-25, 861., 1.0}},
  {3, {0.0000564, 1.20, 0.999}}, {4, {0.000456, 0.000220, 0.967}},
  {5, {0.000477, 0.000209, 0.957}}, {6, {0.000494, 0.000204, 0.947}}}

In[27]:= (* change number of Fourier modes in the center disk *)
multiParameterSolutionQuality[
  {{4, 4, 30}, 50, {4, e}}, {e, 10, 50, 10}]

Out[27]:= {{10, {0, 4.34, 0.945}}, {20, {0, 30.2, 1.00}},
  {30, {0.000456, 0.000220, 0.967}},
  {40, {0.000522, 0.000234, 0.939}}, {50, {0.000497, 0.000244, 0.925}}}

```

We continue by determining the number of basis states needed in the arms and in the center disk. Obviously, the more basis states we take into account, the better the resulting solution. But after considering four states in the arms and about 30 in the center disk, the solution gets better only slowly. For mainly graphical purposes, we obtain a satisfactory quantitative solution.

```

In[28]:= (* change number of basis states in the arms;
          use about 10 more Fourier modes than coefficients *)
multiParameterSolutionQuality[
  {{pq, pq, 30}, 50, {3 + pq, 30}}, {pq, 1, 5, 1}]

Out[28]= {{1, {0.00195, 0.000861, 0.526}},
          {2, {0.00110, 0.000434, 0.790}}, {3, {0.000678, 0.000281, 0.902}},
          {4, {0.000498, 0.000200, 0.938}}, {5, {0.000493, 0.000198, 0.931}}}

In[29]:= (* change number of basis states in the center disk;
          use about 10 more Fourier modes than coefficients *)
multiParameterSolutionQuality[
  {{4, 4, o}, 50, {6, o + 2}}, {o, 10, 50, 10}]

Out[29]= {{10, {0.00215, 0.000576, 0.646}},
          {20, {0.00127, 0.000340, 0.764}}, {30, {0.000512, 0.000206, 0.940}},
          {40, {0.000348, 0.000219, 0.961}}, {50, {0.000329, 0.000217, 0.954}}}

```

We end by finding the precision needed for the calculations. About 40 digits seems appropriate for $p = q \approx 4$ and $o = 30$. (High-precision results with a smaller precision can give wrong results because matrix arithmetic is carried out in fixed precision with precision estimation *a posteriori*.)

```

In[30]:= (* change working precision;
          use about 10 more Fourier modes than coefficients *)
multiParameterSolutionQuality[
  {{4, 4, 30}, prec, {4, 30}}, {prec, 20, 60, 10}]

Out[30]= {{20, {0.00519, 0.00619, 0}},
          {30, {0.00113, 0.000449, 0.811}}, {40, {0.000456, 0.000220, 0.967}},
          {50, {0.000456, 0.000220, 0.967}}, {60, {0.000456, 0.000220, 0.967}}}

```

Here is a high-quality solution where the reflection-transmission coefficient identity is fulfilled to 0.3%.

```

In[31]:= (* change working precision;
          use about 10 more Fourier modes than coefficients *)
multiParameterSolutionQuality[
  {{5, 5, 60}, prec, {5, 60}}, {prec, 100, 100, 10}]

Out[31]= {{100, {0.0000798, 0.000266, 0.997}}}

```

The last inputs show that for energies $\mathcal{E} \approx 10^0$, the following are appropriate: $p = q = 4$, $o = 40$, $d \approx p$, $e \approx o$ and a working precision of about o . We definitely need more Fourier modes than basis states: about 10 more seems a good choice. With these parameters we obtain an average function value of $|\psi(x, y)| \approx 10^{-3}$ along the Dirichlet part of the center disk. The quality of the reflection and transmission coefficients decreases slightly with an increasing number of Fourier modes (because the error gets more homogeneously distributed among the $p + q = o + 1$ expansion coefficients) and typically is in the order of a few percent or less.

■ Example Solution

For some further visualizations, we will use the following solution.

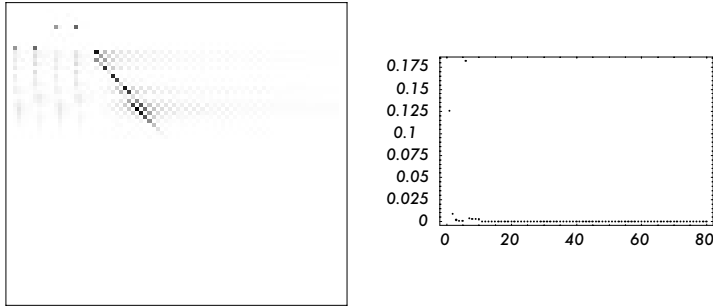
```
In[32]:= (* geometry parameters *)
R = 10; L = 5; E = 5 / 2;
(* appropriate parameter values for the numerical solution *)
{p, o, q} = {5, 60, 5};
{d, e} = {5, 60};
prec = 80;

{lhs, rhs} = makeLHSAndRHS[prec, {d, e}];

In[33]:= Dimensions /@ {lhs, rhs}
Out[33]:= {{80, 71}, {80}}
```

This visualizes the left-hand side matrix and the right-hand side vector.

```
In[34]:= Show[GraphicsArray[Block[{$DisplayFunction = Identity},
(* matrix and right-hand side vector *)
{ArrayPlot[Abs[lhs] // Transpose, PlotRange -> All],
ListPlot[Abs[rhs], Frame -> True, Axes -> False]}]]]
```



```
In[35]:= psi = PseudoInverse[lhs];
sol = If[MatrixQ[psi], psi.rhs, $Failed];
```

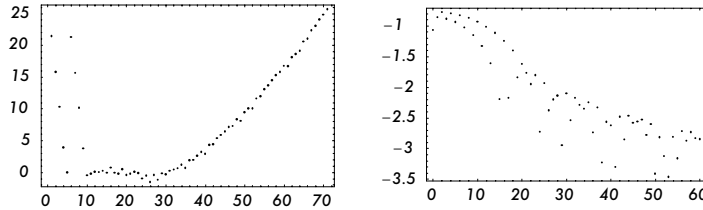
The left graphic is the plot of the solution vector. It clearly shows the two initial sequences belonging to the expansion coefficients in the left and right arms. While the expansion coefficients in the center disk are increasing in absolute value, their influence is decreasing because the factor $\mathcal{J}_m(\sqrt{E} r)$ decreases quickly with m for $r \approx R$. From the absolute size of the coefficients, we also see that the use of high-precision numbers with ≥ 30 digits is appropriate.

The right graphic shows the product of the expansion coefficients with the ansatz functions $\mathcal{J}_m(\sqrt{E} r)$. We see that the influence of the largest modes in the function value at the center disk boundary is approximately 10^{-3} .

```

In[37]:= Show[GraphicsArray[Block[{$DisplayFunction = Identity,
  opts = Sequence[Frame → True, Axes → False]],
  {(* solution vector *)ListPlot[Log[10, Abs[sol]], opts], (* product
    of solution vector and basis function in the center disk *)
    ListPlot[MapIndexed[{#2[[1]] - 1,
      Log[10, Abs[#1 BesselJ[#2[[1]] - 1, Sqrt[E] R}]]] &,
      Take[sol, -(o + 1)]], opts]}]]]

```



We define the corresponding wave functions and their gradients.

```

In[38]:= makeWaveFunctionAndGradientDefinition[{ψ1, nablaψ1}, sol, prec]

```

■ 3D Plots of the Scattering States

Next, we want to visualize the solution $\psi(x, y)$. The following auxiliary functions construct polygons adapted to the shape of the regions for a given solution Ψ . Just like the joining curves, we use the arcs of the boundary of the center disk that touch the left and right arms.

```

In[39]:= makePolygons[points_] := (* make polygons from array of points *)
  Table[Polygon[{points[[i, j]],
    points[[i + 1, j]], points[[i + 1, j + 1]], points[[i, j + 1]]}],
    {i, Length[points] - 1}, {j, Length[points[[1]]] - 1}];

In[40]:= leftPolygons[Ψ_, F_] := (* polygons in the left arm *)
  makePolygons[Table[{x, y, F[Ψ[x, y]]}, {y, 0, L/2, L/2/ppArm},
    {x, -2 R, -R Cos[ArcSin[y/R]], R (2 - Cos[ArcSin[y/R]])/ppArm}]]

In[41]:= rightPolygons[Ψ_, F_] := (* polygons in the right arm *)
  makePolygons[Table[{x, y, F[Ψ[x, y]]}, {y, 0, L/2, L/2/ppArm},
    {x, R Cos[ArcSin[y/R]], 2 R, R (2 - Cos[ArcSin[y/R]])/ppArm}]]

In[42]:= dotPolygons[Ψ_, F_] := (* polygons in the center disk *)
  makePolygons[Table[{r Cos[φ], r Sin[φ], F[Ψ[r Cos[φ], r Sin[φ]]]},
    {r, 0, R, R/ppDot}, {φ, 0, Pi, Pi/ppDot}]]

```

The function `make3DPlot` generates a 3D plot of $\text{reim}(\psi(x, y))$. It makes use of the symmetry about the x -axis.

```

In[43]:= make3DPlot[reim_,  $\Psi$ _, opts___] :=
  Block[{polysL, polysR, polysD, ppDot = 120, ppArm = 60},
    (* generate polygons in upper half-plane *)
    polysL = leftPolygons[ $\Psi$ , reim];
    polysD = dotPolygons[ $\Psi$ , reim];
    polysR = rightPolygons[ $\Psi$ , reim];
    (* show polygons in lower and upper half-planes *)
    Show[Graphics3D[{SurfaceColor[Hue[0.12], Hue[0.72], 2.2],
      polysL, polysD, polysR}] /.
      p_Polygon  $\rightarrow$  {EdgeForm[], p, Polygon[{1, -1, 1} # & /@ p[[1]]}],
      opts, ViewPoint  $\rightarrow$  {-0.3, -3, 2.2}, Axes  $\rightarrow$  False,
      BoxRatios  $\rightarrow$  {4, 2, 0.6},
      PlotRange  $\rightarrow$  {{-2, 2} R, {-1, 1} R, {0, Pi/2}}]]

```

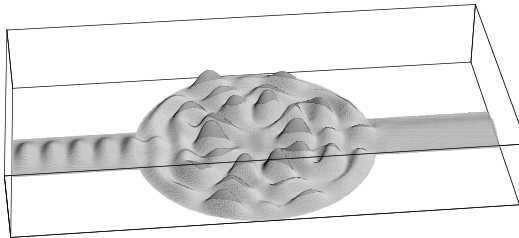
Here is a 3D plot of the probability density. Visually, the Dirichlet boundary condition along the center disk and the continuity to the arms is fulfilled to a high degree. The periodic structure in the left arm comes from the interference of the incoming part of the wave with its partial reflection back into the right arm.

```

In[44]:= (* turn off message from iterator numericalization *)
Off[Table::itflrw];
(* turn off message from interpolating
  function calls outside their domain *)
Off[InterpolatingFunction::dmval];

In[46]:= make3DPlot[ArcTan[Abs[#]^2/4]/(Pi/2) &,  $\Psi$ 1]

```



■ Contour Plots

To visualize the nodal lines $\text{Re}(\psi(x, y)) = 0$ and $\text{Im}(\psi(x, y)) = 0$ [3], we use a contour plot. Because the wave function can be defined outside the quantum dot and the arms through interpolation, we can use the built-in function `ContourPlot` on a rectangular region containing Ω and then cover the parts not belonging to Ω with white polygons. The function `makeReImAbsContourPlot` shows the curves $\text{Re}(\psi(x, y)) = 0$ in red and the curves $\text{Im}(\psi(x, y)) = 0$ in blue. The underlying gray level contour plot shows the probability density.

```

In[47]:= Off[InterpolatingFunction::dmval];

```

```

In[48]:= makeReImAbsContourPlot[ψ_] :=
Module[{ε = 10^-3, cover, border, upperHalf, lowerHalf},
(* cover outside regions *)
cover = {White, Polygon /@ ({#, {-1, 1} # & /@ #} &[
R {-2 - ε, L / 2 / R}, Sequence @@ Table[{Cos[φ], Sin[φ]}, {φ, Pi - φ,
Pi / 2, -(Pi / 2 - φ) / 60}], {0, 1 + ε}, {-2 - ε, 1 + ε}]]});
(* border of the original domain *)border =
{Thickness[0.006], RGBColor[0, 1, 0],
Line /@ ({#, {-1, 1} # & /@ #} &[R {-2, L / 2 / R}, Sequence @@ Table[
{Cos[φ], Sin[φ]}, {φ, Pi - φ, Pi / 2, -(Pi / 2 - φ) / 60}]]]);
(* nodal lines and scaled density in the upper-half part *)
upperHalf = Show[{(* scaled density *)ContourPlot[
ArcTan[Abs[#]^2 / 12] / (Pi / 2) &@ψ[x, y], {x, -2 R, 2 R},
{y, 0, R}, PlotPoints → {320, 160}, Contours → 30,
ContourShading → True, ContourLines → False,
DisplayFunction → Identity, ColorFunction → GrayLevel],
(* nodal lines *)ContourPlot[#1[ψ[x, y]], {x, -2 R, 2 R}, {y, 0, R},
PlotPoints → {320, 160}, Contours → {0}, ContourShading →
False, ContourStyle → {{Thickness[0.004], #2}},
DisplayFunction → Identity] &@@@{{Re, Red}, {Im, Blue}},
Graphics[{cover, border}], PlotRange → All];
(* generate lower-half part by reflection *)lowerHalf =
upperHalf /. (pl : (Polygon | Line)) [l_] => pl[{1, -1} # & /@ l];
(* show lower- and upper-half parts *)Show[
{upperHalf, lowerHalf}, DisplayFunction → $DisplayFunction,
Frame → False, AspectRatio → Automatic]]

```

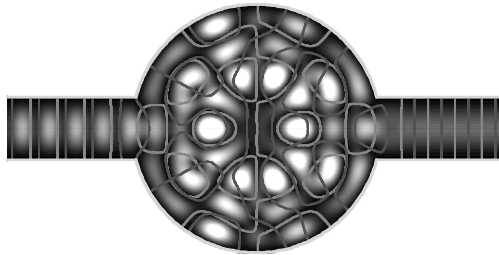
Here is this contour plot for our example function.

```

In[49]:= (* avoid messages from the use of ψ outside the original domain *)
Off[InterpolatingFunction::dmval];

In[50]:= makeReImAbsContourPlot[ψ1]

```



■ Current Plots

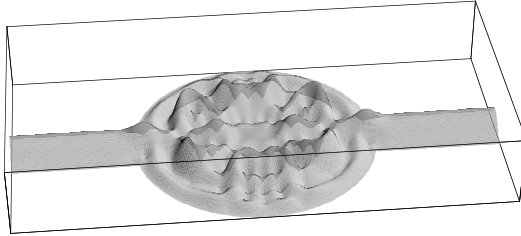
Because we are considering a scattering problem, not only the quantum mechanical density but also the quantum mechanical current is of interest. Because the function `makeReImAbsContourPlot` sets up definitions for the density and

gradient, we can define the quantum mechanical current j easily through the following formula.

```
In[51]:= j[x_, y_] := Re[Conjugate[Ψ1[x, y]] nabraΨ1[x, y] / I]
```

Here is a 3D plot of the norm of the current density for our example solution. We reuse the function `make3DPlot` to conveniently adapt to the shape of the region Ω .

```
In[52]:= make3DPlot[Identity, Function[{x, y}, Norm[j[x, Sqrt[y^2]]] / 3]]
```

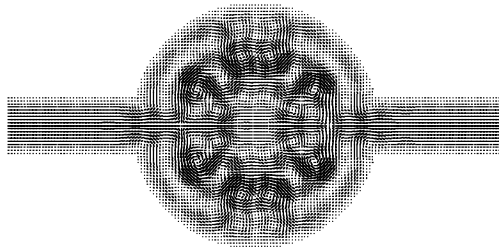


We continue with a simple vector field visualization of the current.

```
In[53]:= (* scaled line segment representing current direction and magnitude *)
jVector[{x_, y_}] :=
  Line[{x, y}, {x, y} + 0.5 ArcTan[j[x, y]] / (Pi / 2)]
```

We see the characteristic vortices appearing in such structures [7].

```
In[54]:= Off[ArcTan::indet]; Off[Graphics::gptn];
(* name for later use *) jGraphic =
  Show[Graphics[#, # /. Line[l_] := Line[{1, -1} # & /@ l]] & @
    Table[If[insideQ[x, y], jVector[{x, y}], {}],
      {x, -2 R, 2 R, 4 R / 160}, {y, 0, R, R / 40}],
  Frame → False, AspectRatio → Automatic]
```



■ Make Streamlines

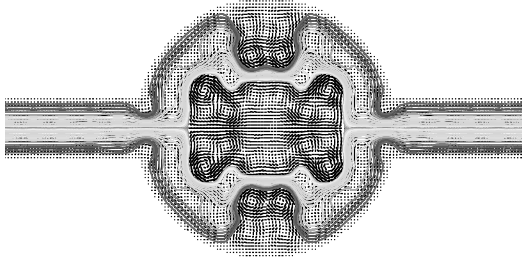
Calculation of streamlines is a natural operation to carry out on a vector field. We start from the left arm.

```
In[55]:= (* current direction *) jN[{x_?NumberQ, y_?NumberQ}] :=
  # / Sqrt[#. #] & @ j[x, Sqrt[y^2]]
```

```

In[56]:= Off[NDSolve::"mxst"];
With[{T = 80, pps = 16},
  Show[Flatten[(* above direction field of the current *)
    jGraphic, {#, # /. Line[1_] := Line[{1, -1} # & /@1]} &@
    (* streamlines in the upper-half part *)Table[
      (* starting positions; use larger density at symmetry axis *)
      X0 = {-2 R, y^2 3/4 L/2};
    (* solve differential equations *)nds =
      NDSolve[{X'[t] == jN[X[t]], X[0] == X0}, X, {t, -T, T},
        PrecisionGoal -> 8, AccuracyGoal -> 8, MaxSteps -> 10^5];
    (* plot streamlines *)ParametricPlot[Evaluate[X[t] /. nds[[1]],
      Evaluate[Flatten[{t, nds[[1, 1, 2, 1]]}], Frame -> True, Axes ->
        False, AspectRatio -> Automatic, DisplayFunction -> Identity,
        PlotStyle -> {{Hue[0.74 y]}}, PlotRange -> {{-2 R, 2 R}, {0, R}},
        PlotPoints -> 200], {y, 0, 1, 1/pps}]],
    PlotRange -> {{-2 R, 2 R}, {-R, R}},
    DisplayFunction ->
      $DisplayFunction]]

```



■ Energy-Dependent Transmission

As a function of the incoming energy, the absolute value of the transmission coefficient $\mathcal{T}$ is of special interest. When the widths L of the left and right arms are small compared with the diameter of the center disk R , a resonance is expected if the energy of the incoming wave is near an eigenvalue of the center disk with Dirichlet boundary conditions. Here we will consider the energy range $1 \leq \mathcal{E} \leq 2$. We start by calculating the lowest eigenstates of the center disk with Dirichlet boundary conditions. The exact eigenvalues for a disk with homogeneous Dirichlet boundary conditions are $\mathcal{E}_{jk} = \mu_{jk}^2 / R^2$, where μ_{jk} is the k th zero of the Bessel function $J_j(z)$.

```

In[58]:= Needs["NumericalMath`BesselZeros`"]

```

```
In[59]:= (* eigenvalues of the center dot with Dirichlet boundary conditions *)
dotEigenvalues = Select[Flatten[
  Table[MapIndexed[{{v, #2[[1]]}, #} &, BesselJZeros[v, 30]^2/R^2],
    {v, 0, 30}], 1], {1 ≤ #[-1] ≤ 2} &]
```

```
Out[59]= {{0, 4}, 1.3904}, {{1, 3}, 1.03499},
  {{1, 4}, 1.77521}, {{2, 3}, 1.35021}, {{3, 3}, 1.69395},
  {{4, 2}, 1.22428}, {{5, 2}, 1.52241}, {{6, 2}, 1.84669},
  {{7, 1}, 1.22908}, {{8, 1}, 1.49453}, {{9, 1}, 1.78337}
```

The function norm calculates the norm of these states.

```
In[60]:= norm[{{v_, _}, E_}] := Sqrt[If[v == 0, 2 Pi, Pi]
  (-R^2/2 BesselJ[v - 1, R Sqrt[E]] BesselJ[v + 1, R Sqrt[E]])]
```

We use the function overlap to calculate the overlap integral between the eigenstates of the center disk and the scattering states within it. We form the overlap integral with the expansion functions symbolically. Due to the orthogonality of the angular parts of the eigenfunctions and the scattering state expansion functions, we have only $o + 1$ terms to consider.

```
In[61]:= Clear[L, R, E]
```

```
overlapFactor[E_, {{v_, n_}, Eev_}] =
  If[v == 0, 2 Pi, Pi] * 1/norm[{{v, n}, Eev}] *
  Integrate[BesselJ[v, Sqrt[E] r] BesselJ[v, Sqrt[Eev] r] r,
    {r, 0, R}] // Simplify[#, R > 0] &
```

```
Out[62]= (Sqrt[2] (Sqrt[Eev] BesselJ[-1 + v, R Sqrt[Eev]] BesselJ[v, R Sqrt[E]] -
  Sqrt[E] BesselJ[-1 + v, R Sqrt[E]] BesselJ[v, R Sqrt[Eev]]))
  If[v == 0, 2 Pi, Pi] / ((E - Eev)
  Sqrt[-BesselJ[-1 + v, R Sqrt[Eev]] BesselJ[1 + v, R Sqrt[Eev]] If[v == 0, 2 Pi, Pi])
```

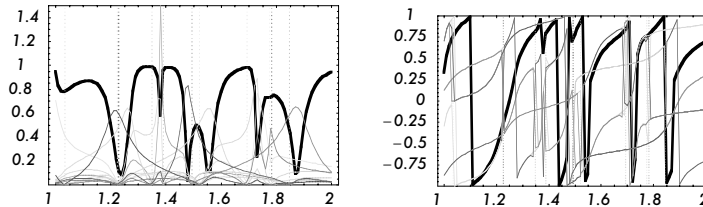
```
In[63]:= overlap[sol_, E_, {{v_, n_}, Eev_}] :=
  sol[[p + q + 1 + v]] overlapFactor[E, {{v, n}, Eev}]
```

Now, we calculate the transmission data. To quickly get a rough idea of its energy dependence, we use a smaller number of basis states.

```
In[64]:= R = 10; L = 5;
prec = 40;
(* use smaller parameter values for a qualitative dependence on E *)
{p, o, q} = {3, 30, 3}; {d, e} = {4, 32};
transmissionData = Table[(* construct matrices *)
  {lhs, rhs} = makeLHSAndRHS[prec, {d, e}];
  (* construct solution vector *) psi = PseudoInverse[lhs];
  sol = If[MatrixQ[psi], psi.rhs, $Failed];
  (* extract transmission
    coefficient and form overlap matrix elements *){E,
    T = sol[[q]], Table[overlap[sol, E, dotEigenvalues[[k]]],
      {k, Length[dotEigenvalues]}]}, {E, 1, 2, 1/100}];
```

Here are the resulting transmission data and overlap integrals. The left graphic shows the transmission with the scaled absolute values of the overlap integrals. The right graphic shows the transmission with the argument of the transmission coefficient and the arguments of the overlap integrals. We clearly see a strong correlation between the transmission coefficient of the incoming wave in resonance with the eigenstates of the center disk with Dirichlet boundary conditions.

```
In[65]:= Show[GraphicsArray[
  Function[{f1, f2, pr}, Show[Graphics[(* transmission data *)
    {Thickness[0.01], Line[{{#[[1]], f1[#[[2]]]}] & /@
      transmissionData}], {PointSize[0.004],
    (* center dot eigenvalues as vertical lines *)MapIndexed[
      {Hue[0.8 #2[[1]] / Length[dotEigenvalues]], Thickness[0.004],
        #} &, Table[Point[{{#[[-1]], h}], {h, -Pi, Pi, 2 Pi / 200}] & /@
        dotEigenvalues}], (* overlap integrals *)MapIndexed[
      {Hue[0.8 #2[[1]] / Length[dotEigenvalues]], Thickness[0.002],
        #} &, Line[Transpose[{{First /@ transmissionData, f2[#]}]}] & /@
        Transpose[Last /@ transmissionData]}]], Frame -> True,
    PlotRange -> pr, DisplayFunction -> Identity]] @@@
  (* function to be displayed *)
  {{Abs,
    (3 / 2 Abs[#] / Max[Abs[Last /@ transmissionData]]) &,
    {0, 3 / 2}},
    {1 / Pi Arg[#] &, 1 / Pi Arg[#] &, {-1, 1}}}]
```



■ Energy-Dependent Eigenfunctions

We end our excursion with an animation showing the dependence of the eigenfunctions on the energy of the incoming wave. We use fewer expansion states than in our example to speed up the calculation. But we take enough states to get a sufficient resolution for an animation.

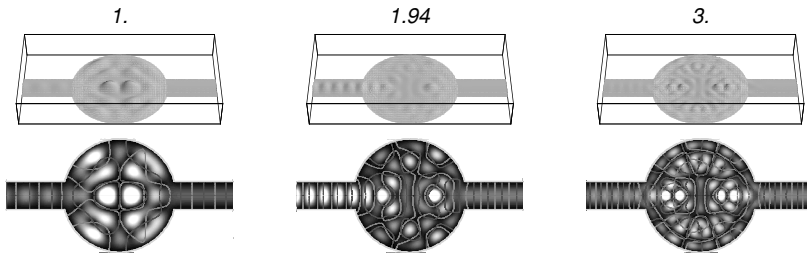
```
In[65]:= {p, o, q} = {4, 40, 4};
R = 10; L = 5;
prec = 50;
```

```

In[68]:= animationGraphicsPair[E_] := Module[{lhs, rhs, psi},
  (* construct matrices *) {lhs, rhs} = makeLHSAndRHS[prec, {4, 40}];
  (* construct solution vector *) psi = PseudoInverse[lhs];
  sol = If[MatrixQ[psi], psi.rhs, $Failed];
  (* construct wave function and gradient *)
  makeWaveFunctionAndGradientDefinition[{Ψ1, nabraΨ1}, sol, prec];
  (* show 3 D plot of density and contour plot of nodal lines *)
  Show[GraphicsArray[Block[{$DisplayFunction = Identity},
    {make3DPlot[ArcTan[Abs[#]^2/12] / (Pi/2) &,
      Ψ1, ViewPoint → {0, -3, 3}, PlotLabel → N[E]}],
    {makeReImAbsContourPlot[Ψ1]}]]]]

In[69]:= Do[animationGraphicsPair[E], {E, 1, 3, 2/100}]

```



This animation shows three frames. For the full result see Corner10-1-full.nb.

■ Summary

In this Corner, we calculated eigenfunctions belonging to the continuous spectrum of an infinite 2D domain with Dirichlet boundary conditions. Using the mode-matching technique and expansion in eigenfunctions of rectangular and circular domains, we reduced the problem to the numerical evaluation of a few integrals and the calculation of pseudoinverses. Using the expansion coefficients, we calculated the eigenfunctions and the corresponding current densities. The interested reader can continue with changing the ratio of the arm width to the center disk diameter L/R or calculating Bohm trajectories of microparticles instead of the streamlines of the current vector field. Or one could search for localized normalizable eigenfunctions below the beginning of the continuum spectrum $(\pi/L)^2$ using singular value decompositions of the left-hand side matrices (no right-hand side vectors) for states symmetric or antisymmetric with respect to the y -axis. Or one could investigate the statistics of the wave function and current values within the center disk [7]. Or one could cut a hole in the center disk and investigate the influence of a Aharonov–Bohm-type magnetic field through this hole [6]. Or...

■ References

- [1] K.-F. Berggren, K. N. Pichugin, A. F. Sadreev, and A. A. Starikov, "Signatures of Quantum Chaos in the Nodal Points and Streamlines in Electron Transport through Billiards," *Journal of Experimental and Theoretical Physics Letters*, **70**(6), 1999 pp. 403–409. DOI-Link: [dx.doi.org/10.1134/1.568188](https://doi.org/10.1134/1.568188)
- [2] K.-F. Berggren, A. F. Sadreev, and A. A. Starikov, "Quantum Chaos among Nodal Points and Streamlines at Ballistic Electron Transport through Open Quantum Dots," *Nanotechnology*, **12**(4), 2001 pp. 562–565. DOI-Link: [dx.doi.org/10.1088/0957-4484/12/4/338](https://doi.org/10.1088/0957-4484/12/4/338)
- [3] K.-F. Berggren, A. F. Sadreev, and A. A. Starikov, "Crossover from Regular to Irregular Behavior in Current Flow through Open Billiards," *Physical Review E*, **66**(1), 2002 pp. 016218-1–016218-10. DOI-Link: [dx.doi.org/10.1103/PhysRevE.66.016218](https://doi.org/10.1103/PhysRevE.66.016218)
- [4] T. Blomquist, H. Schanze, I. V. Zozoulenko, and H.-J. Stöckmann, "Geometry-Dependent Scattering through Quantum Billiards: Experiment and Theory," *Physical Review E*, **66**(2), 2002 pp. 026217-1–026217-4. DOI-Link: [dx.doi.org/10.1103/PhysRevE.66.026217](https://doi.org/10.1103/PhysRevE.66.026217)
- [5] H. Ishio, A. I. Saichev, A. F. Sadreev, and K.-F. Berggren, "Wave Function Statistics for Ballistic Quantum Transport through Chaotic Open Billiards: Statistical Crossover and Coexistence of Regular and Chaotic Waves," *Physical Review E*, **64**(5), 2001 pp. 056208-1–056208-12. DOI-Link: [dx.doi.org/10.1103/PhysRevE.64.056208](https://doi.org/10.1103/PhysRevE.64.056208)
- [6] K. N. Pichugin and A. F. Sadreev, "Aharonov–Bohm Oscillations of Conductance in Two-dimensional Rings," *Physical Review B*, **56**(15), 1997 pp. 9662–9673. DOI-Link: [dx.doi.org/10.1103/PhysRevB.56.9662](https://doi.org/10.1103/PhysRevB.56.9662)
- [7] A. F. Sadreev, "Current Statistics for Transport through Rectangular and Circular Billiards," *Physical Review E*, **70**(1), 2004 pp. 016208-1–016208-7. DOI-Link: [dx.doi.org/10.1103/PhysRevE.70.016208](https://doi.org/10.1103/PhysRevE.70.016208)
- [8] L. Wirtz, C. Stampfer, S. Rotter, and J. Burgdörfer, "Semiclassical Theory for Transmission through Open Billiards: Convergence towards Quantum Transport," *Physical Review E*, **67**(1), 2003 pp. 016206-1–016206-13. DOI-Link: [dx.doi.org/10.1103/PhysRevE.67.016206](https://doi.org/10.1103/PhysRevE.67.016206)

■ Additional Material

Corner10-1-full.nb

Available at www.mathematica-journal.com/issue/v10i1/download.

Michael Trott

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

Efficient, High-Quality Force-Directed Graph Drawing

Yifan Hu

We propose a graph drawing algorithm that is both efficient and high quality. This algorithm combines a multilevel approach, which effectively overcomes local minimums, with the Barnes and Hut [1] octree technique, which approximates short- and long-range force efficiently. Our numerical results show that the algorithm is comparable in speed to Walshaw's [2] highly efficient multilevel graph drawing algorithm, yet gives better results on some of the difficult problems. In addition, an adaptive cooling scheme for the force-directed algorithms and a general repulsive force model are proposed. The proposed graph drawing algorithm and others are included with *Mathematica* 5.1 and later versions in the package `DiscreteMath`Graph`Plot`.

■ 1. Introduction

Graphs are often used to encapsulate the relationship between objects. Graph drawing enables visualization of these relationships. The usefulness of the representation is dependent on the aesthetics of the drawing. While there are no strict criteria for aesthetics, it is generally agreed that minimal edge crossing, evenly distributed vertices, and depiction of graph symmetry is desirable.

This problem has been studied extensively in the literature [3] and many approaches have been proposed. In this article we concentrate on drawing undirected graphs with straight-line edges using force-directed methods [4, 5, 6, 7]. Force-directed methods, however, are one of many classes of methods proposed for straight-edge drawing. Other methods include the spectral method [8] and the high-dimensional embedding method [9].

A force-directed algorithm models the graph drawing problem through a physical system of bodies with forces acting between them. The algorithm finds a good placement of the bodies by minimizing the energy of the system. There are many variations of force-directed algorithms. The algorithm of Fruchterman and Reingold [4], which is based on the work of [5, 6], models the graph drawing problem with a system of springs between neighboring vertices of the graph,

pulling them together. At the same time, repulsive electrical forces that exist push all vertices away from each other. The algorithm of Kamada and Kawai [7], on the other hand, associates springs between all vertices, with the ideal length of a spring proportional to the graph distance of the vertices. In a force-directed algorithm, the energy of the system is typically minimized iteratively by moving the vertices along the direction of the force. This amount may be large initially, but reduces gradually based on a “cooling schedule.”

There are two limiting factors in drawing large graphs for standard force-directed algorithms. The first is that the physical model typically has many local minimums, particularly so for a large graph. Starting from a random configuration, the system is likely to settle in a local minimum. This may be improved, to a limited extent, by using a slow cooling schedule at the expense of more iterations. Nevertheless, it is practically impossible to use the standard force-directed algorithms to find a good layout of very large graphs.

The second limiting factor is the computational complexity of the standard force-directed algorithms. In the algorithm of Fruchterman and Reigold [4], for any given vertex, repulsive force from all other vertices needs to be calculated. This makes the per iteration cost of the algorithm $O(|V|^2)$, with $|V|$ the number of vertices in the graph. The algorithm of Kamada and Kawai [7] requires the calculation of the graph distance among all vertices with the force based on that. Thus the algorithm not only has a computational complexity of $O(|V| \cdot |E|)$, with $|E|$ the number of edges in the graph but also a memory complexity of $O(|V|^2)$, although the latter can be circumvented at the cost of repeated calculations of the graph distances on the fly, instead of storing them in memory.

To overcome the first limiting factor, a multilevel approach was proposed. This idea has been successfully used in many fields, including graph partitioning [10, 11, 12] and was found to be able to overcome the localized nature of the Kernighan–Lin algorithm. Fruchterman and Reigold [4, 1148] alluded to that type of solution when, in the context of overcoming local minimums, they stated “we suspect that if we apply a multi-grid technique that allows whole portions of the graph to be moved, it might be of some help...”. Harel and Koren [13], extending the earlier work of Hadany and Harel [14], proposed the so-called multiscale approach. In that approach, a sequence of coarser and coarser graphs are formed by finding the k -centers and the distance matrix associated with them. They used the Kamada and Kawai spring model [7], thus incurring high computational complexity for distance calculation and memory. Although, for the force calculation the algorithm has a computational complexity of $O(|V| \log(|V|))$, achieved by restricting force calculation to a neighborhood, thus ignoring long-range force. Gajer et al. [15] built up the multilevel of graphs by using maximal independent vertex sets, with the i th level consisting of a maximal independent vertex set such that the vertices are of distance $\geq 2^{i-1} + 1$, $i = 1, 0, \dots$ apart. They avoided the high computational and memory complexity by calculating the graph distances on the fly and only for a restricted neighborhood, thus also ignoring the long-range force. Walshaw [2] proposed a multilevel algorithm and demonstrated that it was able to draw graphs as large as a quarter of a million vertices in a few minutes. Based on the author’s earlier work in graph

partitioning, the coarser graphs are formed by finding a maximal independent edge set and collapsing these edges. The forces between vertices are based on the Fruchterman and Reigold spring-electrical model [4]. Long-range force is again ignored by restricting the force calculation to a neighborhood with a radius that decreases as one moves from coarser to finer graphs and coincides with the radius used in the original graph.

Reducing the computational cost by restricting force calculation to a neighborhood has been an often used practice, dating at least as far back as Fruchterman and Reigold [4]. Such a practice, however, comes at a cost. Because long-range forces are ignored, there is no force to evenly distribute faraway vertices. When used within the multilevel approach, Walshaw [2] argued that global untangling has been achieved on coarser graphs, thus for the final large graphs, restricting force calculation to a small neighborhood does not penalize the quality of the placement. While this is to a large extent true, for some graphs we found that the lack of long-range force did hurt at least one, if not more, of the drawings in [2].

It is possible to take account of long-range forces in an efficient way in the spring-electrical model. In this model the attractive force (the spring force) is only between neighboring vertices, while the repulsive force is global and is proportional to the inverse of the (physical) distance between vertices. The repulsive force calculation resembles the n -body problem in physics, which has been well studied. One of the widely used techniques for calculating the repulsive forces in $O(n \log(n))$ time with good accuracy, but without ignoring long-range force, is to treat groups of faraway vertices as a supernode using a suitable data structure, as in the Barnes and Hut algorithm [1]. This idea was implemented for graph drawing by both Tunkelang [16] and Quigley and Eades [17, 18]. Tunkelang combined the Barnes and Hut algorithm with a conjugate gradient method, thus per iteration computational is only $O(|V| \log(|V|))$, even though long-range forces are approximated to the required accuracy. However, the algorithm is not suitable for large graphs because the conjugate gradient is a local optimization algorithm and the number of conjugate gradient iterations increases as the size of the graph increases. Quigley and Eades [17, 18] also used the Barnes and Hut algorithm for efficient and accurate force calculation. In addition, they employed a multilevel scheme that they called hierarchical clustering. However, they used that scheme for the visual abstraction of graphs, rather than for the placement of vertices. Therefore, the algorithm was not suitable for large graphs.

In this article we propose an algorithm that is both efficient and of high quality for large graphs. The algorithm is included with *Mathematica* 5.1 and later versions in the package `DiscreteMath`GraphPlot`. We combine a multilevel approach, which effectively overcomes local minimums, with the Barnes and Hut octree algorithm, which approximates short- and long-range forces satisfactorily and efficiently. In addition, we propose an adaptive cooling scheme for the basic force-directed algorithms and a scheme for selecting the optimal depth of the octree/quadtree in the Barnes and Hut algorithm. Our numerical results show that the algorithm is competitive with Walshaw's [2] highly efficient graph drawing algorithm, yet gives better results on some of the difficult problems. We also analyze the distortion effect of the standard Fruchterman-Reigold

spring-electrical model and propose a general repulsive force model to overcome this side effect.

In Section 2, we give definitions and notations. In Section 3, we present the basic force-directed algorithms, as well as an adaptive cooling scheme. In Section 4, we briefly introduce the Barnes–Hut force calculation algorithm. In Section 5, we describe the multilevel scheme used. In Section 6, we compare the efficiency and drawings of our algorithm with that of Walshaw [2]. In Section 7, we conclude by suggesting some future works.

■ 2. Definitions and Notations

We use $G = \{V, E\}$ to denote an undirected graph, with V the set of vertices and E the set of edges. We assume the graph is connected. Disconnected graphs can be drawn by laying out each of the components separately.

If two vertices i and j form an edge, we denote that as $i \leftrightarrow j$. The coordinates of node i are denoted as x_i , and we use $\|x_i - x_j\|$ to denote the 2-norm distance between vertices i and j . We use $d(i, j)$ to denote the graph distance between vertices i and j , and we use $\text{diam}(G)$ to denote the diameter of the graph.

The graph layout problem is one of finding a set of coordinates, $x = \{x_i \mid i \in V\}$, with $x_i \in \mathbb{R}^2$ or $x_i \in \mathbb{R}^3$, for 2D or 3D layout, respectively, such that when the graph G is drawn with vertices placed at these coordinates, the drawing is visually appealing.

■ 3. Force-Directed Algorithms

In this section we present the basic force-directed algorithms and analyze the characteristics of the layout given by the spring-electrical model. We will propose a general repulsive force model, as well as an adaptive step control scheme.

□ 3.1. Spring and Spring-Electrical Models

Force-directed algorithms model the graph layout problem by assigning attractive and repulsive forces between vertices and finding the optimal layout by minimizing the energy of the system.

The model of Fruchterman and Reingold [4], which we refer to as the *spring-electrical model*, has two forces. The repulsive force, f_r , exists between any two vertices i and j and is inversely proportional to the distance between them. The attractive force, f_a , on the other hand, exists only between neighboring vertices and is proportional to the square of the distance

$$\begin{aligned} f_r(i, j) &= -C K^2 / \|x_i - x_j\|, & i \neq j, & i, j \in V \\ f_a(i, j) &= \|x_i - x_j\|^2 / K, & i \leftrightarrow j. \end{aligned} \quad (1)$$

The combined force on a vertex i is

$$f(i, x, K, C) = \sum_{i \neq j} \frac{-CK^2}{\|x_i - x_j\|^2} (x_j - x_i) + \sum_{i \leftrightarrow j} \frac{\|x_i - x_j\|}{K} (x_j - x_i). \quad (2)$$

In these formulas, K is a parameter known as the optimal distance [4], or natural spring length [2]. The parameter C regulates the relative strength of the repulsive and attractive forces and was introduced in [2]. It is easy to see that for a graph of two vertices linked by an edge, the force on each vertex diminishes when the distance between them is equal to $K(C)^{1/3}$. The total energy of the system can be considered as

$$\text{Energy}_{\text{se}}(x, K, C) = \sum_{i \in V} f^2(i, x, K, C),$$

where x is the vector of coordinates, $x = \{x_i \mid i \in V\}$.

From a mathematical point of view, changing the parameters K and C does not actually change the minimal energy layout of the graph but merely scales the layout, as the following theorem shows.

Theorem 1. Let $x^* = \{x_i^* \mid i \in V\}$ minimizes the energy of the spring-electrical model $\text{Energy}_{\text{se}}(x, K, C)$, then sx^* minimizes $\text{Energy}_{\text{se}}(x, K', C')$, where $s = (K' / K)(C' / C)^{1/3}$. Here K, C, K' and C' are all positive real numbers.

Proof: This follows simply by the relationship

$$\begin{aligned} f(i, x, K, C) &= \sum_{i \neq j} \frac{-CK^2}{\|x_i - x_j\|^2} (x_j - x_i) + \sum_{i \leftrightarrow j} \frac{\|x_i - x_j\|}{K} (x_j - x_i) \\ &= \left(\frac{C}{C'}\right)^{2/3} \frac{K}{K'} \left(\sum_{i \neq j} \frac{-C'(K')^2}{\|sx_i - sx_j\|^2} (sx_j - sx_i) \right. \\ &\quad \left. + \sum_{i \leftrightarrow j} \frac{\|sx_i - sx_j\|}{K'} (sx_j - sx_i) \right) \\ &= \left(\frac{C}{C'}\right)^{2/3} \frac{K}{K'} f(i, sx, K', C'), \end{aligned}$$

where $s = (K' / K)(C' / C)^{1/3}$. Thus,

$$\text{Energy}_{\text{se}}(x, K, C) = \left(\frac{C}{C'}\right)^{4/3} \left(\frac{K}{K'}\right)^2 \text{Energy}_{\text{se}}(sx, K', C').$$

Even though the parameters K and C do not have any bearing on the optimal layout from a mathematical point of view, from an algorithmic point of view, if an iterative algorithm (see Algorithm 1) is applied to minimize the energy from an initial layout, choosing a suitable K or C to reflect the range of the initial position will help the convergence to the optimal layout. Throughout this article, unless otherwise specified, we fix $C = 0.2$ as in [2], but vary K for this purpose.

One intrinsic feature of graph drawings by the spring-electrical model is that vertices in the periphery tend to be closer to each other than those in the center, even for a uniform mesh. We call this the *peripheral effect*. For example, Figure 1 shows a mesh of 100 vertices laid out using the spring-electrical model, with vertices near the outside boundary clearly closer to each other than those near the center.

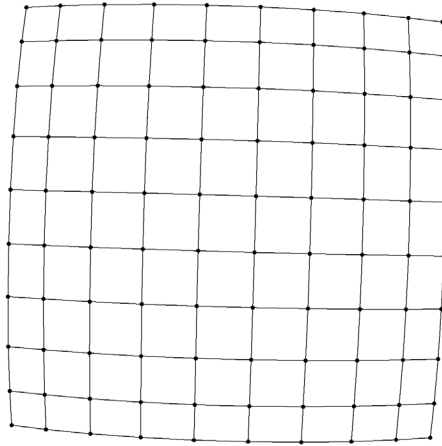


Figure 1. A 10×10 regular mesh laid out using the spring-electrical model. This graph and the others in this article were drawn with the *Mathematica* GraphPlot command, which is discussed in Section 6.

This peripheral effect is more profound for graphs with large diameter. We studied this effect for a line graph, with 100 vertices linked in a line. The vertices are numbered 1 to 100, with vertex 50 and 51 in the middle of the line. We find an accurate layout under the spring-electrical model by finding the root of a system of equations $f(i, x, K, C) = 0$ (see equation (2)) using *Mathematica*'s FindRoot function, instead of the usual force-directed iterative algorithm, because the latter is far less accurate. We set the parameters $K = 1$ and $C = 1$. Theorem 1 shows that the exact values of these two parameters are not important.

Figure 2 shows the distribution of edge lengths. As can be seen, the edge lengths of the 99 edges vary from 4.143 at the middle to 1.523 at the sides, with the ratio of the longest length to the shortest $4.143 / 1.523 = 2.72$.

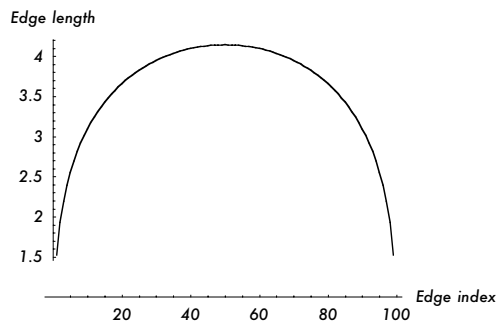


Figure 2. Distribution of edge lengths for a line graph of 100 vertices and 99 edges, laid out using the spring-electrical model.

The reason for this distortion effect at the periphery is the strong long-range force that decays slowly as the distance increases. While typically this strong long-range force does not interfere with the aesthetics of the layout, some applications, such as tree graphs, tend to suffer more. For these classes of graphs, the following general repulsive force model can be used.

$$f_r(i, j) = -C K^{1+p} / \|x_i - x_j\|^p, \quad i \neq j, \quad i, j \in V, \quad p > 0. \quad (3)$$

The larger the parameter p , the weaker the long-range repulsive force. However, too large a value of p , thus too weak a long-range force, could cause the graph that should be spread out to crease instead. We found that $p = 2$ works well. Figure 3 shows the peripheral effect against the size of the line graph, for both the general model (4) with $p = 2$ and $p = 3$, and the Fruchterman and Reigold force model (1), which corresponds to the general model with $p = 1$. As can be seen, the general model with $p > 1$ reduces the peripheral effect significantly.

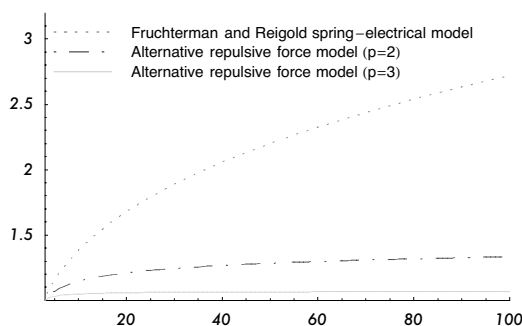


Figure 3. A plot of the ratio between the largest and smallest edge lengths for line graphs of size 3 to 100 vertices.

In the force model of Kamada and Kawai [7], which we will refer to as the *spring model*, springs are attached between any two pairs of vertices, with the ideal

length of a spring proportional to the graph distance between these two vertices. Thus both the repulsive and attractive forces are expressed as

$$f_r(i, j) = f_a(i, j) = \|x_i - x_j\| - d(i, j), \quad i \neq j, \quad i, j \in V \quad (4)$$

and the spring energy is

$$\text{Energy}_s(x) = \sum_{i \neq j, i, j \in V} (\|x_i - x_j\| - d(i, j))^2. \quad (5)$$

The spring model does not suffer from the peripheral effect of the spring-electrical model. It can, however, suffer from its relatively weak repulsive forces (see Section 6.3). Furthermore, as discussed later, in the spring-electrical model, the long-range force between all vertices can be well approximated by grouping vertices together using the octree/quadtrees technique. This is, however, not possible in the spring model.

□ 3.2. An Adaptive Cooling Scheme

The energy of both the spring-electrical and the spring models can be minimized iteratively by moving the vertices along the direction of forces exerted on them. The following force-directed algorithm iteratively minimizes the system energy, with f_a and f_r as defined in (1) or (5). The algorithm starts with a random, or user-supplied, initial layout.

- ForceDirectedAlgorithm(G, x, tol) {
 - converged = FALSE;
 - step = initial step length;
 - Energy = Infinity
 - while (converged equals FALSE) {
 - * $x^0 = x$;
 - * $\text{Energy}^0 = \text{Energy}$; Energy = 0;
 - * for $i \in V$ {
 - $f = 0$;
 - for $(j \leftrightarrow i)$ $f := f + \frac{f_a(i, j)}{\|x_j - x_i\|} (x_j - x_i)$;
 - for $(j \neq i, j \in V)$ $f := f + \frac{f_r(i, j)}{\|x_j - x_i\|} (x_j - x_i)$;
 - $x_{i:} = x_i + \text{step} * (f / \|f\|)$;
 - Energy := Energy + $\|f\|^2$;
 - * }
 - * step := update_steplength(step, Energy, Energy^0);
 - * if ($\|x - x^0\| < K \text{ tol}$) converged = TRUE;
 - }
 - return x ;
- }

Algorithm 1. An iterative force-directed algorithm.

In the algorithm, $\text{tol} > 0$ is a termination tolerance. The algorithm stops when a change in the layout between subsequent iterations is less than $K \text{ tol}$.

As in [2], we update the layout of a vertex i as soon as the force for this vertex is calculated, instead of waiting until forces for all vertices have been calculated. This improves the convergence of the iterative procedure, much like the fact that among stationary iterative linear system solvers, the Gauss–Seidel algorithm is often faster than the Jacobi algorithm.

In Algorithm 1, it is necessary to update the step length step. The “cooling schedule” used in most expositions (e.g., [19]) of force-directed algorithms allows large movements (large step length) at the beginning of the iterations, but the step length reduces as the algorithm progresses. Walshaw [2] used a simple scheme,

$$\text{step} := t \text{ step}, \quad (6)$$

with $t = 0.9$. We found this to be adequate in the refinement phase of a multi-level force-directed algorithm. However, for an application of a force-directed algorithm from a random initial layout, an adaptive step length update scheme is more successful in escaping from local minimums. This adaptive scheme is motivated by the trust region algorithm for optimization [20], where step length can increase as well as reduce, depending on the progress made. Here we measure progress by the decrease in system energy.

- function update_steplength(step, Energy, Energy⁰)
- if (Energy < Energy⁰) {
 - progress = progress + 1;
 - if (progress > = 5) {
 - * progress = 0;
 - * step := step / t ;
 - }
- } else {
 - progress = 0;
 - step := t step;
- }

In the preceding algorithm, progress is a static variable that is initialized to zero and parameter $t = 0.9$. The idea of this algorithm is that the step length is kept unchanged if energy is being reduced and increased to step/t if the energy is reduced more than five times in a row. We only reduce the step length if energy increases.

Compared with the simple step length update scheme (6), the adaptive scheme is much better in escaping from local minimums. Figure 4 shows the result of applying 70 iterations of Algorithm 1 to the jagmesh1 graph with 936 vertices from MatrixMarket (math.nist.gov/MatrixMarket). The adaptive scheme is clearly much better and is able to untangle the graph a lot further than the simple scheme.

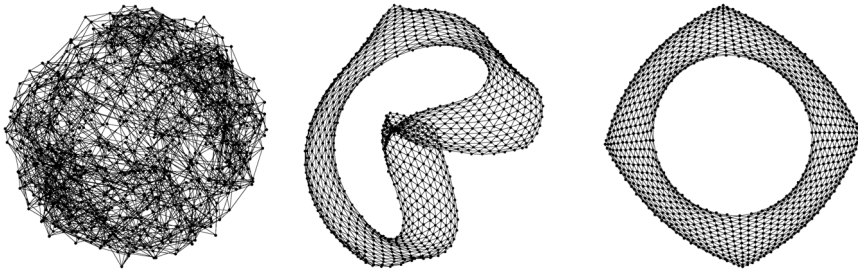


Figure 4. Comparing adaptive and simple step length update schemes on jagmesh1 after 70 iterations: simple scheme (left); adaptive scheme (middle); what the layout should look like (right).

■ 4. Barnes–Hut Force Calculation

Each iteration of Algorithm 1 involves two loops. The outer loop iterates over each $i \in V$. Of the two inner loops that calculate the attractive and repulsive forces, the latter is the most expensive and loops over each $j \neq i, j \in V$. Thus the overall complexity is $O(|V|^2)$.

The repulsive force calculation resembles the n -body problem in physics, which is well studied. One of the widely used techniques to calculate the repulsive forces in $O(n \log n)$ time with good accuracy, but without ignoring long-range forces, is to treat groups of faraway vertices as supernodes, using a suitable data structure [1]. This idea was adopted by Tunkelang [16] and Quigley [18]. Both used an octree (3D) or quadtree (2D) data structure. In principal, other space decomposition methods can also be used. For example, Pulo [21] investigated recursive Voronoi diagrams.

For simplicity, hereafter we use the term octree exclusively, which should be understood as quadtree in the context of 2D layout. An octree data structure is constructed by first forming a square (or cube in 3D) that encloses all vertices. This is the level 0 square. This square is subdivided into four squares (or eight cubes) if it contains more than one vertex and forms the level 1 squares. This process is repeated until level L , where each square contains no more than one vertex. Figure 5 (left) shows an octree on the jagmesh1 graph.

The octree forms a recursive grouping of vertices and can be used to efficiently approximate the repulsive force in the spring-electrical model. The idea is that in calculating the repulsive force on a vertex i , if a cluster of vertices, S , lies in a square that is “far” from i , the whole group can be treated as a supernode. The supernode is assumed to situate at the center of gravity of the cluster, $x_S = (\sum_{j \in S} x_j) / |S|$. The repulsive force on vertex i from this supernode is

$$f_r(i, S) = -|S|CK^2 / \|x_i - x_S\|.$$

However, we need to define what “far” means. Following [16, 18], we define the supernode S to be faraway from vertex i , if the width of the square that contains the cluster is small, compared with the distance between the cluster and the vertex i ,

$$\frac{d_S}{\|x_i - x_S\|} \leq \theta. \quad (7)$$

Here d_S is the width of the square that the cluster lies in, and $\theta \geq 0$ is a parameter. The smaller the value of θ , the more accurate the approximation to the repulsive force and the more computationally expensive it is. We found that $\theta = 1.2$ is a good compromise and will use this value throughout the article. This inequality (7) is called the Barnes–Hut opening criterion [1].

The octree data structure allows efficient identification of all the supernodes that satisfy (7). The process starts from the level 0 square. Each square is checked and recursively opened until the inequality (7) is satisfied. Figure 5 (right) shows all the supernodes (the squares) and the vertices these supernodes consist of, with reference to vertex i located at the top-middle part of the graph. In this case we have 32 supernodes.

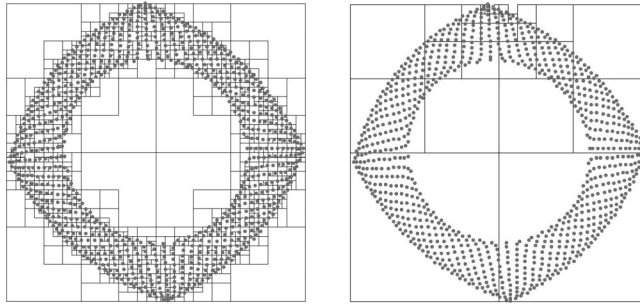


Figure 5. An illustration of the octree data structure: the overall octree (left); supernodes with reference to a vertex at the top-middle part of the graph, with $\theta = 1$ (right).

Under reasonable assumption [1, 22] of the distribution of vertices, it can be proved that building the octree takes a time complexity of $O(|V| \log(|V|))$. Finding all the supernodes with reference to a vertex i can also be done in a time complexity $O(\log(|V|))$. Overall, by using an octree structure to approximate the repulsive force, the complexity for each iteration of Algorithm 1 under the spring-electrical model is reduced from $O(|V|^2)$ to $O(|V| \log(|V|))$.

We only build the octree structure once every outer loop. Note that the positions of the vertices are actually updated continuously within the loop; therefore, as they move about, some vertices may not be in the squares where they are supposed to lie. However, we found that this does not cause a problem in practice. Consequently, we observed that construction of the tree structure only takes a fraction of the total time, and the majority of the time in Algorithm 1 is spent in finding the supernodes, as well as in the force calculations.

For very large graphs, it may happen that some of the vertices are very close to each other. Therefore, if we keep subdividing the squares without a limit, we may end up with a tree structure with one or more very deep branches. This will make the algorithm less efficient. In particular, a disproportionately large amount of time will be spent in searching through the tree to find supernodes. It is therefore necessary to limit the number of levels in the octree. We denote this limit `max_tree_level`. Even if a square at level `max_tree_level` still has multiple vertices, it is not subdivided further. We call such a square a dense leaf (of the octree).

However, it is difficult to decide a priori how many levels should be allowed. If we set the `max_tree_level` too small, we will have many vertices in the same square as at the last level. This increases the average number of supernodes, because when identifying supernodes needed to approximate the repulsive force on a vertex i , if a dense leaf happens to be close to i , each vertex on the dense leaf has to be treated individually as a supernode since no subgrouping is available. In the extreme case when `max_tree_level` = 0, every vertex belongs to one dense leaf, and there are $|V| - 1$ supernodes that correspond to every vertex i . On the other hand, although a large value for `max_tree_level` reduces the average number of supernodes, it increases the number of squares that need to be traversed due to the deep branches.

We use an adaptive scheme to automatically find the optimal `max_tree_level`. This is essentially a one-dimensional optimization problem with the variable being `max_tree_level` and the objective function the CPU time of each outer iteration of Algorithm 1, which consists largely of the time to transverse the octree and the time in the repulsive force calculation. So one way to find the optimal `max_tree_level` is to measure the CPU time of the outer loop and increase/decrease `max_tree_level` by one each time until the bottom of a valley is located. However, CPU time measurement can fluctuate and such a scheme may cause a different `max_tree_level` from run to run. This in turn gives a different layout between runs, which is undesirable. Instead, we use

$$b(\text{max_tree_level}) = \text{counts} + \alpha \text{ ns} \quad (8)$$

as the objective function, where `counts` is the total number of squares traversed, `ns` is the total number of supernodes found during one outer iteration, and α is a parameter chosen so that (8) gives the best estimate of the CPU time. Through numerical experiments, we found that α in the range of 1.5 to 2.0 gives very good correlation to CPU time measurement. Thus we used $\alpha = 1.7$. The adaptive scheme starts with `max_tree_level` = 8. After one outer iteration, we set `max_tree_level` = 9. Then, depending on whether the estimated CPU time increases or decreases, we try a smaller or larger depth. If we ever hit a depth already tried, we end the procedure and use the depth corresponding to the smallest estimated CPU time. Typically, we found that the procedure converges within 3-4 outer iterations, and the `max_tree_level` located is very near the optimal value. For smaller graphs of a few thousand vertices, typically `max_tree_level` settles down at around 8, while for very large graphs, `max_tree_level` can go as high as 11.

■ 5. The Multilevel Algorithm

While approximation of long-range force using octree data structure greatly reduces the complexity of Algorithm 1, it does not change the fact that the algorithm repositions one vertex at a time, instead of laying out a whole region as a unit. Large graphs typically have many local minimal energy configurations, and such an algorithm is likely to settle to one of the local minimums. The adaptive step length control scheme we introduce goes some way toward a better layout, but as Figure 4 shows, is still insufficient toward a global minimum.

The multilevel approach has been used in many large-scale combinatorial optimization problems, such as graph partitioning [11, 12, 23], matrix ordering [24], the traveling salesman problem [25], and has proven to be a very useful meta-heuristic tool [26]. The multilevel approach was also used in graph drawing [2, 13, 14]. In particular, Walshaw [2] was able to lay out graphs with up to 225,000 vertices in a few minutes, and largely of good quality.

The multilevel approach has three distinctive phases: coarsening, coarsest graph layout, and prolongation and refinement. In the coarsening phase, a series of coarser and coarser graphs, $G^0, G^1, \dots, G^l$, are generated. The aim is for each coarser graph G^{k+1} to encapsulate the information needed to lay out its “parent” G^k while containing fewer vertices and edges. The coarsening continues until a graph with only a small number of vertices is reached. The optimal layout for the coarsest graph can be found cheaply. The layouts on the coarser graphs are recursively prolonged to the finer graphs, with further refinement at each level. Hereafter, we use a superscript to denote the level. For example, x^k is the coordinates of vertices in the level k graph G^k , $k = 0, \dots, l$.

□ 5.1. Graph Coarsening

There are a number of ways to coarsen an undirected graph. One frequently used method is based on edge collapsing (EC) [11, 12, 23], in which pairs of adjacent vertices are selected and each pair is coalesced into one new vertex. Each vertex of the resulting coarser graph has an associated weight, equal to the number of original vertices it represents. Each edge of the coarser graph also has a weight associated with it. Initially, all edge weights are set to 1. During coarsening, edge weights are unchanged unless both merged vertices are adjacent to the same neighbor. In this case, the new edge is given a weight equal to the sum of the weights of the edges it replaces. The edges to be collapsed are usually selected using *maximal matching*. This is a maximal set of edges, no two of which are incident to the same vertex. For undirected graph partitioning, heavy-edge matching has been found to work well. Here, the idea is to preferentially collapse heavier edges. When looking through a neighbor list for an unmatched vertex, an edge with the largest weight is selected. In the context of graph drawing, Walshaw [2] chose to keep vertex weights in the coarser graphs as uniform as possible by matching a vertex with a neighbor with the smallest vertex weight. We found that both heavy-edge matching and matching with a vertex of smallest weight give graph layouts of similar quality and used the former in this article. Figure 6 illustrates a graph (left) and the result (middle) of coarsening using EC.

Other coarsening methods have been proposed. In [27], a maximal independent vertex set (MIVS) of a graph is chosen as the vertices for the coarser graph. An independent set of vertices is a subset of the vertices such that no two vertices in the subset are connected by an edge in the graph. An independent set is maximal if the addition of an extra vertex always destroys the independence. Edges of the coarser graph are formed by linking two vertices in MIVS by an edge if their distance apart is no greater than 3. Figure 6 illustrates a graph (left) and the result (right) of coarsening using MIVS.

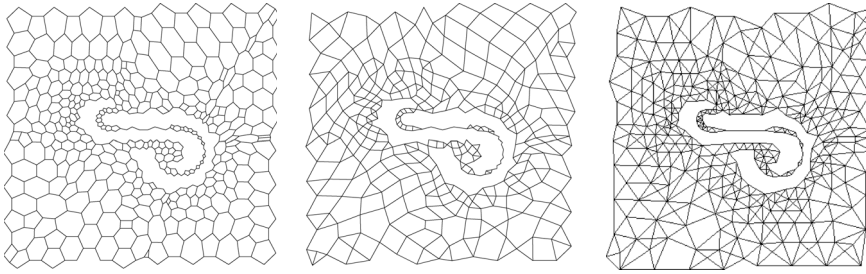


Figure 6. An illustration of graph coarsening: original graph with 788 vertices (left); a coarser graph with 403 vertices resulted from EC (center); a coarser graph with 332 vertices resulted from MIVS (right).

We have implemented both EC and MIVS coarsening schemes for our multi-level graph drawing algorithm. Notice that with EC, the coarse graph always has more than 50% of the vertices of the original graph. For graphs with a high average degree, it may happen that the number of vertices in the coarser graph, G^{i+1} , may be very close to the number of vertices in the original graph, G^i . This can significantly increase the complexity of the multilevel algorithm. Therefore, we will stop coarsening if there are only two vertices in the graph or

$$\frac{|V^{i+1}|}{|V^i|} > \rho. \quad (9)$$

We used $\rho = 0.75$. Here V^i is the vertices in G^i .

On the other hand, for a connected graph, the MIVS coarsening usually, though not necessarily, results in a coarser graph with less than 50% of the number of vertices of the original graph. In our experience, for graph layout, EC tends to give slightly better results than MIVS, probably because it coarsens less aggressively. However, MIVS is usually faster, due to the lower complexity. To overcome the complexity issue of EC, we propose a third scheme, HYBRID. This scheme uses EC whenever possible, however, if the threshold (9) is breached, we use MIVS coarsening instead.

□ 5.2. Initial Layout on the Coarsest Graph

On the coarsest level, we lay out the graph using Algorithm 1 combined with the adaptive step length control scheme. For MIVS and HYBRID coarsening schemes, the coarsest graph has only two vertices, thus a random placement of the two vertices would be sufficient.

□ 5.3. The Refinement Step

The layouts on the coarser graphs are recursively prolonged to the finer graphs, with further refinement at each level.

If graph $G^{i+1} = \{V^{i+1}, E^{i+1}\}$ was derived using EC from $G^i = \{V^i, E^i\}$, the position of a vertex $u \in V^{i+1}$ is given to the two vertices $v, w \in V^i$ that collapse into u . If G^{i+1} was derived using MIVS from G^i , then a vertex $v \in V^i$ either inherits the position if it is in the MIVS, or v must have one or more neighbors in MIVS, in which case the position of v is the average of the positions of these neighbors.

Once the prolongation is carried out to give an initial layout for G^i , this layout is refined using Algorithm 1. As in [2], if in the initial layout two vertices happen to be at the same position, as could be the case with EC-based coarsening, a random perturbation is done to separate them. Because the initial layout is derived from the layout of a coarser graph, this layout is typically already well placed globally and what is required is just some local adjustment. Therefore, we found that it is preferable to use a conservative step length update scheme. The simple scheme (6) works well.

Although the initial layout in graph G^i prolonged from the coarser graph G^{i+1} is usually globally well positioned, a naive application of Algorithm 1 would cause large movement of vertices, thus potentially destroying the useful information inherited. For example, in the context of the spring model, the physical distance between two vertices u and v in the initial layout in G^i is roughly the same as the graph distance of the corresponding vertices in G^{i+1} , that is

$$\|x_u^i - x_v^i\| \approx d_{G^{i+1}}(u', v'),$$

where u' and v' are two vertices in the coarser graph G^{i+1} that corresponds to u and v . However, to minimize the energy on level i , we want $\|x_u - x_v\|$ to be as close to the graph distance of them in G^i , that is, we want

$$\|x_u^i - x_v^i\| \approx d_{G^i}(u, v).$$

Typically $d_{G^{i+1}}(u', v')$ is much smaller than $d_{G^i}(u, v)$. If the initial layout is used as is, then to achieve the minimal energy, the graph has to be expanded by the ratio between these two distances through a series of large moves, which is inefficient and could lose some information in the initial good layout.

Therefore, for a multilevel algorithm based on the spring model, we scale the initial coordinates by the ratio of the pseudo-diameters between the two subsequent levels of graphs,

$$\gamma = \text{diam}(G^i) / \text{diam}(G^{i+1}). \quad (10)$$

For the spring-electrical model, Walshaw [2] suggested keeping the coordinates unchanged but reducing the natural spring length K^i to

$$K^i = K^{i+1} / \gamma,$$

with $\gamma = \sqrt{7/4}$. Walshaw derived this value based on examining a graph with four vertices. We use instead (10) and found it to be equally effective. On the coarsest level, we set K^l to be the average edge length of the initial random layout, as in [2].

To avoid the $O(|V|^2)$ complexity of the repulsive force calculation in the spring-electrical model, Walshaw [2], following Fruchterman and Reingold [4], cut off the contributions from faraway vertices. Only repulsive forces from vertices within a certain radius were considered. Specifically, on the i th level, for vertex v , repulsive forces from vertex u were ignored when $\|x_u^i - x_v^i\| > R^i$. If a small radius R^i is chosen, then repulsive force over even a short distance is ignored. This can cause faraway regions to collapse into each other due to the lack of force to spread them out. On the other hand, a larger radius R^i is expensive. In the extreme, $R^i = \infty$ gives us the $O(|V|^2)$ complexity. Walshaw [2] proposed the radius R^i

$$R^i = 2(i+1)K^i.$$

Notice that the radius is larger, relative to the natural spring length, on the coarser graphs, for which a large R^i value would not be too costly due to the smaller sizes of the graphs. The effect of this cutoff, which would otherwise be more profound, was largely dampened because of the multilevel approach. Overall, the power of the multilevel approach means that the good quality global layout achieved on coarser graphs was inherited by the finer graphs, and a small radius on the finest graphs usually worked well. Nevertheless, we found that for some graphs, particularly those with a hollow interior, such as the graph *finan512* in Section 6, the adverse effect of ignoring long-range repulsive force was obvious. With our use of octree data structure, we no longer need to use a cutoff radius for the spring-electrical model. Nevertheless we set a cutoff radius of

$$R^i = r(i+1)K^i, \quad (11)$$

and by default we set $r = \infty$, which is equivalent to not using a cutoff radius at all. This, however, also allows us to have a finite r value to experiment with. For the spring model, to avoid calculation of the all-to-all distance matrix needed to compute the force (5), we used a similar strategy, by only calculating the distance of two vertices u and v and their attractive/repulsive forces, if

$$d(u, v) \leq r(i+1), \quad (12)$$

with a default r value of 4.

□ 5.4. The Multilevel Algorithm

We present the overall multilevel algorithm in the following. In the algorithm, $n^i = |V^i|$ is the number of vertices in the i th level graph G^i . x^i is the coordinate vector for the vertices in V^i . We represent G^i by a symmetric matrix G^i , with the entries of the matrix the edge weights. The prolongation operator from G_{i+1} to G_i is also represented by a matrix P^i , of dimension $n^i \times n^{i+1}$. For details on the implementation of the multilevel process, and some examples, see [24]. The starting point is the original graph, $G_0 = G$.

function MultilevelLayout (G^i , tol)

- Coarsest graph layout
 - if ($n^{i+1} < \text{MinSize}$ or $n^{i+1} / n^i > \rho$) {
 - * $x^{i+1} = \text{random initial layout}$
 - * $x^i = \text{ForceDirectedAlgorithm}(G^i, x^{i+1}, \text{tol})$
 - * return x^i
 - }
- The coarsening phase:
 - set up the $n^i \times n^{i+1}$ prolongation matrix P^i
 - $G^{i+1} = P^{iT} G^i P^i$
 - $x^{i+1} = \text{MultilevelLayout}(G^{i+1}, \text{tol})$
- The prolongation and refinement phase:
 - prolongate to get initial layout: $x^i = P^i x^{i+1}$
 - refinement: $x^i = \text{ForceDirectAlgorithm}(G^i, x^i, \text{tol})$
 - return x^i

Algorithm 2. A multilevel force-directed algorithm.

In Algorithm 2, coarsening will stop if the graph is too small, $n^{i+1} < \text{MinSize}$, or there is not enough coarsening, $n^{i+1} / n^i > \rho$. We used $\text{MinSize} = 2$ and $\rho = 0.75$.

■ 6. Numerical Results

In this section we demonstrate the drawings using our algorithms on some large examples and also compare our algorithms with those of Walshaw [2], both in terms of efficiency and quality of layout.

The algorithms are implemented in *Mathematica* 5.1, under the GraphPlot package. To load the package, evaluate

```
<< DiscreteMath`GraphPlot`;
```

This package has many graph theory functions. Among them, `GraphPlot` and `GraphPlot3D` draw graphs in 2D and 3D, respectively. Details of the package can be found in the *Mathematica* Help Browser.

The details for the algorithms we will demonstrate and the names they are denoted by follow. We also give the *Mathematica* command corresponding to each of the algorithms, where we use `gr` to represent the graph.

MSE(r) (Multilevel Spring-Electrical Model)

Multilevel Algorithm 2 with HYBRID coarsening scheme, octree data structure, and repulsive/attractive force given by (1). The cutoff radius for the repulsive force calculation is (11), with $r = \infty$ by default. The *Mathematica* command for `MSE( $\infty$ )` is

```
GraphPlot[gr, Method→{SpringElectricalModel, Tolerance→0.01}]
```

The *Mathematica* command for `MSE( $r$ )` ($r < \infty$) is

```
GraphPlot[gr, Method→{SpringElectricalModel, Tolerance→0.01,
InferentialDistance→r}]
```

MSE(r, p) (Multilevel Spring-Electrical Model with General Repulsive Force)

Multilevel Algorithm 2 with HYBRID coarsening scheme, octree data structure, and repulsive/attractive force given by (1). The cutoff radius for the repulsive force calculation is (11), with $r = \infty$ by default. The difference from `MSE( $r$ )` is that the general repulsive force model (4) is used with parameter p . `MSE( $r$ )` is the same as `MSE( $r, 1$ )`. The *Mathematica* command for `MSE( $\infty, 1$ )` is

```
GraphPlot[gr, Method→{SpringElectricalModel, Tolerance→0.01}]
```

The *Mathematica* command for `MSE( $r, 1$ )` ($r < \infty$) is

```
GraphPlot[gr, Method→{SpringElectricalModel, Tolerance→0.01,
InferentialDistance→r}]
```

The *Mathematica* command for `MSE( $r, p$ )` ($r < \infty, p \neq 1$) is

```
GraphPlot[gr, Method→{SpringElectricalModel, Tolerance→0.01,
InferentialDistance→r, RepulsiveForcePower→p}]
```

MS(r) (Multilevel Spring Model)

Multilevel Algorithm 2 with HYBRID coarsening scheme, octree data structure, and repulsive/attractive force given by (5). The cutoff radius for the distance and force calculation is (12), with $r = 4$ by default. The *Mathematica* command for `MS( $\infty$ )` is

```
GraphPlot[gr, Method→{SpringModel, Tolerance→0.01}]
```

The *Mathematica* command for `MS( $r$ )` ($r < \infty$) is

```
GraphPlot[gr, Method→{SpringModel, Tolerance→0.01, InferentialDistance→
r}]
```

SE (Spring-Electrical Model)

Algorithm 1 with octree data structure and repulsive/attractive force given by (1). The *Mathematica* command for SE is

```
GraphPlot[gr, Method→{SpringElectricalModel, Tolerance→0.01,
RecursionMethod→None}]
```

MLFDP

The multilevel force-directed placement algorithm (MLFDP) from [2].

For all the preceding algorithms, we used a tolerance of $\text{tol} = 0.01$ to be comparable with Walshaw [2]. All our results are from a 3.0 GHz Pentium 4 machine with 2 GB of memory and a 512 KB cache under a Linux operating system. Our code is written in C and compiled with gcc using compilation flag “gcc -O2”.

□ 6.1. Comparison with Walshaw

In this section we compare our algorithm with that of Walshaw [2].

Table 1 lists a set of nine test problems from [2]. Some of these problems originate from engineering applications for which there is a known layout. Table 2 gives the CPU time for MSE(2) and MSE(∞), as well as the CPU time for the multilevel force-directed placement algorithm (MLFDP) from [2]. To see the extra cost of the multilevel approach, we also include the CPU time for the single level spring-electrical model, SE, in the last column of the table. The set of nine problems were chosen to be the same as those in Table 3 of [2]. We draw all the graphs in 2D and use these layouts for all the subsequent figures. However to be able to compare them with [2], we also laid out the last four graphs in 3D, even though *sierpinski10* is really a 2D graph and, as Figure 12 shows, we can layout in 2D just fine.

Notice that Walshaw’s CPU results were for a 1 GHz machine, 1/3 of the clock speed of our machine. However, based on our experience of clock speed and actual performance, we would expect the performance of our machine to be less than three times of Walshaw’s.

Walshaw’s MLFDP algorithm should be somewhat similar to our MSE(2), because both employ a similar cutoff radius. There are, however, some differences. MSE(2) uses the octree data structure and searches through the structure to decide if a cluster of vertices can form a supernode or should be excluded because it is outside the cutoff radius. So we have two approximations. In terms of CPU time, forming supernodes reduces the number of repulsive force calculations, but searching through the octree data structure is more expensive than the regular mesh-like data structures used by Walshaw to implement the cutoff radius. Therefore, overall, we expect the complexity of MLFDP and MSE(2) to be comparable. This is confirmed in Table 2. MLFDP and MSE(2) indeed do have quite comparable CPU time in general, although MSE(2) is notably faster on *finan512*, while MLFDP is notably faster on *dime20*. MSE(∞), which does not ignore long-range force, is only on average 21% more expensive than MSE(2).

Comparing multilevel algorithm $MSE(\infty)$ with its single level counterpart SE, it is seen that $MSE(\infty)$ is no more than twice as expensive, in fact for most graphs the difference is small. It is surprising that for *sierpinski10*, SE is more expensive than $MSE(\infty)$ for both the 2D and 3D layouts! A careful examination of the innermost loops of the force calculations and octree code reveals that $MSE(\infty)$ takes between two to three times as many operations as SE. The abnormality in CPU time is due to poor cache performance of the octree code in SE. SE starts from a random layout and never quite gets to a good layout for large graphs. When looping over vertices in the natural order to find their supernodes in the octree code, the locations of the vertices are unpredictable. The multilevel algorithm $MSE(\infty)$, on the other hand, always starts from a good initial layout. For most of the graphs in Table 1, the natural vertex ordering is such that if two vertices are close in their indices, they also tend to be close in their physical locations in the original layout. Thus in the octree code, roughly the same squares tend to be examined again and again when finding supernodes for consecutive vertices. This means that cache performance is very good for multilevel algorithm $MSE(\infty)$, but very poor for single level algorithm SE. This analysis is confirmed when we randomly shuffle the vertices. With such a random ordering, CPU timings for SE stay roughly the same, while the CPU timings for $MSE(\infty)$ increase about twice.

The preceding analysis may suggest that the multilevel algorithm is susceptible to poor initial vertex indexing. However, this is not true. First, large graphs tend to have good natural ordering. Second, poor ordering can be easily remedied by preordering the vertices using a suitable ordering algorithm that is inexpensive. For example, we used the METIS [10] nested dissection algorithm to order previously randomly shuffled graphs, using the adjacency matrix of the graphs, and then applied the multilevel algorithm to the resulting graphs. We found that the CPU timings of $MSE(\infty)$ on these preordered graphs are very comparable to those in Table 2. In fact for *dime20*, the CPU time is reduced from 290.6 to 252.3 with nested dissection preordering! Nested dissection preordering, however, has no effect on the cache performance of SE, due to its poor initial and subsequent layouts. It is possible to improve SE by ordering the vertices using a nested dissection based on the physical locations of the vertices, but since SE is not a good graph drawing algorithm anyway, we will not pursue this further here.

Graph	$ V $	$ E $	Avg. Degree	Diameter	Graph Type
c-fat500-10	500	46627	186.5	4	Random clique test
4970	4970	7400	3.	106	2 D dual
4 elt	15606	45878	5.9	102	2 D nodal
finan512	74752	261120	7.	87	Linear programming
dime20	224843	336024	3.	1179	2 D nodal
data	2851	15093	10.6	79	3 D nodal
add32	4960	9462	3.8	28	32-bit adder
sierpinski10	88575	177147	4.	1024	2 D fractal
mesh100	103081	200976	3.9	203	3 D dual

Table 1. Description of test problems.

	Size		CPU				
Graph	$ V $	$ E $	MLFDP*	MSE (2)	MSE (∞)	MS (4)	SE
2 D							
c-fat500-10	500	46627	5.6	0.37	0.37	0.82	0.2
4970	4970	7400	6.4	2.5	2.7	10.9	1.8
4 elt	15606	45878	24.3	9.4	11.7	102.9	9.2
finan512	74752	261120	363.8	56.6	59.8	3714.9	60.
dime20	224843	336024	264.3	195.5	290.6	1984.6	277.7
data	2851	15093	–	1.1	1.2	17.8	1.3
add32	4960	9462	–	3.1	3.3	44.4	2.6
sierpinski10	88575	177147	–	44.1	65.1	146.8	75.6
mesh100	103081	200976	–	91.6	109.4	5807.8	89.5
3 D							
data	2851	15093	6.6	2.3	2.4	33.	1.6
add32	4960	9462	12.5	6.2	7.1	230.7	3.2
sierpinski10	88575	177147	136.7	64.7	100.9	317.2	114.9
mesh100	103081	200976	431.1	158.	204.	6431.3	138.2

Table 2. CPU time (in seconds) for some force-directed algorithms. *: MLFDP data from [2], was for a 1 GHz Pentium III. All other times are for a 3 GHz Pentium 4. –: data not available.

As we would expect, MS(4) is very slow. This is because the spring model seeks to lay out vertices to have a physical distance equal to the graph distance. It is not obvious how to extend the octree methodology to the spring model. We can certainly work out the average graph distance of a cluster of vertices to another vertex, but to do so we still have to find the individual graph distances first. Therefore, no saving is achieved. A cutoff radius does allow us to reduce the $O(|V|^2)$ complexity. However, we found that for good drawing quality, we have to use a relatively large radius. This is probably because, unlike the spring-electrical model, the spring model does not have a strong repulsive force and with the cutoff radius the repulsive force is weakened further. At a cutoff

radius of 4, a large number of vertices are included, making the algorithm quite costly.

In terms of drawing quality, $MSE(2)$ performs comparably to MLFDP, with $MSE(\infty)$ the best. Both $MSE(2)$ and MLFDP ignore long-range forces. However, the multilevel process enables them to inherit global information from coarser graphs, thus in most cases both still give good quality drawings. Nevertheless, for some problems, the adverse effect can be seen.

□ 6.2. Comparison of Drawings

In the following, we give drawings of the graphs in Table 1. Our drawings of c-fat500-10 are the same as in [2] and are thus not included here. All our drawings are done in 2D, as we found that 2D drawings give us good representation.

Figure 7 (left) gives drawings of 4970 using $MSE(\infty)$. In this case the mesh around three corners is cluttered compared with the drawing in [2]. We believe this is due to the peripheral effect discussed in Section 3.1, which is reduced when there is a cutoff radius, as in Figure 10(b) of [2], and in $MSE(2)$ (middle). An alternative way to reduce the peripheral effect is to explicitly use a weaker repulsive force model (3), as the drawing by $MSE(\infty, 2)$ (right) shows.

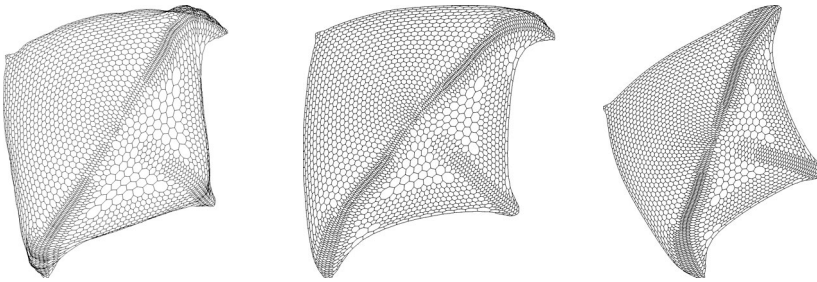


Figure 7. Drawings of 4970 by $MSE(\infty)$ (left), $MSE(2)$ (middle), and weaker repulsive force model $MSE(\infty, 2)$ (right).

Figure 8 (left) gives the drawing of finan512 using $MSE(\infty)$. This drawing is more appealing than Figure 13 of [2]. In that drawing, the circle is elongated, with the “knobs” flat and close to the circle. We believe this is due to the effect of ignoring the long-range repulsive force, so that the circle does not have enough force to make it rigid and rounded, and the knobs do not have enough force to push them out. We observed a similar side effect when we looked at the drawing given by $MSE(2)$ in Figure 8 (right), where the circle is twisted, although it does draw the knobs well.

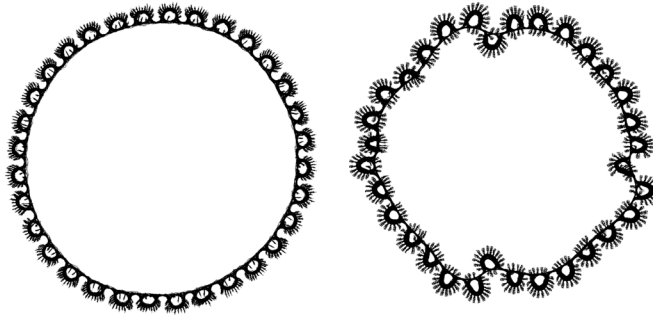


Figure 8. Drawings of finan512 by $MSE(\infty)$ (left) and $MSE(2)$ (right).

Figure 9 shows the drawings of dime20. The drawings are somewhat different from Figure 14(b) of [2]. In that drawing, the tip we see in Figure 9 at the top of the larger hole protrudes through the outer rim. Also, in [2], this large hole was replaced by a figure eight. Comparing the drawings of $MSE(\infty)$ and $MSE(2)$, the drawing by $MSE(2)$ has thicker outer rims, because of the weakened repulsive force, and thus reduced peripheral effect.

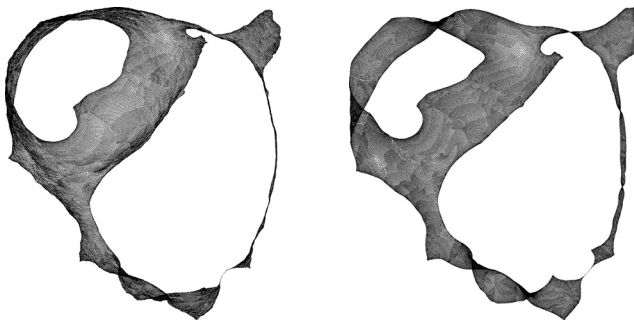


Figure 9. Drawings of dime20 by $MSE(\infty)$ (left) and $MSE(2)$ (right).

Figure 10 gives the drawings of add32. The drawings are different from Figure 16(a) in [2] in that they occupy a larger area. Comparing the drawings of $MSE(\infty)$ and $MSE(2)$, the latter is “fluffier” in that the branches extend to occupy more space. This is another example of the peripheral effect of a strong repulsive force in $MSE(\infty)$. We found that for tree-type applications, drawings by $MSE(\infty)$ tend to have leaves and some branches clinging to the main branches. $MSE(2)$ suffers less because of the weakened repulsive force due to the cutoff radius. For these type of applications, it is often better to apply the general repulsive force model (3) with a weaker force given by $p > 1$. Figure 11 shows drawings with $p = 2$ ($MSE(\infty, 2)$) and $p = 3$ ($MSE(\infty, 3)$). In our view they give more details.

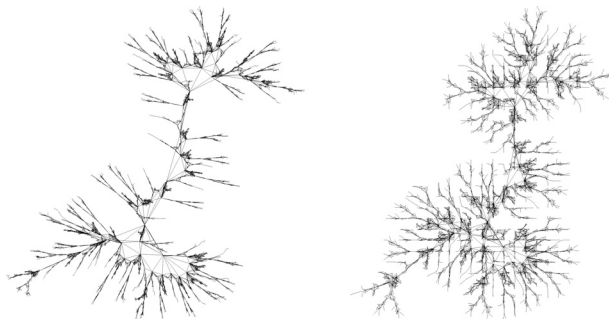


Figure 10. Drawings of add32 by $MSE(\infty)$ (left) and $MSE(2)$ (right).

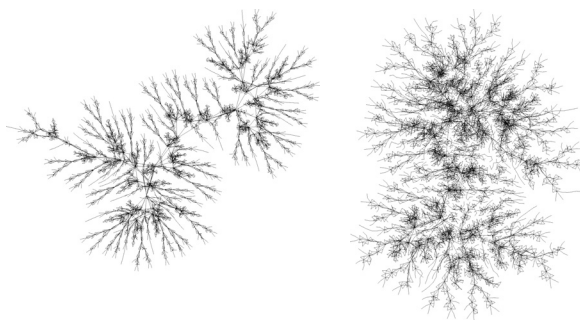


Figure 11. Drawings of add32 with a weaker repulsive force by $MSE(\infty, 2)$ (left) and $MSE(\infty, 3)$ (right).

Figure 12 gives the drawings of sierpinski10. In [2] Walshaw uses a 3D layout since he found 2D layout unsatisfactory. We found our 2D layout to be good, particularly $MSE(2)$. $MSE(\infty)$ demonstrates again the peripheral effect. The strong repulsive force pushes some of the vertices out. The bottom of Figure 12 shows the result of using a general repulsive force model (3) with weaker force given by $p = 2$, which does not suffer from the peripheral effect.

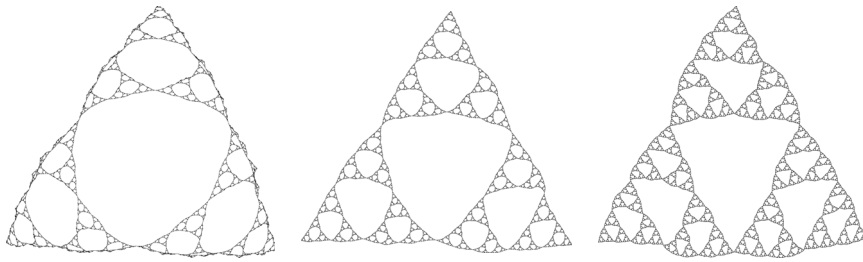


Figure 12. Drawings of *sierpinski10* by $MSE(\infty)$ (left), $MSE(2)$ (middle), and $MSE(\infty, 2)$ (right).

Figure 13 gives drawings of *mesh100*. Compared with the drawing in Figure 18(b) of [2], our drawings bear a closer resemblance to the actual mesh given in Figure 18(a) of [2].

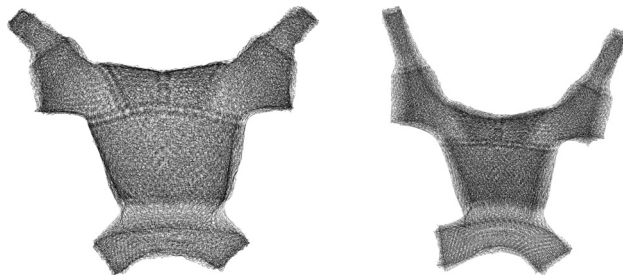


Figure 13. Drawings of *mesh100* by $MSE(\infty)$ (left) and $MSE(2)$ (right).

Overall, our algorithms give comparable drawings to those in [2]. For difficult graphs, notably *finan512* and *sierpinski10*, we achieve more appealing drawings. The general repulsive force model (3) offers choices to overcome the peripheral effect and can sometimes give more appealing drawings. As a further example, Figure 14 shows a good alternative drawing of *finan512*, by using a slightly weaker repulsive force. A comparison with Figure 8 (left), shows that without a very strong repulsive force to push them outward, some of the “spikes” now point inward. We found that further weakening of the repulsive force would cause the circle not to be rounded, much like what happens in Figure 8 (right).

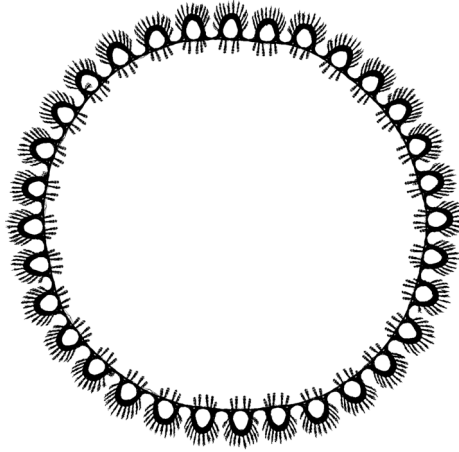


Figure 14. An alternative drawing of *finan512* by $MSE(\infty, 1.3)$.

□ 6.3. Comparing the Spring-Electrical Model with the Spring Model

In addition to efficiency considerations that favor the spring-electrical model, we also found that compared with the spring model, it gives good drawings for most graphs. The spring model, on the other hand, works particularly well for graphs originated from uniform or near-uniform meshes, albeit requiring more CPU time. The spring model works well for such graphs because it is possible to lay them out so that the physical distance is very close to the graph distance of vertices.

For example, Figure 4 (right) gives a drawing of *jagmesh1* using $MSE(\infty)$. It is seen that closer to the outer boundary, the peripheral effect of the spring-electrical model is obvious. This effect can be reduced using a weakened repulsive force, as Figure 15 (left) shows using $MSE(\infty, 2)$. However, the spring model, $MS(4)$, gives probably the most appealing drawing, as shown in Figure 15 (right). The same happens to *sierpinski10* (Figure 16) and *mesh100* (Figure 17). Both come very close to what the graphs look like in their original layout, in Figure 17(a) and Figure 18(a) of [2]. $MS(4)$ also draws the data graph quite well, compared with $MSE(\infty)$, as shown in Figure 18.

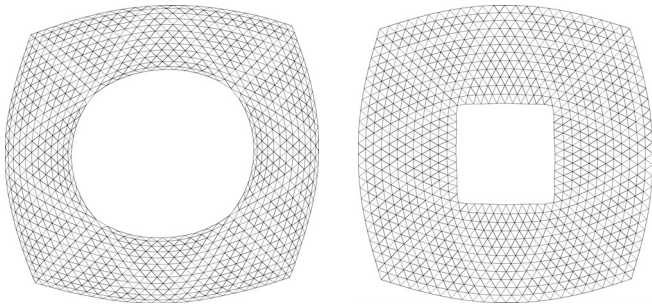


Figure 15. Alternative drawings of jagmesh1 of Figure 4 by $MSE(\infty, 2)$ (left) and $MS(4)$ (right).

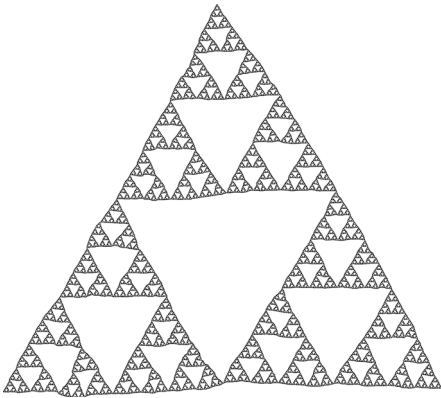


Figure 16. A drawing of sierpinski10 using $MS(4)$.

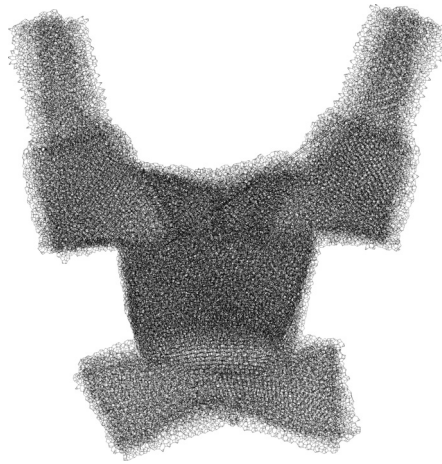


Figure 17. A drawing of mesh100 using MS(4).

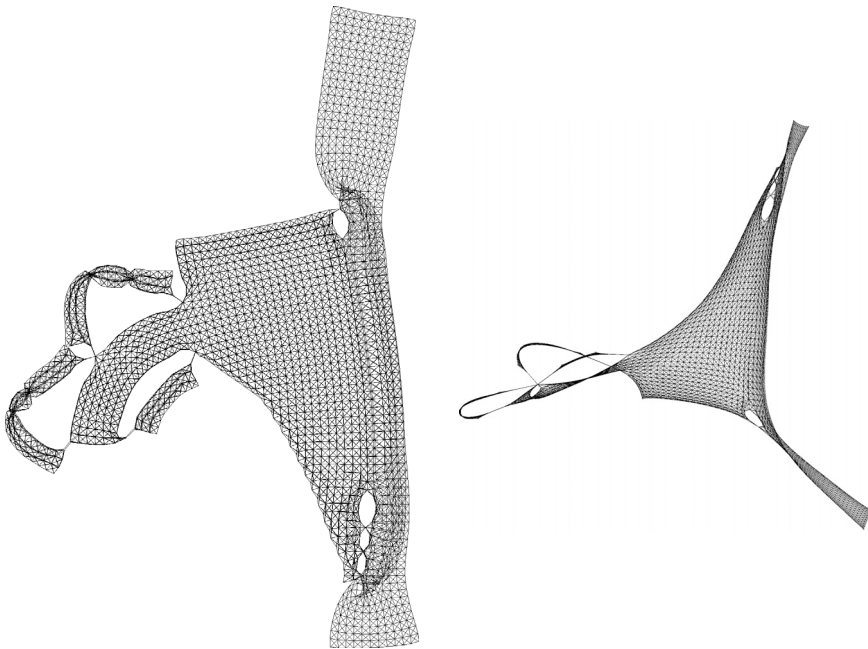


Figure 18. Drawings of data using MS(4) (left) and MSE(∞) (right).

However, for graphs that come from a locally refined mesh, the spring model works poorly. For example, Figure 19 shows the drawing of 4elt by MS(4) (right) and MSE(∞). It is clear that MS(4) strives to draw the graph as uniformly as

possible, but since this is not possible for such a highly refined graph, and in the absence of a strong long-range repulsive force, a lot of foldings occur near the highly refined regions. Therefore, overall we favor the multilevel SE algorithm for its efficiency and general good quality of drawings.

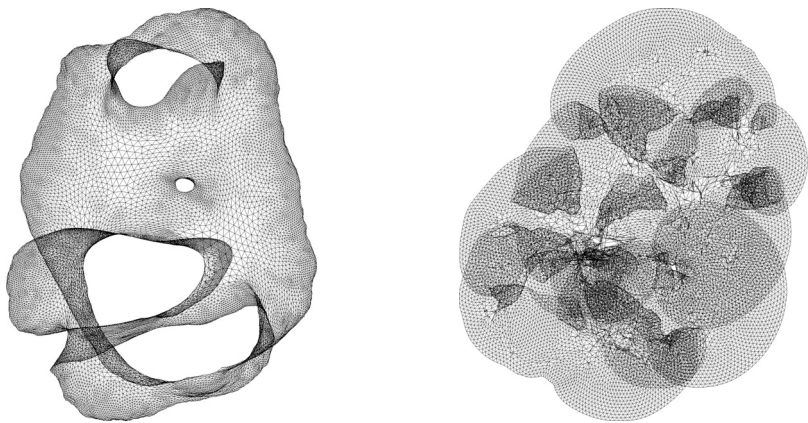


Figure 19. Drawings of 4elt by $MSE(\infty)$ (left) and $MS(4)$ (right).

□ **6.4. Further Examples**

In this section we demonstrate our algorithms with further examples from the University of Florida Sparse Matrix Collection (www.cise.ufl.edu/research/sparse/matrices). Three of the graphs (skirt, bodyy6 and pwt) have known layouts. Table 3 describes the graphs and the CPU time taken to lay these out in 2D.

Graph	$ V $	$ E $	Diameter	Graph Type	CPU
skirt	12598	91961	98 ¹	NASA matrix	8.4
bodyy6	19366	57421	122	NASA matrix	14.4
pwt	36519	144794	262 ²	NASA matrix	25.0
pkustk01	22044	478668	26	Beijing botanical exhibition hall	14.3
pkustk02	10800	399600	33	Feiyue twin tower building	9.5

Table 3. Problem description and CPU time (in seconds) for some graphs. ¹: skirt has seven components; four of them are nontrivial and have diameters 98, 68, 68, and 39, respectively. ²: pwt has 57 components, 56 components are just a single vertex.

Figure 20 shows the original layout of the skirt graph. It is surprising to us that this mesh actually consists of seven components; three of these are just an isolated vertex. Of the four nontrivial components, one is the main tube and nozzle, another is the skirt/struts at the bottom of Figure 20 (right). The other two

components are the two “rings” connecting the main tube/nozzle and the skirt. They are seen in Figure 20 (right) as one thick horizontal belt.

Figure 21 shows the drawings of the tube/nozzle and the skirt/struts. The tube/nozzle has a much finer mesh near the nozzle end, thus the drawing has the nozzle part expanded. The drawing of the skirt/struts is interesting. In the drawing, two struts protruding out of Figure 21 (left) are drawn separated from three pieces of the skirt, revealing the weak linkage between the struts and the pieces of skirts.

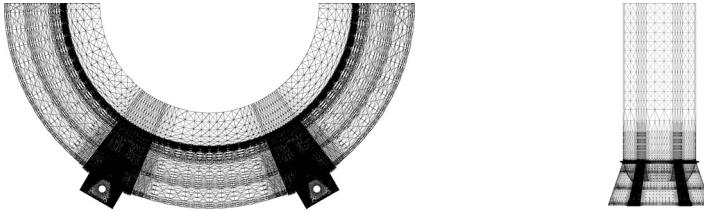


Figure 20. Original layout of skirt: two views.

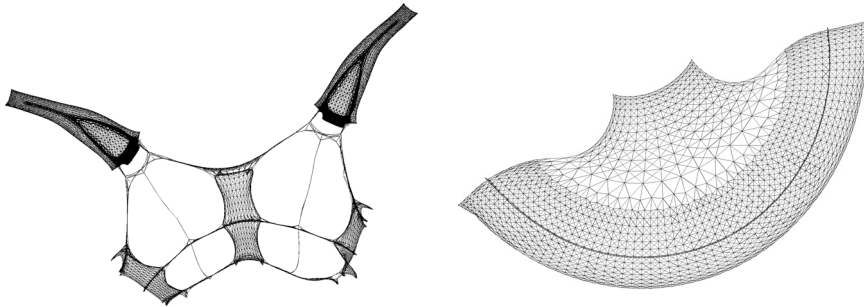


Figure 21. Drawings of two components of skirt by $MSE(\infty)$: the skirt/struts (left); the tube/nozzle (right).

Figure 22 (left) shows the original layout of a highly refined mesh. The mesh is highly refined around the middle void and to its right. The drawing algorithm, in its effort to draw edges as uniformly as possible, turned the mesh inside out (Figure 22, right). The hole in the middle is actually the middle void in the original mesh. The original mesh is so highly refined to the right of the middle void that the drawing shows a folding. However, contrary to our intuition, using algorithms $MSE(\infty, 2)$ or $MSE(2)$, both having a weaker repulsive force, removes the folding (Figure 23). So it seems the folding is due to the strong repulsive force of the spring-electrical model, another example of the peripheral effect. The original mesh is nearly symmetric and all the drawings also exhibit good symmetry.

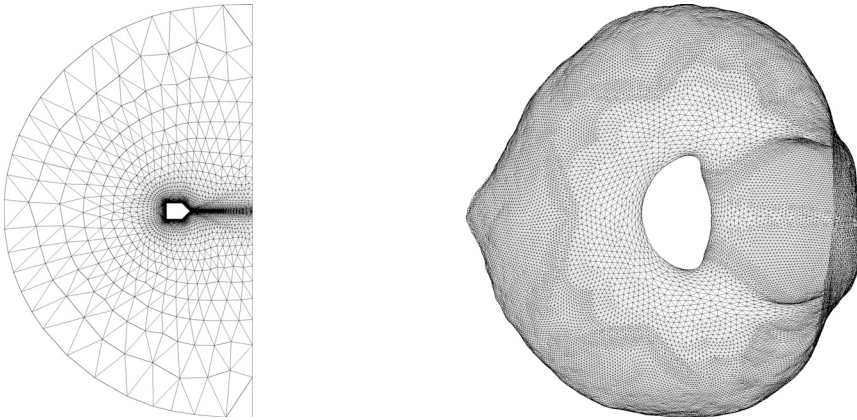


Figure 22. Original layout of bodyy6 (left) and drawing by $MSE(\infty)$ (right).

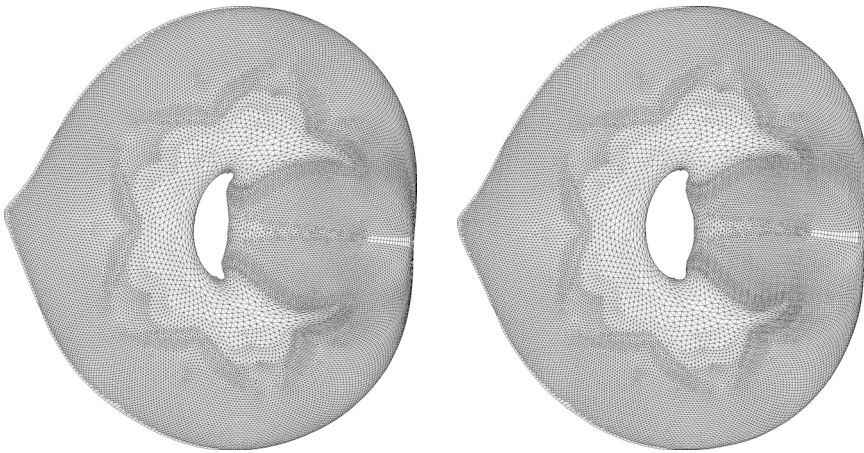


Figure 23. Drawings of bodyy6 using $MSE(\infty, 2)$ (left) and $MSE(2)$ (right).

Figure 24 (left) shows the pwt mesh, which is probably a mesh for a pressured wind tunnel. The drawing by $MSE(\infty)$ corresponds to the original layout well. The large chamber in the original mesh has a mesh density similar to the pipe and is thus indistinguishable from the pipe in the drawing. In Figure 25 close-up views of the smaller chamber in the original layout and our corresponding drawing are given. The drawing depicts the details well.

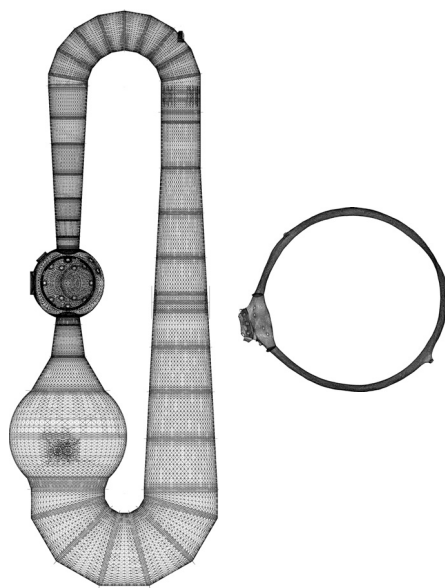


Figure 24. Original layout of pwt (left) and a drawing by $MSE(\infty)$ (right).

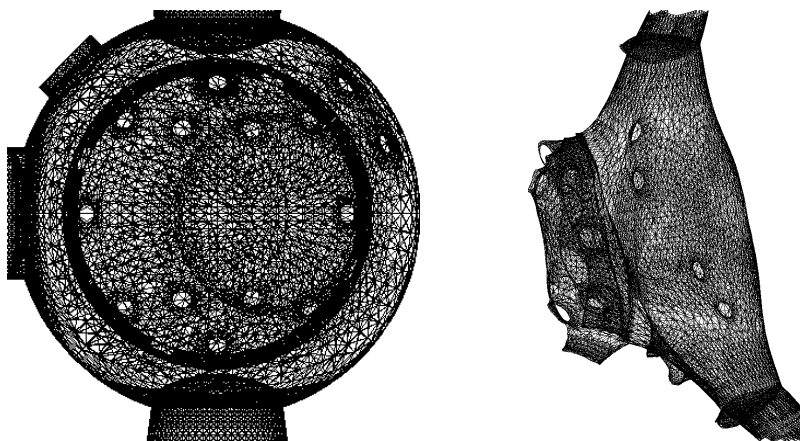


Figure 25. Close-up view of the original layout of pwt (left) and a drawing by $MSE(\infty)$ (right).

Finally, Figure 26 shows drawings of pkustk01 and pkustk02. These two graphs have high average degrees (43 and 74, respectively). However, judging by the drawings, our algorithms performed very well.

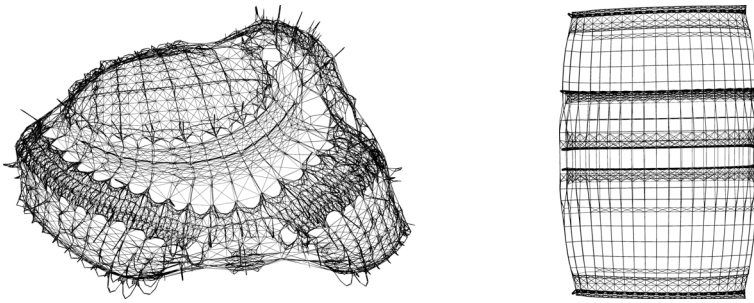


Figure 26. Drawings of pkustk01 (left) and pkustk02 (right) using $MSE(\infty)$.

■ 7. Conclusions and Future Work

In this article we proposed an algorithm that uses a multilevel approach to find global optimal layouts and the octree technique to approximate short- and long-range forces satisfactorily and efficiently. These two techniques were each proposed earlier for graph drawing [2, 16, 18], but as far as we are aware, were never combined to form one powerful algorithm for large-scale graph drawing. A number of practical techniques, including adaptive step and octree depth control, and a hybrid coarsening scheme, were introduced for the algorithm to work effectively. This algorithm is demonstrated to be both efficient and of high quality for large graphs, competitive to Walshaw's [2] highly successful graph drawing algorithm, yet gives better drawings on some difficult problems.

We also proposed a general repulsive force model to overcome the peripheral effect of the spring-electrical model. Finally, we compared the spring-electrical model with the spring model and demonstrated examples where the spring model may be suitable.

Both the multilevel approach and the octree data structure do have limitations. For example, both the EC coarsening scheme and the MIVS-based coarsening scheme would not work effectively on star graphs (a graph with one vertex connected to all other vertices and no two other vertices connected). The former would coarsen too slowly thus having unacceptable complexity, while the latter would coarsen too fast and not preserve the graph information on the coarser graphs. The parameter θ in the Barnes and Hut octree algorithm is empirically fixed (to 1.2 in all our drawings). We experienced one case where this value gives an artifact only corrected with a smaller value $\theta = 0.8$. It may be possible to correct this artifact without changing the θ value by adding a random offset to the first square of the octree. These limitations remain a topic for further investigations. However, in general the proposed algorithm performed extremely well on a range of graphs from different application areas, a small number of which were shown.

The graph drawing algorithm presented in this article is in the *Mathematica* 5.1 release (November 2004).

After the completion of this article, our attention was drawn to an independent work by Hachul and Jünger [28]. In that paper, the multilevel approach is combined with the multipole expansion technique [29] to give a $O(|V| \log(|V|) + |E|)$ algorithm. The efficiency and quality of the algorithm in that paper appear to be at the same level as this article, although many details differ, including the multilevel scheme and the force approximation.

■ Acknowledgments

The author would like to thank Chris Walshaw for providing the graphs in Table 1 and Andrew A. de Laix for bringing space decomposition techniques to my attention.

■ References

- [1] J. Barnes and P. Hut, "A Hierarchical $O(n \log n)$ Force-Calculation Algorithm," *Nature*, **324**(4), 1986 pp. 446–449.
- [2] C. Walshaw, "A Multilevel Algorithm for Force-Directed Graph Drawing," *Journal of Graph Algorithms and Applications*, **7**(3), 2003 pp. 253–285.
- [3] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*, Englewood Cliffs, NJ: Prentice Hall, 1999.
- [4] T. M. J. Fruchterman and E. M. Reigold, "Graph Drawing by Force-Directed Placement," *Software—Practice and Experience*, **21**(11), 1991 pp. 1129–1164.
- [5] P. Eades, "A Heuristic for Graph Drawing," *Congressus Numerantium*, **42**, 1984 pp. 149–160.
- [6] N. R. Quinn and M. A. Breuer, "A Force Directed Component Placement Procedure for Printed Circuit Boards," *IEEE Transactions on Circuits and Systems*, **CAS-26**(6), 1979 pp. 377–388.
- [7] T. Kamada and S. Kawai, "An Algorithm for Drawing General Undirected Graphs," *Information Processing Letters*, **31**(1), 1989 pp. 7–15.
- [8] Y. Koren, L. Carmel, and D. Harel, "Drawing Huge Graphs by Algebraic Multigrid Optimization," *Multiscale Modeling and Simulation*, **1**(4), 2003 pp. 645–673.
- [9] D. Harel and Y. Koren, "Graph Drawing by High-Dimensional Embedding," *Journal of Graph Algorithms and Applications*, **8**(2), 2004 pp. 195–214.
- [10] G. Karypis and V. Kumar, "Multilevel k -way Partitioning Scheme for Irregular Graphs," *Journal of Parallel and Distributed Computing*, **48**(1), 1998 pp. 96–129.
- [11] B. Hendrickson and R. Leland, "A Multilevel Algorithm for Partitioning Graphs," *Technical Report SAND93-1301*, Albuquerque, NM: Sandia National Laboratories, 1993. Also in *Proceeding of Supercomputing'95 (SC95)*, San Diego, CA www.supercomp.org/sc95/proceedings/509_BHEN/SC95.HTM.
- [12] C. Walshaw, M. Cross, and M. G. Everett, "Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes," *Journal of Parallel and Distributed Computing*, **47**(2), 1997 pp. 102–108.
- [13] D. Harel and Y. Koren, "A Fast Multi-Scale Method for Drawing Large Graphs," *Journal of Graph Algorithms and Applications*, **6**(3), 2002 pp. 179–202.
- [14] R. Hadany and D. Harel, "A Multi-Scale Algorithm for Drawing Graphs Nicely," *Discrete Applied Mathematics*, **113**(1), 2001 pp. 3–21.

- [15] P. Gajer, M. T. Goodrich, and S. G. Kobourov, "A Fast Multi-Dimensional Algorithm for Drawing Large Graphs," *Lecture Notes in Computer Science*, **1984**, New York: Springer-Verlag, 2000 pp. 211–221.
- [16] D. Tunkelang, "A Numerical Optimization Approach to General Graph Drawing," Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1999.
- [17] A. Quigley and P. Eades, "Fade: Graph Drawing, Clustering, and Visual Abstraction," *Lecture Notes in Computer Science*, **1984**, New York: Springer-Verlag, 2000 pp. 183–196.
- [18] A. Quigley, "Large Scale Relational Information Visualization, Clustering, and Abstraction," Ph.D. thesis, Department of Computer Science and Software Engineering, University of Newcastle, Australia, 2001.
- [19] R. Davison and D. Harel, "Drawing Graphs Nicely Using Simulated Annealing," *ACM Transactions on Graphics*, **15**(4), 1996 pp. 301–331.
- [20] R. Fletcher, *Practical Methods of Optimization*, 2nd ed., New York: John Wiley & Sons, 2000.
- [21] K. J. Pulo, "Recursive Space Decompositions in Force-Directed Graph Drawing Algorithms," in *Proceedings of the Australian Symposium on Information Visualisation (InVis.au 2001)*, Sydney, Australia (P. Eades and T. Pattison, eds.), *Conferences in Research and Practice in Information Technology Series*, **9**, Darlinghurst, Australia: Australian Computer Society, 2001 pp. 95–102.
- [22] S. Pfalzner and P. Gibbon, *Many-Body Tree Methods in Physics*, New York: Cambridge University Press, 1996.
- [23] A. Gupta, G. Karypis, and V. Kumar, "Highly Scalable Parallel Algorithms for Sparse Matrix Factorization," *IEEE Transactions on Parallel and Distributed Systems*, **8**(5), 1997 pp. 502–520.
- [24] Y. F. Hu and J. A. Scott, "A Multilevel Algorithm for Wavefront Reduction," *SIAM Journal on Scientific Computing*, **23**(4), 2001 pp. 1352–1375.
- [25] C. Walshaw, "A Multilevel Approach to the Travelling Salesman Problem," *Operations Research*, **50**(5), 2002 pp. 862–877.
- [26] C. Walshaw, "Multilevel Refinement for Combinatorial Optimisation Problems," *Annals of Operations Research*, **131**, 2004 pp. 325–372; originally published as *University of Greenwich Technical Report 01/IM/73*, 2001.
- [27] S. T. Barnard and H. D. Simon, "A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems," *Concurrency: Practice and Experience*, **6**(2), 1994 pp. 101–117.
- [28] S. Hachul and M. Jünger, "Drawing Large Graphs with a Potential-Field-Based Multilevel Algorithm," in *Proceedings of the Twelfth International Symposium on Graph Drawing (GD 2004)*, New York (J. Pach, ed.), *Lecture Notes in Computer Science*, **3383**, New York: Springer-Verlag, 2004 pp. 285–295.
- [29] L. F. Greenguard, *The Rapid Evaluation of Potential Fields in Particle Systems*, ACM Distinguished Dissertations Series, Cambridge, MA: The MIT Press, 1988.

Yifan Hu

Senior Developer
 Wolfram Research, Inc.
 100 Trade Center Drive
 Champaign, IL 61820, USA
 yifanhu@wolfram.com

Ordered and Unordered Factorizations of Integers

Arnold Knopfmacher
Michael Mays

We study the number of ways of writing a positive integer n as a product of integer factors greater than one. We survey methods from the literature for enumerating and also generating lists of such factorizations for a given number n . In addition, we consider the same questions with respect to factorizations that satisfy constraints, such as having all factors distinct. We implement all these methods in *Mathematica* and compare the speeds of various approaches to generating these factorizations in practice.

■ Introduction

To study the number of ways of writing a positive integer n as a product of integer factors greater than one, there are two basic cases to consider. First, we can regard rearrangements of factors as different. In the case of $n = 12$, this gives the following eight ordered factorizations.

$$\{\{2, 2, 3\}, \{2, 3, 2\}, \{2, 6\}, \{3, 2, 2\}, \{3, 4\}, \{4, 3\}, \{6, 2\}, \{12\}\}$$

Alternatively we can ignore the order of the factors, which then gives the following four unordered factorizations.

$$\{\{3, 2, 2\}, \{4, 3\}, \{6, 2\}, \{12\}\}$$

These two functions, which we denote by $H(n)$ and $P(n)$, respectively, can be considered multiplicative analogs of compositions and partitions of integers. A composition is an ordered set of positive integers that sum to n . For example, we have eight compositions of $n = 4$, namely $\{\{4\}, \{3,1\}, \{1,3\}, \{2,2\}, \{2,2,1\}, \{1,2,1\}, \{1,1,2\}, \{1,1,1,1\}\}$. In general the number of compositions of n , $C(n)$, is equal to 2^{n-1} . A partition is a set that sums to n in which order is disregarded. There are five partitions of four. To count them we can use the function `PartitionsP`, and to list them we can use `Partitions` from the *Combinatorica* package.

```
In[1]:= << DiscreteMath`Combinatorica`
```

```
In[2]:= Partitions[4]
```

```
Out[2]= {{4}, {3, 1}, {2, 2}, {2, 1, 1}, {1, 1, 1, 1}}
```

```
In[3]:= PartitionsP[Range[10]]
```

```
Out[3]= {1, 2, 3, 5, 7, 11, 15, 22, 30, 42}
```

In fact the number of compositions and partitions arise as special cases of our factorization functions, in the sense that if p is a prime number, then $H(p^r) = C(r) = 2^{r-1}$ and $P(p^r) = \text{PartitionsP}[r]$. For general numbers n with prime factorizations $n = p_1^{r_1} p_2^{r_2} \dots p_k^{r_k}$, the enumeration of $H(n)$ and $P(n)$ is more complicated, as we shall discuss later.

We will also discuss factorizations into distinct parts. In the ordered case, $H_d(12) = 5$ since we have the factorizations $\{\{2,6\}, \{3,4\}, \{4,3\}, \{6,2\}, \{12\}\}$. In the unordered case, $P_d(12) = 3$ since there are three cases, $\{\{4,3\}, \{6,2\}, \{12\}\}$. If p is a prime number, then as special cases we have $H_d(p^r) = C_d(r)$ and $P(p^r) = \text{PartitionsQ}[r]$, where $C_d(r)$ denotes the number of compositions of r into distinct parts (see Richmond and Knopfmacher [1]), and PartitionsQ is the *Mathematica* function for counting partitions into distinct parts.

```
In[4]:= PartitionsQ[Range[10]]
```

```
Out[4]= {1, 1, 2, 2, 3, 4, 5, 6, 8, 10}
```

The enumeration and generation of integer partitions and compositions are problems discussed in standard books on combinatorics. A definitive reference is Andrews [2]. However, the corresponding problems for ordered and unordered factorizations have not until now received a comprehensive treatment.

■ Ordered Factorizations

For historical reasons, we will discuss formulas to enumerate factorizations before we discuss methods to generate the corresponding factorizations. In addition, some of the recursive methods to generate factorizations are extensions of the corresponding recursions to enumerate factorizations.

□ Recursions for Enumerating Ordered Factorizations

We begin with two recurrence formulas given by Hille [3]. The first element of an ordered factorization of $n > 1$ can be any number d such that d divides n . By appending to d all possible ordered factorizations of n/d , we obtain the recursion $H(1) = 1$; $H(n) = \sum_{d|n} H(d)$ for $n \geq 2$. We implement this as follows.

```
In[5]:= H1[1] := 1;
        H1[n_] := H1[n] = Total[H1 /@ Drop[Divisors[n], -1]]
```

```
In[7]:= Table[H1[n], {n, 1, 12}]
```

```
Out[7]= {1, 1, 1, 2, 1, 3, 1, 4, 2, 3, 1, 8}
```

Using the Möbius inversion formula, Hille also derived a second recursion,

$$H(n) = 2 \left(\sum_{p_i} H\left(\frac{n}{p_i}\right) - \sum_{p_i, p_j} H\left(\frac{n}{p_i p_j}\right) + \cdots + (-1)^{k-1} H\left(\frac{n}{p_1 p_2 \cdots p_k}\right) \right),$$

where $n = p_1^{r_1} p_1^{r_1} p_2^{r_2} \cdots p_k^{r_k}$, which holds for $n \geq 2$.

This finds the list of distinct prime factors of n .

```
In[8]:= PrimeFactorList[n_] := First /@ FactorInteger[n]
```

This recursion requires the initial value $\frac{1}{2}$.

```
In[9]:= H2[1] = 1 / 2;
```

The recursion can be rendered elegantly as a oneliner.

```
In[10]:= H2[n_] := H2[n] = -2 Total[(-1)^(Length[#]) H2[n/Times @@ #] & /@
Rest[Subsets[PrimeFactorList[n]]]]
```

```
In[11]:= Table[H2[n], {n, 2, 12}]
```

```
Out[11]= {1, 1, 2, 1, 3, 1, 4, 2, 3, 1, 8}
```

□ MacMahon's Formula for H

MacMahon [4] derived an explicit formula for the value of $H(n)$ as a double sum over a product. Given $n = p_1^{r_1} p_1^{r_1} p_2^{r_2} \cdots p_k^{r_k}$,

$$H(n) = \sum_{j=1}^q \sum_{i=0}^{j-1} (-1)^i \binom{j}{i} \prod_{b=1}^k \binom{r_b + j - i - 1}{r_b},$$

where $q = \sum_{i=1}^k r_i$, is the sum of the prime exponents of n . We program this by

```
In[12]:= H3[n_] :=
```

```
  Total[##] \sum_{j=1}^j \sum_{i=0}^{j-1} (-1)^i Binomial[j, i] Apply[Times,
```

```
    Binomial[# + j - i - 1, #] & /@ #] & [Last /@ FactorInteger[n]]
```

```
In[13]:= H3[2^5 3^4 5] // Timing
```

```
Out[13]= {0. Second, 102576}
```

□ A Recursion for Factorizations with Distinct Parts

Warlimont [5] derived a Dirichlet series generating function for the function $H_d(k, n)$, which denotes the number of ordered factorizations of n into k distinct parts. Although Warlimont was only interested in this for asymptotic purposes, his generating function can be used to derive the following recurrence:

$$H_d(k+1, n) = k! \sum_{j=0}^k \frac{(-1)^j}{(k-1-j)!} \sum H_d\left(k-j, \frac{n}{d}\right),$$

where the inside sum is taken over all d such that $d \mid n$ and for $d \geq 2$, d is a $(j+1)$ -st power.

There does not seem to be a simple combinatorial interpretation for this formula. We implement this starting with the appropriate boundary conditions.

```
In[14]:= Hd[1, n_] := 1;
          Hd[k_, 1] := 0;
          Hd[0, n_] := 0
```

```
In[17]:= Hd[0, 1] := 1;
          Hd[1, 1] := 0;
```

When n is a prime number we have the following.

```
In[19]:= Hd[k_, n_?PrimeQ] := KroneckerDelta[k, 1]
```

Also observe that the number of parts k must satisfy $2^k \leq n$.

```
In[20]:= Hd[k_, n_ /; 2k > n] := 0
```

Now we implement the general formula.

```
In[21]:= Hd[k_, n_] :=
          Hd[k, n] = (k - 1)! ∑j=0k-1 ((-1)j / (k - 1 - j)! Total[Hd[k - 1 - j, n/#] & /@
          Select[Rest[Divisors[n]], IntegerQ[#1/(j+1)] &]])
```

```
In[22]:= Hd[#, 24] & /@ Range[0, 4]
```

```
Out[22]= {0, 1, 6, 6, 0}
```

We verify these counts by using the function `DistinctOrderedFactorizations`, which is defined in the next subsection.

```
In[23]:= DistinctOrderedFactorizations[24]
```

```
Out[23]= {{2, 3, 4}, {2, 4, 3}, {2, 12}, {3, 2, 4}, {3, 4, 2}, {3, 8},
          {4, 2, 3}, {4, 3, 2}, {4, 6}, {6, 4}, {8, 3}, {12, 2}, {24}}
```

To obtain the total number of distinct ordered factorizations, we must sum $H_d[k, n]$ over all permissible values of the number of parts k .

```
In[24]:= Hd[n_] := Total[Hd[#, n] & /@ Range[Floor[Log[2, n]]]]
```

```
In[25]:= Hd[36]
```

```
Out[25]= 13
```

In practice this recursion is slow. A faster counting method is discussed in a later section.

□ Generating Ordered Factorizations

The first recursion for $H(n)$ suggests a natural recursive approach to generate all the ordered factorizations of n .

We append to the first factor d in a factorization of $n > 1$, all possible ordered factorizations of $\frac{n}{d}$, where d can be any divisor of n .

```
In[26]:= OrderedFactorizations[1] = {{}};

In[27]:= OrderedFactorizations[n_?PrimeQ] := {{n}}

In[28]:= OrderedFactorizations[n_] :=
  OrderedFactorizations[n] = Flatten[Function[d, Prepend[#, d] & /@
    OrderedFactorizations[n/d]] /@ Rest[Divisors[n]], 1]

In[29]:= OrderedFactorizations[24]

Out[29]= {{2, 2, 2, 3}, {2, 2, 3, 2}, {2, 2, 6}, {2, 3, 2, 2}, {2, 3, 4}, {2, 4, 3},
  {2, 6, 2}, {2, 12}, {3, 2, 2, 2}, {3, 2, 4}, {3, 4, 2}, {3, 8},
  {4, 2, 3}, {4, 3, 2}, {4, 6}, {6, 2, 2}, {6, 4}, {8, 3}, {12, 2}, {24}}
```

One way to list all the ordered factorizations with distinct parts is to simply select these from the list of all ordered factorizations.

```
In[30]:= DistinctOrderedFactorizations[n_] :=
  Select[OrderedFactorizations[n], Unequal @@ # &]

In[31]:= DistinctOrderedFactorizations[24]

Out[31]= {{2, 3, 4}, {2, 4, 3}, {2, 12}, {3, 2, 4}, {3, 4, 2}, {3, 8},
  {4, 2, 3}, {4, 3, 2}, {4, 6}, {6, 4}, {8, 3}, {12, 2}, {24}}
```

However, there are faster methods for doing this, which we discuss in a later section.

■ Unordered Factorizations

There do not appear to be any explicit formulas for $P(n)$ in the literature. Also, the recurrence relations that are known tend to lack simple combinatorial interpretations.

□ A Product Recursion

Harris and Subbarao [6] give the following product recursion for $P(n)$. For any positive integer a , let $d_i = a^{1/i}$ if a is an i th power and $d_i = 1$ otherwise. Let $\bar{d} = \prod_{i=1}^{\infty} d_i$. This gives $\prod_{d|n} \bar{d}^{p(n/d)} = n^{p(n)}$. To make use of this, we take logs of the recurrence. First, we define the $\bar{d}$ values in terms of a given positive integer a . One approach is to use Product.

```
In[32]:= d1[a_] := Product[Ceiling[Log[2,a]]
  If[IntegerQ[c = a^(1/i)], c, 1], {i, 1}]
```

Alternatively, here is a more elegant construction.

```
In[33]:= d2[a_] := Apply[Times, Select[a^(1/Range[Ceiling[Log[2,a]]], IntegerQ]]
```

Then we can define P2.

```
In[34]:= P2[1] = 1;
```

```
In[35]:= P2[n_] := P2[n] = Round[1 / (Log[n])
Simplify[Total[Log[d2[#]] P2[n/#] & /@ Rest[Divisors[n]]]]]
```

The use of Simplify in P2 is not required, but speeds up the overall computation. Round produces the correct integer value for P2 much faster than by using additional simplification methods to remove the logarithms.

```
In[36]:= P2[33 23] // Timing
```

```
Out[36]:= {0.078Second, 31}
```

□ A Recursion for Unordered Factorizations with Largest Part m

Let $g(m, n)$ denote the number of unordered factorizations of n with largest part at most m . Hughes and Shallit [7] gave the recursion

$$g(m, n) = \sum_{\substack{d|n \\ d \leq m}} g\left(d, \frac{n}{d}\right).$$

This particular recursion is easy to explain: Let $n = a_1 a_2 \dots a_n$ be an unordered factorization of n with parts at most m and parts arranged in decreasing order, so that the largest part is a_1 . The number of ways to choose $a_2, \dots, a_k$ is then $g(a_1, n/a_1)$. For a_1 we can choose any divisor d of n such that $d \leq m$. Summing over all such d gives the result. We implement this as follows.

```
In[37]:= g[m_, 1] := 1;
g[1, n_] := 0;
g[1, 1] = 1;
g[m_, n_] :=
g[m, n] = Total[g[#, n/#] & /@ Select[Divisors[n], # ≤ m &]]
```

All unordered factorizations are counted by $g(n, n)$.

```
In[41]:= P1[n_] := g[n, n]
```

```
In[42]:= P1 /@ Range[12]
```

```
Out[42]:= {1, 1, 1, 2, 1, 2, 1, 3, 2, 2, 1, 4}
```

This gives a much faster recursion than P2.

```
In[43]:= P1[36 24] // Timing
```

```
Out[43]:= {0.016Second, 323}
```

```
In[44]:= P2[36 24] // Timing
```

```
Out[44]:= {0.422Second, 323}
```

Canfield, Erdős, and Pomerance [8] remarked in their paper that it is not particularly easy to compute $P(n)$. They mention that even showing $P(1800) = 137$ takes some work. Their approach was based on a tree traversal algorithm. With our recursion this computation presents no problem.

```
In[45]:= P1[1800] // Timing
```

```
Out[45]:= {0.015Second, 137}
```

□ Generating Unordered Factorizations

One inefficient approach is simply to sort the larger list of ordered factorizations and remove duplicates. However, a better approach is to use the Hughes–Shallit idea to recursively build up a list of ordered factorizations with largest part at most m .

```
In[46]:= UnorderedFactorizations[m_, 1] = {};
UnorderedFactorizations[1, n_] = {};

In[48]:= UnorderedFactorizations[m_, n_ /; PrimeQ[n]] := If[m < n, {}, {{n}}]

In[49]:= UnorderedFactorizations[m_, n_] :=
  UnorderedFactorizations[m, n] = Flatten[
    Function[d, Prepend[#, d] & /@ UnorderedFactorizations[d, n/d]] /@
    Rest[Select[Divisors[n], # ≤ m &]], 1]

In[50]:= UnorderedFactorizations[n_] := UnorderedFactorizations[n, n]
```

Now we test this out.

```
In[51]:= UnorderedFactorizations[24]

Out[51]:= {{3, 2, 2, 2}, {4, 3, 2}, {6, 2, 2}, {6, 4}, {8, 3}, {12, 2}, {24}}

In[52]:= Length[UnorderedFactorizations[34 24]] == P1[34 24]

Out[52]:= True
```

□ Unordered Factorizations with Distinct Parts

A modification of Hughes–Shallit reasoning gives a recursion for unordered factorizations with distinct parts and largest part at most m . We merely observe that the part added to d should have largest part less than or equal to $d - 1$:

$$\text{gd}(m, n) = \sum_{\substack{d|n \\ d \leq m}} \text{gd}\left(d - 1, \frac{n}{d}\right).$$

We program this with necessary boundary conditions to start the recursion.

```
In[53]:= gd[m_, 1] := 1;
gd[1, n_] := 0;
gd[1, 1] = 1;
gd[0, n_] := 0;
gd[m_, n_] :=
  gd[m, n] = Total[gd[# - 1, n/#] & /@ Select[Divisors[n], # ≤ m &]]

In[58]:= P_d[n_] := gd[n, n]
```

Here is an example.

```
In[59]:= P_d[34 22 5 7] // Timing

Out[59]:= {0.016 Second, 253}
```

Generating Unordered Factorizations with Distinct Parts

Again we can simply select the permissible factorizations from the larger list of all unordered factorizations, but a better approach is to recursively generate them using the idea for the recursion for $\text{gd}(m, n)$.

```
In[60]:= DistinctUnorderedFactorizations[m_, 1] = {{}};
          DistinctUnorderedFactorizations[1, n_] = {};
          DistinctUnorderedFactorizations[0, n_] = {};

In[63]:= DistinctUnorderedFactorizations[m_, n_ /; PrimeQ[n]] :=
          If[m < n, {}, {{n}}]

In[64]:= DistinctUnorderedFactorizations[m_, n_] :=
          DistinctUnorderedFactorizations[m, n] = Flatten[Function[d,
              Prepend[#, d] & /@ DistinctUnorderedFactorizations[d - 1, n / d]] /@
              Rest[Select[Divisors[n], # ≤ m &]], 1]

In[65]:= DistinctUnorderedFactorizations[n_] :=
          DistinctUnorderedFactorizations[n, n]
```

Here are the distinct unordered factorizations of 24.

```
In[66]:= DistinctUnorderedFactorizations[24]

Out[66]= {{4, 3, 2}, {6, 4}, {8, 3}, {12, 2}, {24}}
```

Here are the nondistinct factorizations.

```
In[67]:= Complement[UnorderedFactorizations[24], %]

Out[67]= {{6, 2, 2}, {3, 2, 2, 2}}
```

We check that we are generating the right number of cases.

```
In[68]:= P_d[3^4 2^4] == Length[DistinctUnorderedFactorizations[3^4 2^4]]

Out[68]= True
```

□ Faster Generation of Ordered Factorization Lists

The lists of unordered factorizations constructed earlier lead to a much faster way of generating the corresponding lists of ordered factorizations. We merely observe that all ordered cases arise as permutations of unordered cases. This gives a different ordering of the factorizations to the earlier method. However, the lists are easily checked to be the same.

```
In[69]:= OrderedFactorizations2[n_] :=
          Flatten[Permutations /@ UnorderedFactorizations[n], 1]

In[70]:= OrderedFactorizations2[24]

Out[70]= {{3, 2, 2, 2}, {2, 3, 2, 2}, {2, 2, 3, 2},
          {2, 2, 2, 3}, {4, 3, 2}, {4, 2, 3}, {3, 4, 2}, {3, 2, 4},
          {2, 4, 3}, {2, 3, 4}, {6, 2, 2}, {2, 6, 2}, {2, 2, 6},
          {6, 4}, {4, 6}, {8, 3}, {3, 8}, {12, 2}, {2, 12}, {24}}
```

```
In[71]:= Sort[%] == Sort[OrderedFactorizations[24]]
```

```
Out[71]= True
```

The new approach gives a faster method for generating the factorizations list.

```
In[72]:= Length[OrderedFactorizations2[30^3]] // Timing
```

```
Out[72]= {0.078 Second, 64324}
```

```
In[73]:= Length[OrderedFactorizations[30^3]] // Timing
```

```
Out[73]= {0.172 Second, 64324}
```

Factorizations with Distinct Parts

Again all ordered cases arise as permutations of unordered distinct cases. This also leads to a large speedup in computation time.

```
In[74]:= DistinctOrderedFactorizationsNew[n_] :=  
  Flatten[Permutations /@ DistinctUnorderedFactorizations[n], 1]
```

```
In[75]:= Sort[DistinctOrderedFactorizationsNew[60]] ==  
  Sort[DistinctOrderedFactorizations[60]]
```

```
Out[75]= True
```

□ Faster Count for Ordered Factorizations with Distinct Parts

Let $P_d(k, n)$ denote the number of unordered factorizations into k distinct parts. We observe that $H_d(n) = \sum_k k! P_d(k, n)$. We do not have a formula for $P_d(k, n)$, but we can compute its values by sorting the lists of distinct unordered factorizations according to length. Although we generate (generally short) lists of factorizations as part of the counting process, this turns out to be the fastest method we have found to compute the usually large values of H_d .

First, we sort our lists of factorizations according to length. For example, here $n = 36$.

```
In[76]:= lis =  
  Sort[DistinctUnorderedFactorizations[36], Length[#1] < Length[#2] &]
```

```
Out[76]= {{36}, {18, 2}, {12, 3}, {9, 4}, {6, 3, 2}}
```

Then, we split up the different classes with respect to length, count how many of length k occur, multiply by $k!$, and sum.

```
In[77]:= Split[lis, Length[#1] == Length[#2] &]
```

```
Out[77]= {{{36}}, {{18, 2}, {12, 3}, {9, 4}}, {{6, 3, 2}}}
```

```
In[78]:= a = Length /@ %;  
  Plus @@ (a Range[Length[a]] !)
```

```
Out[79]= 13
```

This agrees with our previous computation. We put this method together as a oneliner.

```
In[80]:=  $\hat{H}_d[n_] := \text{Plus} @@ (\# \text{Range}[\text{Length}[\#]]!) \&[$ 
       $\text{Length} /@ \text{Split}[\text{Sort}[\text{DistinctUnorderedFactorizations}[n],$ 
       $\text{Length}[\#1] < \text{Length}[\#2] \&], \text{Length}[\#1] == \text{Length}[\#2] \&]]$ 
```

```
In[81]:=  $\hat{H}_d[30^3]$  // Timing
```

```
Out[81]:= {0.047 Second, 11655}
```

In general, $\hat{H}_d$ seems to provide a considerable speedup over H_d .

```
In[82]:=  $H_d[30^3]$  // Timing
```

```
Out[82]:= {4.375 Second, 11655}
```

■ Highly Factorable Numbers

Now that we have implemented various methods to count ordered and unordered factorizations, we will put them to use to produce lists of numbers that have a greater number of factorizations than any smaller positive integer.

We say that a natural number n is highly factorable with respect to the function f , if $f(m) < f(n)$ for all m , $1 \leq m < n$. In [8], Canfield, Erdős, and Pomerance computed a list of highly factorable numbers with respect to the function P . Since the functions P and H depend on the prime exponents but not the primes themselves, it is clear that a highly factorable number must be of the form $n = p_1^{r_1} p_2^{r_2} \cdots p_k^{r_k}$ with $r_1 \geq r_2 \geq \cdots r_k \geq 1$ and where p_i denotes the i th prime number. We use `Partitions` to generate a list of acceptable exponents and then define a function `ExponentsToNumber` to produce a natural number n using the exponents from the partition and the corresponding first few primes.

```
In[83]:= ExponentsToNumber[exponentList_] := Times @@
      (Prime[Range[#]] & /@ (Length /@ exponentList)^exponentList)
```

```
In[84]:= Partitions[4]
```

```
Out[84]:= {{4}, {3, 1}, {2, 2}, {2, 1, 1}, {1, 1, 1, 1}}
```

```
In[85]:= ExponentsToNumber[%]
```

```
Out[85]:= {16, 24, 36, 60, 210}
```

To produce all numbers of this form less than a given value *bound*, we must consider all partitions of numbers 1 to `Log[2, bound]`, as the smallest number arising from a partition of n is 2^n .

```
In[86]:= lst[bound_] := Flatten[Table[Partitions[k], {k, Log[2, bound]}], 1];
```

For example, let us find all the highly factorable numbers less than 1000.

```
In[87]:= a = Sort[ExponentsToNumber[lst[1000]]];
```

Now eliminate numbers greater than *bound* in our list, compute the value of the P function for each number in the list, and eliminate the values that are not highly factorable using a replacement rule.

```

In[88]:= b = Select[a, # < 1000 &];
In[89]:= valsP = {#, P1[#]} & /@ b;
In[90]:= LargePlist = valsP //.
          {x___, {y1_, y2_}, {z1_, z2_}, w___} => {x, {y1, y2}, w} /; y2 ≥ z2;

```

Now we use GridBox to display our table of highly factorable P numbers.

```

In[91]:= StyleBox[FormBox[GridBox[Prepend[LargePlist, {"n", "P(n)"}],
          GridFrame → 3, ColumnLines → 1, RowLines → {2, 1}, ColumnSpacings →
          0.5, ColumnAlignments → "."], "TraditionalForm"],
          Background → GrayLevel[0.85], FontSize → 8] // DisplayForm

```

Out[91]//DisplayForm=

n	$P(n)$
2	1
4	2
8	3
12	4
16	5
24	7
36	9
48	12
72	16
96	19
120	21
144	29
192	30
216	31
240	38
288	47
360	52
432	57
480	64
576	77
720	98
960	105

Replacing $P1$ by $H3$ leads to the following list of highly factorable numbers with respect to H .

```

In[92]:= Off[General::spell1]

```

```
In[93]:= LargeHlist = {#, H3[#]} & /@ b //.
  {x___, {y1_, y2_}, {z1_, z2_}, w___} >=> {x, {y1, y2}, w} /; y2 >= z2;
  StyleBox[FormBox[GridBox[Prepend[LargeHlist, {"n", "H(n)"}],
    GridFrame -> 3, ColumnLines -> 1, RowLines -> {2, 1}, ColumnSpacings ->
      0.5, ColumnAlignments -> "."], "TraditionalForm"],
    Background -> GrayLevel[0.85], FontSize -> 8] // DisplayForm

Out[94]//DisplayForm=
```

<i>n</i>	<i>H</i> (<i>n</i>)
2	1
4	2
6	3
8	4
12	8
24	20
36	26
48	48
72	76
96	112
120	132
144	208
192	256
240	368
288	544
360	604
432	768
480	976
576	1376
720	1888
864	2208
960	2496

Erdős, Canfield, and Pomerance were able to compute a table of all highly factorable numbers less than 10^9 with respect to P in their paper [8]. The approach just used gives a much faster method to find the 118 highly factorable numbers less than 10^9 with respect to P , as well as the 124 highly factorable numbers less than 10^9 with respect to H .

□ **Numbers Highly Factorable with Respect to Both P and H**

There appear to be many numbers common to both of the displayed lists. To find these common numbers, join the two lists and extract the first elements (the common values of n). Find the pairs by using `Split`. Then extract the common numbers as the first element of the sublists of length 2.

```
In[95]:= commonPositions = First /@ (Join[LargePlist, LargeHlist] // Sort)

Out[95]:= {2, 2, 4, 4, 6, 8, 8, 12, 12, 16, 24, 24, 36, 36, 48, 48, 72, 72,
  96, 96, 120, 120, 144, 144, 192, 192, 216, 240, 240, 288, 288,
  360, 360, 432, 432, 480, 480, 576, 576, 720, 720, 864, 960, 960}
```

```
In[96]:= First /@ Select[Split[commonPositions, #1 == #2 &], Length[#] == 2 &]
Out[96]:= {2, 4, 8, 12, 24, 36, 48, 72, 96, 120,
           144, 192, 240, 288, 360, 432, 480, 576, 720, 960}
```

So up to 1000, most highly factorable numbers appear in both lists. However, common numbers seem to become less frequent as we increase our bound. For example, we find that there are 55 common highly factorable numbers less than 10^9 , the largest of these being 43545600.

■ Factorizations with Relatively Prime Parts

In this final section we investigate an interesting class of restricted factorizations, namely the class of factorizations in which the factors must be relatively prime to each other. Clearly this is a stronger restriction than requiring distinct factors. The asymptotic growth of such factorizations has been studied by Warlimont [5]. We note that in the special case of *squarefree* integers, all factorizations are necessarily relatively prime and distinct. Thus, for squarefree integers the values of the three functions that count unrestricted or distinct or relatively prime factorizations, all coincide for the ordered and unordered cases respectively.

We will discuss and compare several different approaches to generate the corresponding lists of factorizations.

□ Ordered Factorizations with Relatively Prime Parts

In the ordered case, factorizations of $n = p_1^{r_1} p_2^{r_2} \dots p_k^{r_k}$ into relatively prime parts have a natural correspondence to ordered partitions of a set with the k elements $p_1^{r_1}, p_2^{r_2}, \dots, p_k^{r_k}$. The exponential generating function for the number of ordered set partitions is $\frac{1}{2-E^x}$. (These and further results on set partitions that follow can be found in Wilf's book [9].) From this exponential generating function we can easily compute the first few values.

```
In[97]:= CoefficientList[Series[1/(2 - E^x), {x, 0, 10}], x] Range[0, 10] !
Out[97]:= {1, 1, 3, 13, 75, 541, 4683, 47293, 545835, 7087261, 102247563}
```

Here is another expression for the number of ordered set partitions (sometimes called ordered Bell numbers) due to Carlitz [10].

```
In[98]:= OrderedBell[0] := 1;

In[99]:= OrderedBell[r_] := Sum[Sum[(-1)^(k-j) Binomial[k, j] j^r,
                                   {j, 0, k}], {k, 0, r}]

In[100]:= Table[OrderedBell[r], {r, 0, 10}]
Out[100]:= {1, 1, 3, 13, 75, 541, 4683, 47293, 545835, 7087261, 102247563}
```

In addition there is also this pretty expression as an infinite sum.

$$\text{In}[101]:= \text{OrderedBell12}[n_]:= \frac{1}{2} \sum_{m=0}^{\infty} m^n 2^{-m}$$

Using either of these it is easy to count the number of ordered relatively prime factorizations.

```
In[102]:= H1[n_] := OrderedBell[Length[FactorInteger[n]]]
```

```
In[103]:= H1[30^4]
```

```
Out[103]= 13
```

□ Unordered Factorizations with Relatively Prime Parts

In the unordered case, factorizations of $n = p_1^{r_1} p_2^{r_2} \dots p_k^{r_k}$ into relatively prime parts now have a natural correspondence to unordered partitions of a set with the k elements $p_1^{r_1}, p_2^{r_2}, \dots, p_k^{r_k}$. The exponential generating function for the number of ordered set partitions is E^{E^x-1} . We use this to easily compute the first few values.

```
In[104]:= CoefficientList[Series[E^{E^x-1}, {x, 0, 10}], x] Range[0, 10] !
```

```
Out[104]= {1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975}
```

Again there is a pretty expression for the number of partitions of a set (or Bell numbers) as an infinite sum. Now it is easy to count the number of unordered relatively prime factorizations.

```
In[105]:= Off[General::spell]
```

$$\text{In}[106]:= \text{Bell}[n_]:= \frac{1}{E} \sum_{m=0}^{\infty} m^n / m !$$

```
In[107]:= P1[n_] := Bell[Length[FactorInteger[n]]]
```

```
In[108]:= P1[30^4]
```

```
Out[108]= 5
```

□ Generating Lists of Relatively Prime Factorizations

There are several different approaches that can be used to find the desired lists.

Unordered Relatively Prime Factorizations by Selection

To generate a list of relatively prime factorizations, we need only search among the distinct unordered factorizations and pick out those with relatively prime parts.

Method 1. By Factorization

Our first approach to selecting the relatively prime cases is to factor the numbers in each distinct unordered factorization, and after flattening and sorting, see if this matches the factorization of n .

```

In[109]:= Sort[Flatten[FactorInteger[{15, 4}], 1]] == FactorInteger[60]
Out[109]= True
In[110]:= Sort[Flatten[FactorInteger[{6, 10}], 1]] == FactorInteger[60]
Out[110]= False

```

This shows that {15, 4} is admissible as a relatively prime factorization of 60 but {6, 10} is not. We illustrate the method in the case $n = 60$.

```

In[111]:= DistinctUnorderedFactorizations[60]
Out[111]= {{5, 4, 3}, {6, 5, 2}, {10, 3, 2},
           {10, 6}, {12, 5}, {15, 4}, {20, 3}, {30, 2}, {60}}
In[112]:= a = Flatten[#, 1] & /@ FactorInteger /@ %;

```

We determine the positions of the cases with admissible factorizations. Then we read off these factorizations from the list.

```

In[113]:= Position[Sort /@ a, FactorInteger[60]]
Out[113]= {{1}, {5}, {6}, {7}, {9}}
In[114]:= DistinctUnorderedFactorizations[60][[Flatten[%]]]
Out[114]= {{5, 4, 3}, {12, 5}, {15, 4}, {20, 3}, {60}}

```

We put this together into one function.

```

In[115]:= UnorderedRelativelyPrime1[n_] := DistinctUnorderedFactorizations[
           n][[Flatten[Position[Sort /@ (Flatten[#, 1] & /@ (FactorInteger /@
           DistinctUnorderedFactorizations[n])), FactorInteger[n]]]]]
In[116]:= UnorderedRelativelyPrime1[240]
Out[116]= {{16, 5, 3}, {16, 15}, {48, 5}, {80, 3}, {240}}

```

Method 2. By Greatest Common Divisors

A second approach is to check that the factors are relatively prime directly, by finding the greatest common divisors of every pair of factors. However, Bressoud and Wagon [11] give a much more efficient way to test a long list for relative primality of all pairs.

```

In[117]:= RelativelyPrimeList[lst_] := LCM @@ lst == Times @@ lst

```

We rewrite our function accordingly.

```

In[118]:= UnorderedRelativelyPrime2[n_] := DistinctUnorderedFactorizations[
           n][[Flatten[Position[RelativelyPrimeList /@
           DistinctUnorderedFactorizations[n], True]]]]
In[119]:= Length[UnorderedRelativelyPrime2[240 49 11 13]] // Timing
Out[119]= {0.438 Second, 203}
In[120]:= Length[UnorderedRelativelyPrime1[240 49 11 13]] // Timing
Out[120]= {0.219 Second, 203}

```

Method 3: By Generating Set Partitions

The idea here is to generate the partitions of a set with k elements and then replace these elements with the appropriate prime powers to obtain a list of relatively prime factorizations. We can recursively compute the lists of set partitions by noting that the partitions of a set with n elements can be created by appending the *singleton* $\{n\}$ to each list of partitions of $n - 1$ elements, or by appending the *element* n to each partition of $n - 1$ elements. An implementation of this recursion using `ReplaceList` is given by Dickau [12] as follows.

```
In[121]:= BellRule1[n_] := {S__} -> {S, {n}};
          BellRule2[n_] := {b___, {S__}, a___} -> {b, {S, n}, a};

In[123]:= BellLists[1] = {{1}};

In[124]:= BellLists[n_Integer?Positive] := BellLists[n] =
          Flatten[
            Map[ReplaceList[#, {BellRule1[n], BellRule2[n]}] &,
              BellLists[n - 1]], 1]
```

Here is an example using Dickau's function.

```
In[125]:= BellLists[3]

Out[125]= {{1}, {2}, {3}}, {{1, 3}, {2}},
          {{1}, {2, 3}}, {{1, 2}, {3}}, {{1, 2, 3}}
```

Now if, for example, $n = 180 = 4 * 9 * 5$, we must substitute $1 \rightarrow 4$, $2 \rightarrow 9$, $3 \rightarrow 5$ to obtain the factors in the relatively prime factorizations. Finally, we multiply the factors together to produce the desired result.

```
In[126]:= BellLists[3] /. MapThread[Rule, {Range[3], {4, 9, 5}}]

Out[126]= {{4}, {9}, {5}}, {{4, 5}, {9}},
          {{4}, {9, 5}}, {{4, 9}, {5}}, {{4, 9, 5}}

In[127]:= Apply[Times, #, 1] & /@ %

Out[127]= {{4, 9, 5}, {20, 9}, {4, 45}, {36, 5}, {180}}
```

Putting this together we have a nice oneliner. This approach is unsurprisingly much faster than the previous ones!

```
In[128]:= UnorderedRelativelyPrime3[n_] := With[{f = FactorInteger[n]},
          Apply[Times, #, 1] & /@ (BellLists[Length[f]] /.
            MapThread[Rule, {Range[Length[f]], Power @@ f}])]

In[129]:= Length[UnorderedRelativelyPrime3[240 49 11 13]] // Timing

Out[129]= {0. Second, 203}
```

Ordered Relatively Prime Factorizations

Whichever of the three approaches is used to generate the unordered cases, we need only permute the elements of each such unordered factorization to produce the lists of the ordered ones.

```
In[130]:= OrderedRelativelyPrime[n_] :=
          Flatten[Permutations /@ UnorderedRelativelyPrime1[n], 1]
```

■ Conclusion

We have studied methods to enumerate and to construct ordered and unordered factorizations of integers, subject to various constraints on the parts. Recursive descriptions of these objects, together with *Mathematica*'s functional programming techniques, were used to implement these algorithms efficiently.

■ Acknowledgments

The authors thank the referee for his careful reading of the manuscript and for enhancements to both the efficiency and elegance of the coding. In addition, the authors thank Glenn Scholebo, the compositor of *The Mathematica Journal*, for improving the typesetting of the paper.

■ References

- [1] B. Richmond and A. Knopfmacher, "Compositions with Distinct Parts," *Aequationes Mathematicae*, **49**, 1995 pp. 86–97.
- [2] G. E. Andrews, *The Theory of Partitions*, Encyclopedia of Mathematics and Its Applications 2, New York: Addison-Wesley, 1976.
- [3] E. Hille, "A Problem in 'Factorisatio Numerorum,'" *Acta Arithmetica*, **2**, 1936 pp. 134–144.
- [4] P. A. MacMahon, "Memoir on the Theory of the Compositions of Numbers," *Philosophical Transactions of the Royal Society of London (A)*, **184**, 1893 pp. 835–901.
- [5] R. Warlimont, "Factorisatio Numerorum with Constraints," *Journal of Number Theory*, **45**, 1993 pp. 186–199.
- [6] V. C. Harris and M. V. Subbarao, "On Product Partitions of Integers," *Canadian Mathematical Bulletin*, **34**(4), 1991 pp. 474–479.
- [7] J. F. Hughes and J. O. Shallit, "On the Number of Multiplicative Partitions," *American Mathematical Monthly*, **90**(7), 1983 pp. 468–471.
- [8] E. R. Canfield, P. Erdős, and C. Pomerance, "On a Problem of Oppenheim Concerning 'Factorisatio Numerorum,'" *Journal of Number Theory*, **17**, 1983 pp. 1–28.
- [9] H. S. Wilf, *generatingfunctionology*, 2nd ed., New York: Academic Press, 1994.
- [10] I. Carlitz, "Extended Bernoulli and Eulerian Numbers," *Duke Mathematical Journal*, **31**, 1964 pp. 667–689.
- [11] D. M. Bressoud and S. Wagon, *A Course in Computational Number Theory*, New York: Springer-Verlag, 2000.
- [12] R. Dickau, "Visualizing Combinatorial Enumeration," *Mathematica in Education and Research*, **8**, 1999 pp. 11–18.

About the Authors

Arnold Knopfmacher is a Professor of Mathematics at the University of the Witwatersrand, Johannesburg, where he also obtained his Ph.D. degree. His main research interests are in enumerative combinatorics and elementary number theory. He is Director of The John Knopfmacher Centre for Applicable Analysis and Number Theory, which was established in 1992 by his late father, a distinguished number theorist. Prior to John's death in 1999, Arnold and John collaborated extensively producing over thirty joint papers together.

Michael Mays is a Professor of Mathematics at West Virginia University, where he does research in combinatorics and number theory and develops software and course materials for the Institute for Math Learning. He has maintained his collaboration with Arnold Knopfmacher with five visits over the last decade to The John Knopfmacher Centre for Applicable Analysis and Number Theory at the University of the Witwatersrand.

Arnold Knopfmacher

*The John Knopfmacher Centre for Applicable Analysis and Number Theory
University of the Witwatersrand
Johannesburg 2050, South Africa
arnoldk@cam.wits.ac.za
www.wits.ac.za/science/number_theory/arnold.htm*

Michael Mays

*Department of Mathematics
West Virginia University
Morgantown, WV 26506-6310
mays@math.wvu.edu
www.math.wvu.edu/~mays*

Approximating Solutions of Linear Ordinary Differential Equations with Periodic Coefficients by Exact Picard Iterates

Armando G. M. Neves

Linear ordinary differential equations (ODEs) with periodic coefficients appear in various interesting applications, such as determining the linear stability regions of systems of vertically driven multiple pendula. Sinha and Butcher [1, 2] have obtained very good approximations to the solutions of such equations by calculating approximate Picard iterates symbolically in the parameters on which the system depends. In this article we show an improvement to the method of Sinha and Butcher. We are able to calculate exact, rather than approximate, Picard iterates of high order. The key point in the programming is the necessity of introducing a user-defined function to carry out the integrations that appear in the definition of the Picard iterates. After introducing the concept of Picard iteration and explaining its fast implementation, we apply the method to determine the stability regions for linearized systems of vertically driven multiple pendula.

■ 1. Introduction

Systems of differential equations with periodic coefficients arise in several different applications in physics and engineering. Because of their applications, such systems also deserve attention from pure scientists. One important method for understanding them is linearization around an equilibrium solution. This justifies the attention we devote to systems of linear differential equations with periodic coefficients.

Despite being simpler than nonlinear systems, linearized systems of differential equations in general are not exactly solvable if their coefficients are not constant. It is thus natural to consider approximate solutions for these systems. Besides traditional numerical methods, other kinds of methods, such as perturbation theory [3] and averaging [4], are used. Whereas traditional numerical methods suffer from the problem of being difficult to apply in cases where the equations depend on parameters, perturbation and averaging usually perform well only in some limited regions in the space of parameters.

Picard iteration, described in Section 2, is a well-known method because of its theoretical importance in proving the existence-uniqueness theorem for differential equations [5]. We were delighted and surprised to see it appear in [1] as a practical tool for approximating solutions of differential equations. Furthermore, it can also be used if the equations depend on parameters.

The authors of [1] expand the coefficients matrix of a linear system with periodic coefficients in a series of shifted Chebyshev polynomials. By cleverly using properties of the Chebyshev polynomials, they are able to trade the integrations occurring in Picard iteration by matrix multiplications. As a result, excellent *approximations* for the coefficients of the Chebyshev polynomial series of high-order Picard iterates can be obtained. Their method is thus approximate in a double sense.

First, the infinite sequence of Picard iterates converges to the exact solution, but the Picard iterate at any finite order is only an approximate solution for the differential equations. Second, they do not obtain exact Picard iterates but a truncation of a series converging to them.

While reading [1] we asked ourselves what the difficulty would be if, instead of approximate Picard iterates, we tried to calculate exact ones. Picard iterates are seldom used as an approximate solution method to general differential equations because functions that are impossible to integrate are likely to appear during the iteration. We will see that this is indeed what happens in a simple case, the nonlinear pendulum equation. However, this will not be the case for systems of linear ODEs with periodic coefficients. This assertion will be proved in Section 3. In order to approximately solve these equations, all we then need is a sufficiently fast integration function.

In [6], we showed that it is possible to obtain practical results by calculating *exact* Picard iterates for linear differential equations with periodic coefficients. The number of iterations performed were equal to the ones computed by Sinha and Butcher in [1] and [2]. Although the computers and *Mathematica* versions used are different, the times needed are much shorter with our method. Moreover, exactness in our results leads to important differences in accuracy in the parameter regions considered. In this article we will concentrate on explaining the implementation in detail instead of comparing results.

Our method directly uses definition (2) of the Picard iterate and the specialized integration function we created, which calculates all necessary integrals much faster than by using the built-in function `Integrate`. We provide the package

(described in Section 4) in Additional Material that carefully implements all these ideas.

Section 5 is devoted to the application of the package to the question of *linear* stability of a system of upside-down pendula driven by vertical periodic motion of its suspension point. This kind of system has been considered recently both by pure [7, 8, 9] and applied [1, 2] scientists. Besides being an application of the method devised, it is also a good illustration of the power of *Mathematica*. Not only is it used to approximately solve the differential equations, but also for deducing the equations of motion of the pendula, linearizing them, diagonalizing matrices, and graphing the results.

■ 2. Picard Iterates

Let $t \in \mathbb{R}$ and $x \equiv (x_1, \dots, x_n) \in \mathbb{R}^n$. If $f(t, x)$ is a function with values in $\mathbb{R}^n$, then consider the following general initial value problem for a system of n ordinary differential equations:

$$\begin{cases} \frac{dx}{dt} = f(t, x) \\ x(t_0) = x_0 \end{cases} \quad (1)$$

The Picard iterate of a function $y(t)$ with values in $\mathbb{R}^n$ related to the initial condition $x(t_0) = x_0$ is a new function defined as

$$(T_{x_0} y)(t) = x_0 + \int_{t_0}^t f(s, y(s)) ds. \quad (2)$$

Taking any initial function $y_0(t)$, let $y_1(t) = (T_{x_0} y_0)(t)$, $y_2(t) = (T_{x_0} y_1)(t)$, ... It can be shown that if f and $\frac{\partial f}{\partial x}$ in equation (1) are both continuous functions in a neighborhood of the initial point (t_0, x_0) , then the sequence of Picard iterates $y_1(t)$, $y_2(t)$, ... converges in a neighborhood of t_0 to the solution of the initial value problem (1). This is in fact the usual proof of the existence-uniqueness theorem for differential equations [5]. In general we choose as the initial function $y_0(t)$ the constant function equal to the initial condition x_0 . This ensures that the very first iterates are good approximations to the solution for t close enough to t_0 .

Although convergence to the solution of the problem is guaranteed (and exponentially fast in the number of iterates), Picard iteration is seldom used as a method for approximating solutions to differential equations. The problem with it lies in the integration contained in the very definition (2) of the Picard iterate, which may be laborious or even impossible to perform. Take as an example the *pendulum equation*

$$\theta'' + \frac{g}{l} \sin \theta = 0, \quad (3)$$

whose solution cannot be written in terms of elementary functions. We rewrite it as a first-order system of the form (1) by taking $x(t) = (x_1(t), x_2(t)) = (\theta(t), \theta'(t))$,

$f(t, x(t)) = (x_2(t), \frac{-g}{l} \sin x_1(t))$. A function `onepicard` that performs the operation $T_{x_0} y$ in equation (2) is defined as follows.

```
In[1]:= onepicard[f_, tzero_, xzero_, y_] :=
      Function[t, xzero + ∫tzerot f[s, y[s]] ds]
```

Both $f = f(t, x)$, the function on the right-hand side of the system of differential equations, and $y = y(t)$, the function that we want to iterate, should be supplied as pure functions. As we will also be interested in iterating the operator T_{x_0} over an initial function that is a constant function equal to the initial condition x_0 , we define the function `picard`.

```
In[2]:= picard[f_, tzero_, xzero_, n_Integer] :=
      Nest[onepicard[f, tzero, xzero, #1] &, xzero &, n]
```

Using $g = l = 1$ to simplify matters, here is the fifth Picard iterate for the pendulum equation with initial condition $x(0) = (1, 0)$.

```
In[3]:= picard[{#2[2], -Sin[#2[1]]} &, 0, {1, 0}, 5][t][2]
```

$$\text{Out[3]} = \int_0^t -\text{Sin}\left[1 - \text{Cot}[1] + \text{Cos}\left[1 - \frac{1}{2}s^2 \text{Sin}[1]\right] \text{Csc}[1] - s \text{FresnelC}\left[\frac{s}{\sqrt{\pi \text{Csc}[1]}}\right] \sqrt{\pi \text{Sin}[1]} + s \text{Cot}[1] \text{FresnelS}\left[\frac{s}{\sqrt{\pi \text{Csc}[1]}}\right] \sqrt{\pi \text{Sin}[1]}\right] ds$$

Notice that this result is an unevaluated integral! The reader may rework the last input to find that all integrals can be calculated up to the fourth Picard iterate. This shows that high-order Picard iterates may be difficult or impossible to calculate. By comparing the graphs of the fourth Picard iterate and the solution of the same problem by a traditional numerical method, such as the one implemented in `NDSolve`, the reader will also notice that the approximation is not accurate enough in a time interval as small as the period of the pendulum.

■ 3. Picard Iteration for Systems of Linear ODEs with Periodic Coefficients

In this section we show that for systems of linear differential equations with periodic coefficients we never encounter integrals that are impossible to calculate. But we will also see that evaluating Picard iterates, even if technically possible, may take too long if we use the built-in `Integrate` command.

More exactly, we will be concerned with systems of the form

$$\frac{dx}{dt} = P(t)x(t), \quad (4)$$

where $P(t)$ is a matrix satisfying

$$P(t+T) = P(t) \quad (5)$$

for all $t \in \mathbb{R}$. As a simplification, we restrict ourselves a bit more by considering the case where $P(t)$ is a finite sum of terms proportional to sines and cosines of period T , that is,

$$P(t) = P_0 + \sum_{k=1}^N \left[P_k^c \cos\left(\frac{2\pi k}{T} t\right) + P_k^s \sin\left(\frac{2\pi k}{T} t\right) \right]. \quad (6)$$

In Section 5 of this article we will see some very interesting examples belonging to this class, but any reasonable periodic matrix with period T can be cast approximately in the previous form by retaining a finite number of terms in its Fourier series.

An important example is the *Mathieu equation*

$$y'' + (a + b \cos t) y = 0, \quad (7)$$

which can be rewritten in the form (4) by making $x_1 = y$ and $x_2 = y'$ and taking

$$P(t) = \begin{pmatrix} 0 & 1 \\ -(a + b \cos t) & 0 \end{pmatrix}.$$

Although the Mathieu equation is exactly solvable with *Mathematica*, we will use it as our prototype example for its simplicity and importance.

`In[4]:= DSolve[y''[t] + (a + b Cos[t]) y[t] == 0, y[t], t]`

`Out[4]= {{y[t] -> C[1] MathieuC[4 a, -2 b, t/2] + C[2] MathieuS[4 a, -2 b, t/2]}}`

We may also compare our approximate solutions with the exact ones. It should be noted that Mathieu functions, that is, the solutions of the Mathieu equation, are notably difficult to implement with a computer. For this reason they are the subject of recent research [10, 11, 12].

Let us evaluate some Picard iterates approximating the solution of the Mathieu equation, so that we can learn something from them. Taking an initial condition such as $x(0) = (1, -2)$, here are the first, second, and third iterates.

`In[5]:= Table[Expand[`

`picard[{{0, 1}, {-(a + b Cos[#1]), 0}}.#2 &, 0, {1, -2}, i][t]], {i, 3}]`

`Out[5]= {{1 - 2 t, -2 - a t - b Sin[t]}, {1 - b - 2 t - \frac{a t^2}{2} + b Cos[t],`
`-2 - 2 b - a t + a t^2 + 2 b Cos[t] - b Sin[t] + 2 b t Sin[t]},`
`{1 - b - 2 t - 2 b t - \frac{a t^2}{2} + \frac{a t^3}{3} + b Cos[t] - 2 b t Cos[t] + 4 b Sin[t],`
`-2 - 2 b - a t + a b t - \frac{b^2 t}{2} + a t^2 + \frac{a^2 t^3}{6} + 2 b Cos[t] +`
`a b t Cos[t] - b Sin[t] - 2 a b Sin[t] + b^2 Sin[t] +`
`2 b t Sin[t] + \frac{1}{2} a b t^2 Sin[t] - \frac{1}{2} b^2 Cos[t] Sin[t]}}`

The reader can see that the expanded form of all these iterates is a sum of terms that either are constants, positive integer powers of t , sines or cosines, or assume

one of the forms $t^r \cos(qt)$ or $t^r \sin(qt)$, where r is a positive integer. Let us define $\mathbb{F}$ as the family of the finite linear combinations of all such functions and see whether the next iterate still belongs to the family $\mathbb{F}$.

If we calculate the next iterate by hand, we first have to multiply matrix P by the last element in the previous list and then integrate.

```
In[6]:= {{0, 1}, {-(a + b Cos[t]), 0}}.%[[3]]
```

```
Out[6]= {-2 - 2 b - a t + a b t -  $\frac{b^2 t}{2}$  + a t2 +  $\frac{a^2 t^3}{6}$  + 2 b Cos[t] +  
a b t Cos[t] - b Sin[t] - 2 a b Sin[t] + b2 Sin[t] + 2 b t Sin[t] +  
 $\frac{1}{2}$  a b t2 Sin[t] -  $\frac{1}{2}$  b2 Cos[t] Sin[t], (-a - b Cos[t])  
(1 - b - 2 t - 2 b t -  $\frac{a t^2}{2}$  +  $\frac{a t^3}{3}$  + b Cos[t] - 2 b t Cos[t] + 4 b Sin[t])}
```

Keeping in mind the possible appearance of difficult integrals, the main difficulty at this point is the appearance of products involving sines and cosines when we expand the last expression. But we may transform such products into sums of trigonometric functions by using `TrigReduce`, which again brings our expression back to the explicit form of a function in $\mathbb{F}$. We still need to integrate this expression. That may be done by using the recursive formulas

$$\int t^r \cos q t dt = \frac{t^r}{q} \sin q t - \frac{r}{q} \int t^{r-1} \sin q t dt \quad (8)$$

$$\int t^r \sin q t dt = -\frac{t^r}{q} \cos q t + \frac{r}{q} \int t^{r-1} \cos q t dt, \quad (9)$$

obtained by an easy integration by parts. We thus see that all integrals of functions in $\mathbb{F}$ always belong to $\mathbb{F}$.

Although we are not interested in giving a formal proof here, all the reasoning with the third iterate can be turned into a proof by induction of the following result.

Theorem: Any Picard iterate of linear systems in the form of equation (4) with P in the form of equation (6) is a function in the family $\mathbb{F}$.

Although Picard iteration is thus technically possible at any finite order, as the order grows the resulting expressions become increasingly complicated. As a consequence, the time needed to compute them increases quickly.

```
In[7]:= Table[
```

```
Timing[picard[{{0, 1}, {-(a + b Cos[#1]), 0}}.#2 &, 0, {1, -2}, n][t];][[
```

```
1], {n, 8}]
```

```
Out[7]= {0. Second, 0.05 Second, 0.11 Second, 0.39 Second,  
0.99 Second, 2.41 Second, 8.02 Second, 17.91 Second}
```

Although one might assume that eight Picard iterates are enough, we caution the reader that for the applications we have in mind it will be necessary to evaluate the solution to the Mathieu equation at time $t = 2\pi$, that is, at the end of the period of the matrix of the system. To see that eight iterations are too few, we

recommend comparing the graphs of the approximate solution obtained by Picard iteration (substituting a and b by typical values 0.9 and 0.6, respectively) and the exact solution given by `DSolve`.

■ 4. Fast Implementation of Picard Iteration for Systems of Linear ODEs with Periodic Coefficients and the *FastPicard* Package

By constructing an alternative function, we will see that the main reason for the slowness of `picard` is that it uses the built-in function `Integrate` for integration. `Integrate` is of course very good for general purposes, but it is too slow in cases such as ours, in which we have to calculate a very large number of integrals of functions in a limited class. As shown in the previous section, we do not need all the generality of `Integrate` because all integrands belong to the family $\mathbb{F}$.

We created a new function, specialized for definite and indefinite integration of functions in $\mathbb{F}$, to be used instead of `Integrate` in the Picard iteration scheme. We called it `NewIntegrate` and gave it the same syntax as `Integrate`. It is implemented in our *FastPicard* package (see Additional Material) and exported by it.

First, `NewIntegrate` recognizes a linear combination of terms and uses the fact that the integral is a linear operator. This is a standard trick used, for example, in the implementation of Laplace transforms [9, 13]. It then readily performs indefinite integrals of constants, powers, sines, and cosines. The more difficult functions in $\mathbb{F}$ are dealt with by using the recursive formulas of equations (8) and (9). As usual in these cases, we track all results, so that the same integral need not be calculated more than once. This is accomplished by the old trick of using a delayed and an immediate assignment, as explained in Section 2.4.9 of [14].

In calculating definite integrals, a trick for further speed is to use $\frac{1}{m} - \frac{1}{m} \cos mx$ instead of $-\frac{1}{m} \cos mx$ as the expression for the indefinite integral of $\sin mx$. In most cases, we will be interested in using $t_0 = 0$ in equation (2). This trick ensures that all definite integrals of functions in $\mathbb{F}$ with a lower limit equal to zero will coincide with the corresponding indefinite integrals.

The reader is invited to test the accuracy and performance of `NewIntegrate` by producing a large random function in $\mathbb{F}$ and integrating it with both functions. In our tests, we found that `NewIntegrate` is tens of times faster than the built-in function.

The package also exports the function `FastPicard[matrix, tzero, xzero, n]` that is similar to `picard` but uses `NewIntegrate`. For functions in $\mathbb{F}$ to be recognized by `NewIntegrate`, it is necessary to transform products of sines and cosines into sums before integration. As we are always dealing with linear systems, the syntax is a bit simplified—instead of providing the function $f(t, x)$, the reader should provide matrix $P(t)$.

Of course, times are much shorter than the ones obtained by `picard`. This is shown in the following example, which also illustrates the syntax.

```
In[8]:= << FastPicard.m

In[9]:= Table[
  Timing[FastPicard[{{0, 1}, {-a - b Cos[#1], 0}} &, 0, {1, -2}, n][t];],
  {n, 10}]

Out[9]= {{0. Second, Null}, {0. Second, Null},
  {0. Second, Null}, {0. Second, Null}, {0.05 Second, Null},
  {0.06 Second, Null}, {0.11 Second, Null},
  {0.16 Second, Null}, {0.22 Second, Null}, {0.44 Second, Null}}
```

Besides using `NewIntegrate`, *FastPicard* incorporates other improvements we discovered after some experimenting. Here is a list of the differences.

- In order to avoid calculating the same integral more than once, similar terms are gathered before they are acted on by `NewIntegrate`. The gathering is performed by an auxiliary function named `gather`, not exported by the package, and is responsible for part of the speed increase when the number of iterates is large enough to compensate for the extra work of gathering.
- To our surprise, we found that in Version 4.1 the built-in function `TrigReduce` is also slower than a set of transformation rules designed to transform products of sines and cosines into sums. We implemented such rules in *FastPicard*.
- As we are mostly interested in Picard iterates with $t_0 = 0$, we made the second argument of `FastPicard` optional with a default value of 0. Instead of calculating definite integrals, in this case we may replace them by indefinite ones that evaluate much faster.

The package also exports the functions `FundamentalMatrix` and `FloquetMatrix`.

A *fundamental matrix solution* to a linear system such as equation (4) is a matrix whose columns are solutions to the system with linear independent initial conditions. The function `FundamentalMatrix[matrix, tzero, n]` calculates n Picard iterates approximating the particular fundamental matrix solution that takes the columns of the identity matrix as initial conditions at time `tzero`. The second argument `tzero` is optional with a default value of 0. For example, with 20 iterates and initial time 0, an approximate fundamental matrix solution to the system with

$$P(t) = \begin{pmatrix} -1 + a \cos^2 t & 1 - a \sin t \cos t \\ -1 - a \sin t \cos t & -1 + a \sin^2 t \end{pmatrix}$$

is obtained by

```
In[10]:= p[t_] = TrigReduce[{{-1 + a Cos[t]^2, 1 - a Sin[t] Cos[t]},
  {-1 - a Sin[t] Cos[t], -1 + a Sin[t]^2}}];
```

```
In[11]:= fms = FundamentalMatrix[p, 20][t];
```

One advantage of having a fundamental matrix solution calculated at initial time t_0 is that with it we may recover solutions to the system with any desired initial conditions at time t_0 . This is even simpler when we use the particular fundamental matrix solution calculated by `FundamentalMatrix`. For example, this gives the approximate numerical value for the solution of this system with $a = 2$ at time $t = 2\pi$ and initial condition $y(0) = (1, -2)$.

```
In[12]:= ((fms /. {a -> 2., t -> 2 Pi}).{1, -2})
```

```
Out[12]= {535.492, -0.00373489}
```

These values can be compared with the corresponding exact ones. In fact, the exact fundamental matrix solution for the system is given in [1]. Using it, we obtain

```
In[13]:= N[{{e^(a-1) t Cos[t], e^-t Sin[t]}, {-e^(a-1) t Sin[t], e^-t Cos[t]}}.{1, -2} /.  
          {a -> 2, t -> 2 Pi}]
```

```
Out[13]= {535.492, -0.00373489}
```

The *Floquet transition matrix* for a linear system of differential equations with periodic coefficients of period T is just the special fundamental matrix solution calculated by `FundamentalMatrix` with initial time 0 evaluated at time T . In the next section we will see an interesting application for this matrix. An approximation to it using n Picard iterates is calculated by `FloquetMatrix[matrix, per, n]`, where the second argument, the period of the matrix, is optional with a default value of 2π .

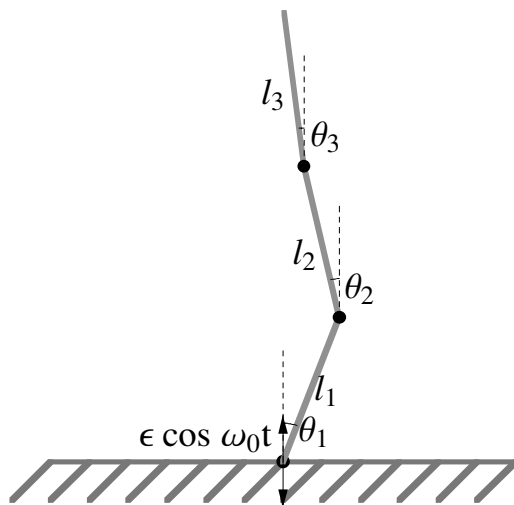
For example, here is an approximation to the Floquet transition matrix for the Mathieu equation with $a = -0.1$ and $b = 0.55$.

```
In[14]:= ftmmathieu = FloquetMatrix[{{0, 1}, {-a - b Cos[#1], 0}} &, 20];  
          ftmmathieu /. {a -> -0.1, b -> 0.55}
```

```
Out[15]= {{-0.0397796, 18.6577}, {-0.0535123, -0.0397796}}
```

■ 5. Application to Linearized Systems of Driven Pendula

Consider a system of N upside-down physical pendula (homogeneous rods, in fact) such as in the following graphic, where we depict the case $N = 3$. It is assumed that each rod may rotate without friction around its bearing and that gravity points down.



It is a fact of everyday experience that the upside-down equilibrium position in which all rods are fixed at the vertical position $\theta_i = 0$, $i = 1, 2, \dots, N$ is highly unstable.

Curiously, however, if the support point of the first rod is driven by an external force such that it executes a vertical sinusoidal motion of amplitude ϵ and angular frequency ω_0 , and if ϵ and ω_0 are chosen from a suitable range, then the upside-down equilibrium becomes stable. This fact seems to have been first predicted in 1908 by Stephenson [15] for a single rod and subsequently extended by him to the case of two and three rods in 1909 [16]. Interest in such systems was aroused more recently by Acheson, who proved in [7] that if the nonlinear equations of motion for the rods are linearized in the neighborhood of the equilibrium solution, then, for any value of N , there exists a stability region for this system of *linearized* upside-down rods. In other words, there exists a range in ϵ and ω_0 where the upside-down equilibrium for the linearized equations is stable. If the values of the angles θ_i and their derivatives are small, one may hope that the solution of the linearized equations is close to the solution of the nonlinear equations, but a mathematical proof of stability for nonlinear rods is much more difficult [9]. In fact, even an experimental proof of stability for systems of one, two, and three rods has merited a good deal of attention recently [8]. Since then, the number of papers in this field has been very large and even outside of the physics and mathematics communities. Stability of upside-down rods (or pendula) have attracted interest in areas ranging from control engineering to biomechanics. A good reference from the mathematical physics point of view with many further references is [9].

□ The Lagrangian

In this subsection we use the Lagrangian formalism of classical mechanics [17] to deduce the equations of motion for *undamped* systems of periodically driven

physical pendula and then linearize them around the upside-down equilibrium position. The Euler–Lagrange equations of motion are

$$\frac{d}{dt} \frac{\partial L}{\partial \dot{q}_i} - \frac{\partial L}{\partial q_i} = 0, \quad i = 1, 2, \dots, N, \quad (10)$$

where the Lagrangian function L is the difference between kinetic and potential energies.

Consider a system such as the one in the previous graphic, where the i th pendulum has length l_i and mass m_i . Suppose that the mass of the i th pendulum is distributed such that its center of mass is located at a distance H_i from its bottom extremity and its inertia moment through an axis passing through the bottom extremity and orthogonal to the plane of motion is $\frac{1}{2} m_i R_i^2$, that is, the gyration radius is R_i . Suppose also that the support point of the lower pendulum is externally driven so that its position vector is $\vec{r}_1(t) = (\xi(t), \eta(t))$. We will denote with $\vec{v}_i(t)$ the velocity vector of the bottom extremity of the i th pendulum with respect to the inertial lab frame. The velocity vector of the top of pendulum i with respect to the noninertial frame of its bottom will be denoted with $\vec{w}_i(t)$. Using well-known expressions for K_i and V_i , the kinetic and gravitational potential energies of a rigid body, and g for the acceleration of gravity, we obtain as follows the Lagrangian L for the system.

$$\begin{aligned} \text{In}[16] := & \quad \mathbf{v}[1] := \partial_t \{ \xi[t], \eta[t] \}; \\ & \quad \mathbf{w}[i_] := \partial_t \{ l[i] \sin[\theta[i][t]], l[i] \cos[\theta[i][t]] \}; \\ & \quad \mathbf{v}[i_] := \mathbf{v}[i-1] + \mathbf{w}[i-1]; \\ & \quad \mathbf{K}[i_] := \\ & \quad \quad \frac{1}{2} m[i] \mathbf{v}[i] \cdot \mathbf{v}[i] + \frac{1}{2} m[i] R[i]^2 (\partial_t \theta[i][t])^2 + \frac{m[i] H[i] \mathbf{v}[i] \cdot \mathbf{w}[i]}{l[i]}; \\ & \quad \mathbf{V}[i_] := m[i] g \left(\sum_{k=1}^{i-1} l[k] \cos[\theta[k][t]] + H[i] \cos[\theta[i][t]] \right); \\ & \quad \mathbf{L}[i_] := \sum_{j=1}^i (\mathbf{K}[j] - \mathbf{V}[j]) \end{aligned}$$

$\mathbf{L}[n]$ will produce the Lagrangian for a system of n pendula.

□ A Single Rod

Here is the Lagrangian for a single upside-down driven rod, for example.

$$\begin{aligned} \text{In}[22] := & \quad \mathbf{L}[1] \\ \text{Out}[22] := & \quad -g \cos[\theta[1][t]] H[1] m[1] + \\ & \quad \frac{1}{2} m[1] (\eta'[t]^2 + \xi'[t]^2) + \frac{1}{2} m[1] R[1]^2 \theta[1]'[t]^2 + \\ & \quad \frac{1}{l[1]} (H[1] m[1] (-l[1] \sin[\theta[1][t]] \eta'[t] \theta[1]'[t] + \\ & \quad \cos[\theta[1][t]] l[1] \xi'[t] \theta[1]'[t])) \end{aligned}$$

From it we can evaluate the left-hand side of equation (10).

In[23]:= **lhs[1] = ∂_t ($\partial_{\theta[1]'$ [t] L[1]) - $\partial_{\theta[1]}$ [t] L[1];**

For a rod with uniform mass distribution ($H_1 = \frac{l}{2}$, $R_1 = \frac{l}{\sqrt{3}}$) and sinusoidal (in fact, cosinusoidal) vertical driving, the equation of motion is

In[24]:= **Simplify**[**lhs[1]** /. {**m[1]** → **m**, **l[1]** → **l**, **H[1]** → $\frac{l}{2}$,
 $R[1] \rightarrow \frac{1}{\sqrt{3}}$, $\xi[t] \rightarrow 0$, $\xi'[t] \rightarrow 0$, $\xi''[t] \rightarrow 0$, $\eta[t] \rightarrow \epsilon \cos[\omega_0 t]$,
 $\eta'[t] \rightarrow \partial_t (\epsilon \cos[\omega_0 t])$, $\eta''[t] \rightarrow \partial_{\{t, 2\}} (\epsilon \cos[\omega_0 t])$ }]

Out[24]= $\frac{1}{6} l m (-3 g \sin[\theta[1][t]] + 3 \epsilon \cos[t \omega_0] \sin[\theta[1][t]] \omega_0^2 + 2 l \theta[1]''[t])$

Here linearization is substituting $\sin \theta$ by θ .

In[25]:= **Collect**[**Expand**[$\frac{3}{m l^2} \% /. \sin[\theta[1][t]] \rightarrow \theta[1][t]$], $\theta[1][t]$]

Out[25]= $\left(-\frac{3 g}{2 l} + \frac{3 \epsilon \cos[t \omega_0] \omega_0^2}{2 l}\right) \theta[1][t] + \theta[1]''[t]$

It is possible to put the equation of motion in a simpler form, depending on a smaller number of parameters, if we rescale the time variable defining

$$\tau = \omega_0 t, \quad (11)$$

the new form assumed by the equation is then

$$\theta''(\tau) + \left(-\frac{3 g}{2 \omega_0^2 l} + \frac{3 \epsilon}{2 l} \cos \tau\right) \theta(\tau) = 0, \quad (12)$$

which is the Mathieu equation (7) with

$$a = -\frac{3 g}{2 \omega_0^2 l} \quad (13)$$

and

$$b = \frac{3 \epsilon}{2 l}. \quad (14)$$

So, the question of stability of an upside-down vertically driven single rod amounts to stability of the solutions of the Mathieu equation. This has been studied for a long time [18].

The Floquet theory [18] for linear differential equations with periodic coefficients examines, among other things, the stability of equilibrium solutions. It is proved that the equilibrium solution $x(t) = 0$ for a linear system with periodic coefficients is stable if and only if all eigenvalues λ_i of the Floquet transition matrix (FTM) are such that $|\lambda_i| \leq 1$.

In the particular case of Mathieu equations, one proves that the transition from stability to instability occurs when both eigenvalues are equal to 1 or both equal to -1 . In other words, the curves in the $a b$ plane where solutions to the Mathieu

equation change from stable to unstable or vice versa are given by the condition that

$$\text{Tr FTM} = \pm 2. \quad (15)$$

Before we use this condition to produce a plot of the stability regions for a single inverted rod, let us make a numerical comparison of our results with exact ones. We will use the approximation `ftmmathieu` to the FTM calculated previously with 20 iterates and choose two values of b , $b = 0.1$ and $b = 1.5$, in order to find the values of a corresponding to the stability boundaries at these values of b . We do that as follows.

```
In[26]:= With[{trftm = Expand[Tr[ftmmathieu /. b -> .1]]},
  Sort[Select[Join[a /. NSolve[trftm == -2, a],
    a /. NSolve[trftm == 2, a]], Im[#1] == 0 &]]]
Out[26]:= {-0.00497832, 0.198781, 0.298719, 0.993783, 1.01023, 2.2429}
```

```
In[27]:= With[{trftm = Expand[Tr[ftmmathieu /. b -> 1.5]]},
  Sort[Select[Join[a /. NSolve[trftm == -2, a],
    a /. NSolve[trftm == 2, a]], Im[#1] == 0 &]]]
Out[27]:= {-0.708598, -0.696345, 0.62976, 0.81923, 1.51241, 2.3711}
```

For the sake of comparison, the exact values corresponding to the approximate ones, are related to the so-called Mathieu characteristic numbers implemented in *Mathematica*.

```
In[28]:= Sort[ $\frac{1}{4}$  Join[Table[MathieuCharacteristicA[r, -2 0.1], {r, 0, 3}],
  Table[MathieuCharacteristicB[r, -2 0.1], {r, 2}]]]
Out[28]:= {-0.00497832, 0.198781, 0.298719, 0.999167, 1.00414, 2.25059}
```

```
In[29]:= Sort[ $\frac{1}{4}$  Join[Table[MathieuCharacteristicA[r, -2 1.5], {r, 0, 3}],
  Table[MathieuCharacteristicB[r, -2 1.5], {r, 2}]]]
Out[29]:= {-0.708598, -0.696345, 0.62976, 0.81923, 1.5113, 2.30578}
```

Instead of plotting the curves where equation (15) holds in variables a and b , it is interesting to use physical variables related to the pendula. So we define b (dimensionless amplitude) and v (dimensionless frequency) as

$$b = \frac{\epsilon}{l} \quad (16)$$

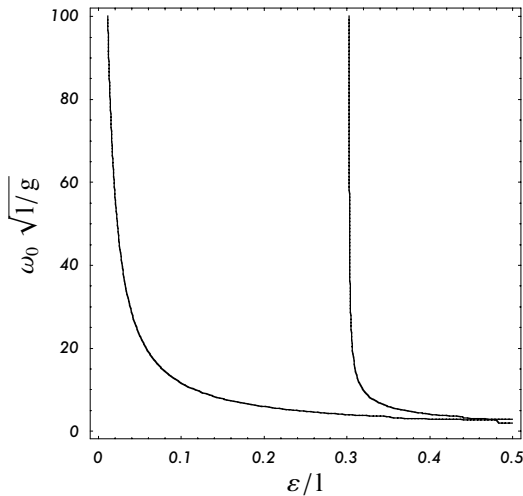
and

$$v = \omega_0 \sqrt{\frac{l}{g}}. \quad (17)$$

The relationship between these variables and a and b is $a = -\frac{3}{2v^2}$, $b = \frac{3}{2}b$. In order to get a good graphic, we need to use a large value for the option `Plot`.

Points of ContourPlot. It takes a reasonable time to run, but the resulting graph is as follows.

```
In[30]:= Timing[pend1 =
  With[{matrix = ftmmathieu /. {a -> - 3
    2 v^2, b -> 3 h
    2}}, ContourPlot[
    Tr[matrix], {h, 0, 0.5}, {v, .1, 100}, Contours -> {-2, 2},
    ContourShading -> False, PlotPoints -> 120, FrameLabel ->
    {StyleForm["\!\(\epsilon/1\)", "TR", FontSize -> 10], StyleForm[
    "\!\(\omega_0\)\!\(\@1/g\)\)", "TR", FontSize -> 10]}];]
```



```
Out[30]= {118.15 Second, Null}
```

We claim that the inverted rod is stable if the frequency and amplitude of the driving are such that the corresponding point lies in the region between the two curves. To see that, we choose an arbitrary point in this region and check stability by computing the corresponding eigenvalues of the FTM. For example, the point (0.2, 40) in (b, v) coordinates lies in the mentioned region. The absolute values of the eigenvalues of the FTM are thus.

```
In[31]:= Abs[Eigenvalues[ftmmathieu /. {a -> - 3
    2 40^2, b -> 3 0.2
    2}]]]
```

```
Out[31]= {1., 1.}
```

This confirms our claim! We see that for reasonably small amplitudes, such as $\frac{\epsilon}{l} < 0.3$, the inverted rod is stable (in the linear approximation) if the frequency is high enough. For larger amplitudes, the inverted rod is stable only in a precise range of frequencies.

□ Two Rods

The left-hand sides of the Euler–Lagrange equations of motion for a system of two pendula, restricted to the simpler case of equal masses, equal lengths, and uniform mass distributions follow.

$$\begin{aligned}
 \text{In}[32]:= & \text{nlefts} = \text{Simplify}\left[\frac{\partial_t (\partial_{\theta[1]'} L[2]) - \partial_{\theta[1]} L[2], \partial_t (\partial_{\theta[2]'} L[2]) - \partial_{\theta[2]} L[2]}{\left\{ m[i_]\rightarrow m, l[i_]\rightarrow l, H[i_]\rightarrow \frac{1}{2}, R[i_]\rightarrow \frac{1}{\sqrt{3}} \right\}} \right] \\
 \text{Out}[32]= & \left\{ \frac{1}{6} l m \left(-9 g \sin[\theta[1][t]] + 3 l \sin[\theta[1][t] - \theta[2][t]] \theta[2]'[t]^2 - \right. \right. \\
 & 9 \sin[\theta[1][t]] \eta''[t] + 9 \cos[\theta[1][t]] \xi''[t] + \\
 & 8 l \theta[1]''[t] + 3 l \cos[\theta[1][t] - \theta[2][t]] \theta[2]''[t] \left. \right), \\
 & \frac{1}{6} l m \left(-3 g \sin[\theta[2][t]] - 3 l \sin[\theta[1][t] - \theta[2][t]] \theta[1]'[t]^2 - \right. \\
 & 3 \sin[\theta[2][t]] \eta''[t] + 3 \cos[\theta[2][t]] \xi''[t] + \\
 & \left. \left. 3 l \cos[\theta[1][t] - \theta[2][t]] \theta[1]''[t] + 2 l \theta[2]''[t] \right) \right\}
 \end{aligned}$$

Notice the presence of several nonlinear terms. Again, the first step in linearizing the equations around the equilibrium $\theta_1 = 0, \theta_2 = 0$ is to replace sines and cosines by their Taylor expansions, retaining terms of at most order 1. But now, this does not eliminate all nonlinearities. In order to finish the linearization procedure, we first create a function to detect the degree of a monomial in a given variable and a Boolean function that tests for monomials of degree not larger than 1.

```

In[33]:= deg[c_, var_] := 0 /; FreeQ[c, var];
deg[c_. var_n_, var_] := n /; FreeQ[c, var]

In[35]:= notlargerthan1[x_, listvars_List] :=
  Plus@@(deg[x, #1] &) /@ listvars ≤ 1

```

Then we use it to select only the terms of degree not larger than 1 in all variables $\theta_1, \theta_2, \theta_1', \theta_2', \theta_1'', \theta_2''$.

$$\begin{aligned}
 \text{In}[36]:= & \text{lhs} = \frac{1}{m l^2} \\
 & \left((\text{Select}[\#1, \text{notlargerthan1}[\#1, \{\theta[1][t], \theta[2][t], \theta[1]'[t], \theta[2]'[t], \right. \\
 & \quad \left. \theta[1]''[t], \theta[2]''[t]\} \& \&]) /@ \right. \\
 & \quad \left. \text{Expand}[\text{nlefts} /. \{\sin[x_] \rightarrow x, \cos[x_] \rightarrow 1\}] \right) \\
 \text{Out}[36]= & \left\{ \frac{1}{l^2 m} \left(-\frac{3}{2} g l m \theta[1][t] - \frac{3}{2} l m \theta[1][t] \eta''[t] + \right. \right. \\
 & \frac{3}{2} l m \xi''[t] + \frac{4}{3} l^2 m \theta[1]''[t] + \frac{1}{2} l^2 m \theta[2]''[t] \left. \right), \\
 & \frac{1}{l^2 m} \left(-\frac{1}{2} g l m \theta[2][t] - \frac{1}{2} l m \theta[2][t] \eta''[t] + \frac{1}{2} l m \xi''[t] + \right. \\
 & \left. \left. \frac{1}{2} l^2 m \theta[1]''[t] + \frac{1}{3} l^2 m \theta[2]''[t] \right) \right\}
 \end{aligned}$$

We would like to write the previous linear expression in matrix form. In order to obtain the matrices, we use the command `LinearExpressionToMatrix` in the `Developer`` context (or, equivalently, transform the expression into a system of equations and then use the `LinearEquationstoMatrices` command in the standard package `LinearAlgebra`MatrixManipulation``).

`In[37]:= << Developer``

The matrix that serves as the coefficient for the vector $\theta''(t) = (\theta_1''(t), \theta_2''(t))$ is obtained as

`In[38]:= A = LinearExpressionToMatrix[lhs, {θ[1]''[t], θ[2]''[t]}}][[1]]`

`Out[38]=` $\left\{ \left\{ \frac{4}{3}, \frac{1}{2} \right\}, \left\{ \frac{1}{2}, \frac{1}{3} \right\} \right\}$

The coefficient of the vector $\theta(t) = (\theta_1(t), \theta_2(t))$ is

`In[39]:= LinearExpressionToMatrix[lhs, {θ[1][t], θ[2][t]}}][[1]]`

`Out[39]=` $\left\{ \left\{ -\frac{3g}{2l} - \frac{3\eta''[t]}{2l}, 0 \right\}, \left\{ 0, -\frac{g}{2l} - \frac{\eta''[t]}{2l} \right\} \right\}$

and the remaining terms are

`In[40]:= remainder =
Simplify[lhs - (A.{θ[1]''[t], θ[2]''[t]} + %.{θ[1][t], θ[2][t]})]`

`Out[40]=` $\left\{ \frac{3\xi''[t]}{2l}, \frac{\xi''[t]}{2l} \right\}$

Notice that the remainder vanishes in the case of vertical driving, that is, $\xi(t) = 0$. Using the cosine vertical driving $\eta(t) = \epsilon \cos \omega_0 t$, the term proportional to $\theta(t)$ becomes

`In[41]:= %% /. η''[t] → ∂t,2 (ε Cos[ω0 t])`

`Out[41]=` $\left\{ \left\{ -\frac{3g}{2l} + \frac{3\epsilon \cos[t \omega_0] \omega_0^2}{2l}, 0 \right\}, \left\{ 0, -\frac{g}{2l} + \frac{\epsilon \cos[t \omega_0] \omega_0^2}{2l} \right\} \right\}$

and again rescaling the time as in equation (11), we arrive finally at

$$A \theta''(\tau) + \left(-\frac{g}{\omega_0^2 l} + \frac{\epsilon}{l} \cos \tau \right) B \theta(\tau) = 0, \quad (18)$$

where A was previously obtained as

$$A = \begin{pmatrix} 4/3 & 1/2 \\ 1/2 & 1/3 \end{pmatrix} \quad (19)$$

and B is the diagonal matrix

$$B = \begin{pmatrix} 3/2 & 0 \\ 0 & 1/2 \end{pmatrix}. \quad (20)$$

$$\text{In}[42]:= \mathbf{B} = \text{Simplify}\left[\frac{\%}{-\frac{g}{1} + \frac{\epsilon \cos[\omega_0 \tau] \omega_0^2}{1}}\right]$$

$$\text{Out}[42]= \left\{\left\{\frac{3}{2}, 0\right\}, \left\{0, \frac{1}{2}\right\}\right\}$$

Multiplying both sides of equation (18) on the left by A^{-1} and realizing that

$$\text{In}[43]:= \text{Inverse}[\mathbf{A}] . \mathbf{B}$$

$$\text{Out}[43]= \left\{\left\{\frac{18}{7}, -\frac{9}{7}\right\}, \left\{-\frac{27}{7}, \frac{24}{7}\right\}\right\}$$

we can rewrite it into the form of equation (4) by making $x_1 = \theta_1$, $x_2 = \theta_2$, $x_3 = \theta'_1$, $x_4 = \theta'_2$, with the matrix $P(\tau)$ in equation (4) being

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{18}{7} \left(-\frac{g}{\omega_0^2 l} + \frac{\epsilon}{l} \cos \tau\right) & \frac{9}{7} \left(-\frac{g}{\omega_0^2 l} + \frac{\epsilon}{l} \cos \tau\right) & 0 & 0 \\ \frac{27}{7} \left(-\frac{g}{\omega_0^2 l} + \frac{\epsilon}{l} \cos \tau\right) & -\frac{24}{7} \left(-\frac{g}{\omega_0^2 l} + \frac{\epsilon}{l} \cos \tau\right) & 0 & 0 \end{pmatrix}$$

and obeying equation (5) with $T = 2\pi$.

At this point, we can calculate the twentieth approximation to the FTM as

$$\begin{aligned} \text{In}[44]:= & \text{ftmrod2} = \text{FloquetMatrix}[\\ & \{ \{0, 0, 1, 0\}, \{0, 0, 0, 1\}, \{-18 (a + b \cos[\#1]), 9 (a + b \cos[\#1]), \\ & 0, 0\}, \{27 (a + b \cos[\#1]), -24 (a + b \cos[\#1]), 0, 0\} \} \&, 20]; \end{aligned}$$

where we should have in mind that

$$a = -\frac{g}{7 \omega_0^2 l} \equiv -\frac{1}{7 v^2},$$

$$b = \frac{\epsilon}{7 l} \equiv \frac{b}{7},$$

and the dimensionless variables b and v are the same as defined in equations (16) and (17).

$$\text{In}[45]:= \text{ftmrod2} = \text{ftmrod2} /. \left\{a \rightarrow -\frac{1}{7 v^2}, b \rightarrow \frac{b}{7}\right\};$$

In order to produce a picture of the stability boundaries for the double rod in the dimensionless variables, in analogy with what was done in the case of one rod, we must locate the points where (in absolute value) the maximum eigenvalue of the Floquet matrix changes from greater than one to equal to one. In the case of the single rod, this was conveniently accomplished by the trace condition of equation (15), but that does not hold anymore.

As we do not know any condition similar to equation (15) to use instead, the best we can do is locate some points in the stability boundaries. We may do that by using an idea similar to the *bisection method* for solving equations. Whenever the

largest eigenvalue (in absolute value) of the FTM is greater than one at some point P and equal to one at another point Q , then there must exist a point between P and Q through which a stability boundary passes. We may approximate this point by taking the point R halfway between P and Q (hence the name bisection) and calculating at R the maximum eigenvalue of the FTM in absolute value. If this number is equal to one, then the stability boundary must pass between P and R , otherwise it must pass between R and Q . By repeating the bisection procedure a sufficient number of times we may approximate a point in the stability boundary between P and Q with an error smaller than any prescribed tolerance.

By experience we know that the stability boundary lies in the region $0 \leq b \leq 0.10$, $0 < v \leq 100$. We then produce a grid of points in that region and calculate the largest eigenvalue of the FTM in absolute value at each point in the grid.

```
In[46]:= pointgrid = Map[{{#1[[1]], #1[[2]]},
    Max[Abs[Eigenvalues[ftmrod2 /. {h -> #1[[1]], v -> #1[[2]]}]]]} &,
    Table[{x, y}, {x, 0, 0.10, .01}, {y, 0.10, 100, 10}], {2}];
```

In order to select those pairs of points in the grid between which we know that the stability boundary passes, we create the following Boolean function and use it.

```
In[47]:= locstabchange[{{{_, _}, m1_}, {{{_, _}, m2_}}} :=
    Chop[m1 - 1] == 0 && Chop[m2 - 1] > 0 || Chop[m2 - 1] == 0 && Chop[m1 - 1] > 0
```

```
In[48]:= Short[verticalpairs = Flatten[(Select[#1, locstabchange] &) /@
    (Partition[#1, 2, 1] &) /@ pointgrid, 1], 5]
```

```
Out[48]//Short= {{{{0.02, 80.1}, 1.01665}, {{0.02, 90.1}, 1.}},
    {{{0.03, 50.1}, 1.04569}, {{0.03, 60.1}, 1.}},
    << 6 >>, {{{0.09, 20.1}, 1.}, {{0.09, 30.1}, 1.53615}}}
```

Notice that because of the particular way `pointgrid` was produced, we have selected pairs of points in the grid with the same horizontal coordinate, that is, points on vertical lines. Later we will repeat the procedure for selecting points lying on horizontal lines.

We can now create the function `bisect` that takes each pair of points in the output of the previous line and calculates the midpoint of the pair, the largest eigenvalue in absolute value of the FTM at the midpoint, and then selects from among the three points the two between which the stability boundary must lie.

```
In[49]:= bisect[{{fir : {{x1_, y1_}, m1_}, sec : {{x2_, y2_}, m2_}}} :=
    Module[{mid =  $\frac{1}{2}$  (fir[[1]] + sec[[1]]), eigenmid},
    eigenmid = Max[Abs[Eigenvalues[ftmrod2 /. {h -> mid[[1]], v -> mid[[2]]}]]];
    Select[{{fir, {mid, eigenmid}}, {mid, eigenmid}, sec}],
    locstabchange][[1]]]
```

If we want to locate the point in the stability boundary between each pair of selected points, it is necessary to iterate `bisect` the right number of times. If the tolerance `tol` is specified, that can be done by the following function.

```

In[50]:= findbybisection[{fir_, sec_}, tol_] :=
  With[{bis = Nest[bisect, {fir, sec},
    Max[1, Ceiling[Log[2,  $\frac{\text{Max}[\text{Abs}[\text{fir}[[1]] - \text{sec}[[1]]]}{\text{tol}}$ ]]]]}],
     $\frac{1}{2} (\text{bis}[[1, 1]] + \text{bis}[[2, 1]])]$ 

```

So, this finds with tolerance 0.1 (in the vertical direction) some points in the stability boundaries.

```

In[51]:= stabpoints1 = (findbybisection[#1, .1] &) /@ verticalpairs;

```

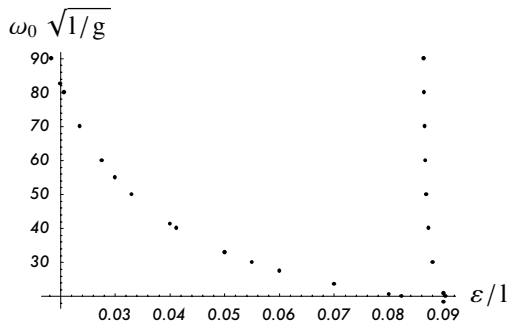
We may now repeat the procedure using `Transpose[pointgrid]` instead of `pointgrid`. That will produce pairs of points lying on horizontal lines between which some point in the stability boundaries must exist. Of course, as the scale of the horizontal axes in the picture is smaller, we must accordingly reduce the tolerance `tol`.

```

In[52]:= horizontalpairs = Flatten[(Select[#1, locstabchange] &) /@
  (Partition[#1, 2, 1] &) /@ Transpose[pointgrid], 1];
stabpoints2 = (findbybisection[#1, .0001] &) /@ horizontalpairs;

In[54]:= stabpointspicture =
  ListPlot[Join[stabpoints1, stabpoints2], PlotStyle -> PointSize[.01],
    AxesLabel -> {StyleForm["\!(\epsilon/l)", "TR", FontSize -> 10],
      StyleForm["\!(\omega\_0)\ \!(\@ (1/g)\)", "TR", FontSize -> 10]}}]

```



An alternative exists for generating this graph: we may use our Mathieu equation results together with the introduction of *normal coordinates* that decouple equation (18) into two separate Mathieu equations. These normal coordinates, which we now briefly explain, were used by Acheson in the proof of his pendulum theorem [7].

Notice that as the matrix $A^{-1} B$ has two distinct real eigenvalues, then it must be diagonalizable over the real numbers. Let U be the orthogonal matrix that diagonalizes $A^{-1} B$. We define the *normal coordinates* ψ as $\psi = U^{-1} \theta$. Substituting θ by $U \psi$ in equation (18) and then left multiplying it by $U^{-1} A^{-1}$, we obtain

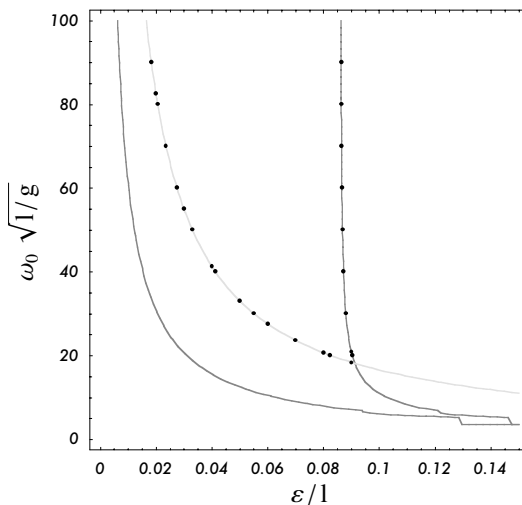
$$\psi''(\tau) + U^{-1} A^{-1} B U \left(-\frac{g}{\omega_0^2 l} + \frac{\epsilon}{l} \cos \tau \right) \psi(\tau) = 0,$$

which, because $U^{-1} A^{-1} B U$ is diagonal, decouples into

$$\psi_i''(\tau) + \mu_i \left(-\frac{g}{\omega_0^2 l} + \frac{\epsilon}{l} \cos \tau \right) \psi_i(\tau) = 0, \quad (21)$$

where $i = 1, 2$ and μ_i are the eigenvalues of $A^{-1} B$. These are two Mathieu equations (7) with $a = -\frac{g}{\omega_0^2 l} \mu_i \equiv -\frac{\mu_i}{v^2}$ and $b = \frac{\epsilon}{l} \mu_i \equiv \mu_i b$.

As the original variables θ_1, θ_2 are linear combinations of the normal variables ψ_1, ψ_2 , then the equilibrium solution of equation (18) will be stable if and only if the equilibrium solution for both equations (21), $i = 1, 2$ is stable. We may then determine the stability region for the linearized double rod in equation (18) by intersecting the two stability regions for Mathieu equations. This can be done by using the already calculated approximate FTM for the Mathieu equation. In the following result, the red curve is the stability picture for the mode $i = 1$, the cyan curve corresponds to the mode $i = 2$, and the points over the curves are the ones calculated previously. Notice that both ways of producing the picture lead to indistinguishable results!



In order to check the numerical accuracy of one of the points in the picture, we may use the built-in Mathieu characteristic numbers together with the normal coordinates method.

```
In[55]:= stabpoints1[[5]]
```

```
Out[55]:= {0.06, 27.5609}
```

Given that the first coordinate is 0.06, here is the exact value for the second coordinate of the chosen point.

```
In[56]:= With[{eigenval = Min[Eigenvalues[Inverse[A].B]]},  
             Sqrt[-eigenval / (1/4 MathieuCharacteristicA[0, -2 eigenval %[[1]]])]]
```

```
Out[56]:= 27.5568
```

This is in perfect agreement with the value obtained by our method. Remember that the second coordinate of that point was calculated by the bisection method with a tolerance of 0.1.

It can be shown that the method of normal coordinates works for any number of rods, not only for two. The procedure for producing the previous graph for three or more rods is the same and we do not reproduce it here, leaving it as an exercise for the reader.

■ 6. Conclusions

We explained our implementation of a method for producing very good approximations to solutions of linear differential equations with periodic coefficients. The method is based on calculating exact high-order Picard iterates by using a fast integration function and also works for equations or initial conditions depending on parameters. As a first accuracy test, we compared the numerical values for the first Mathieu characteristic numbers as predicted by our method (with 20 iterates) and the ones calculated by built-in *Mathematica* functions. The agreement was very good for the parameter values considered. Of course, the built-in functions are much faster, but we cannot forget that the proposed method is very general and the Mathieu equation is just an interesting example.

We also visually compared the stability chart for the vertically driven upside-down double pendulum obtained by two different methods, both based on calculating Picard iterates. The first method was to directly calculate the Picard iterates for a system of four equations and the second was to decouple the system of four equations into two Mathieu equations, for which we had already calculated the necessary Picard iterates. Again, the results with both methods were visually indistinguishable, providing an indirect check of the accuracy of the Picard iterates. We also checked directly that one of the points in that stability chart agreed numerically with the exact calculated value within the tolerance of the bisection method used.

By running the *FastPicard* package with *Mathematica* 4.1 on a 1.4 GHz Pentium 4 with 256 MB of RAM, we were able to calculate up to 50 iterates for the Mathieu equation in less than one hour. Results were reported in [6].

We should also note that there exist mathematically rigorous upper bounds on the difference between the exact solution of an equation and the n th Picard iterate approximating it. These bounds decrease exponentially fast in n . In principle, they can be used with symbolic Picard iterates to give computer-assisted proofs of properties of solutions for differential equations depending on parameters.

We used the phrase “in principle” in the previous paragraph for the following reason. The reader should have noticed from the example with the Mathieu equation that the errors committed in approximating solutions by a fixed number of Picard iterates increase with time. In our stability examples, it was necessary to approximate the solution of the differential equations up to time 2π . This is indeed quite a long time interval for the number of iterates we are able to calculate. As a consequence, the error bounds are not very interesting. In applications where the time interval is shorter, it is possible that the error bounds could produce interesting proofs.

There also exist rigorous bounds on the errors committed by most traditional numerical methods. But these methods are difficult to implement in the case of equations depending on parameters and, even when the equations do not depend on parameters, the calculations in these methods are subject to roundoff errors that are difficult to overcome.

Therefore we propose exact Picard iteration as an alternative method for producing approximate solutions to linear differential equations with periodic coefficients depending on parameters. For problems in which the time intervals are short, we may even provide interesting upper bounds for the errors due to calculating only a finite number of iterates.

■ Acknowledgments

This work was partially supported by FAPEMIG, the Research Funding Foundation of the State of Minas Gerais, Brazil.

■ References

- [1] S. C. Sinha and E. A. Butcher, “Symbolic Computation of Fundamental Solution Matrices for Linear Time-Periodic Dynamical Systems,” *Journal of Sound and Vibration*, **206**(1), 1997 pp. 61–85.
- [2] E. A. Butcher and S. C. Sinha, “Symbolic Computation of Local Stability and Bifurcation Surfaces for Nonlinear Time-Periodic Systems,” *Nonlinear Dynamics*, **17**(1), 1998 pp. 1–21.
- [3] A. H. Nayfeh, *Perturbation Methods*, New York: John Wiley & Sons, 1973.
- [4] J. A. Sanders and F. Verhulst, *Averaging Methods in Nonlinear Dynamical Systems*, New York: Springer-Verlag, 1985.

- [5] M. Braun, *Differential Equations and Their Applications*, 2nd ed., New York: Springer-Verlag, 1978.
- [6] A. G. M. Neves, "Symbolic Computation of High-Order Exact Picard Iterates for Systems of Linear Differential Equations with Time-Periodic Coefficients," in *Computational Science—ICCS 2003, International Conference, Melbourne, Australia and St. Petersburg, Russia, June 2003, Proceedings, Part 1*; Lecture Notes in Computer Science, Vol. 2657, pp. 838–847, Heidelberg: Springer-Verlag, 2003.
- [7] D. J. Acheson, "A Pendulum Theorem," *Proceedings of the Royal Society of London, Series A, Mathematical and Physical Sciences*, **443**, 1993 pp. 239–245.
- [8] D. J. Acheson and T. Mullin, "Upside-down Pendulums," *Nature*, **366**, 1993 pp. 215–216.
- [9] M. V. Bartuccelli, G. Gentile, and K. V. Georgiou, "On the Dynamics of a Vertically Driven Damped Planar Pendulum," *Proceedings of the Royal Society of London, Series A, Mathematical and Physical Sciences*, **457**, 2001 pp. 3007–3022.
- [10] F. A. Alhargan, "Algorithms for the Computation of All Mathieu Functions of Integer Orders," *ACM Transactions on Mathematical Software (TOMS)*, **26**(3), 2000 pp. 390–407.
- [11] D. Frenkel and R. Portugal, "Algebraic Methods to Compute Mathieu Functions," *Journal of Physics A, Mathematical and General*, **34**, 2001 pp. 3541–3551.
- [12] A. G. M. Neves, "Upper and Lower Bounds on Mathieu Characteristic Numbers of Integer Orders," *Communications on Pure and Applied Analysis*, **3**(3), 2004 pp. 447–464.
- [13] R. E. Maeder, *The Mathematica Programmer*, Boston: AP Professional, 1994.
- [14] S. Wolfram, *The Mathematica Book*, 4th edition, Champaign, Oxford: Wolfram Media/Cambridge University Press, 1999.
- [15] A. Stephenson, "On a New Type of Dynamical Stability," *Memoirs and Proceedings of the Manchester Literary and Philosophical Society*, **52**(8), 1908 pp. 1–10.
- [16] A. Stephenson, "On Induced Stability," *Philosophical Magazine*, **15**, 1908 pp. 233–236.
- [17] H. Goldstein, *Classical Mechanics*, 2nd ed., Reading, MA: Addison-Wesley, 1980.
- [18] D. W. Jordan and P. Smith, *Nonlinear Ordinary Differential Equations*, 2nd ed., Oxford: Clarendon Press, 1987.

■ Additional Material

FastPicard.m

Available at www.mathematica-journal.com/issue/v10i1/download.

About the Author

Armando G. M. Neves majored in physics in 1986 at the Federal University of Minas Gerais (UFMG) in Brazil. After receiving a master's degree, he obtained his doctoral degree in physics at Rome University "La Sapienza" in Italy in 1993. Although his degrees are in physics, he has always searched for mathematical rigor. Since 1992 he has held a position in the Mathematics Department of UFMG.

His general research interest is mathematical physics. His principal interests are in statistical mechanics, quantum field theory, and, more recently, dynamical systems. Although his first academic works were noncomputational, after using *Mathematica* he realized that symbolic computation opened up entirely new fields of problems and provided new ways of attacking older problems.

Armando G. M. Neves

UFMG - Departamento de Matemática

Av. Antônio Carlos, 6627 - Caixa Postal 702

30123-970 Belo Horizonte - MG

BRAZIL

aneves@mat.ufmg.br

www.mat.ufmg.br/~aneves

A Flexible Implementation for Support Vector Machines

Roland Nilsson
Johan Björkegren
Jesper Tegnér

Support vector machines (SVMs) are learning algorithms that have many applications in pattern recognition and nonlinear regression. Being very popular, SVM software is available in many versions. Still, existing implementations, usually in low-level languages such as C, are often difficult to understand and adapt to specific research tasks. In this article, we present a compact and yet flexible implementation of SVMs in *Mathematica*, traditionally named *MathSVM*. This software is designed to be easy to extend and modify, drawing on the powerful high-level language of *Mathematica*.

■ Background

A *pattern recognition problem* amounts to learning how to discriminate between data points x_i belonging to two *classes*, defined by class labels $y_i \in \{+1, -1\}$, when given only a set of examples (x_i, y_i) from each class. These problems are found in various applications, from automated handwriting recognition to medical expert systems, and pattern recognition or *machine learning* algorithms are routinely applied to solve them.

It may be helpful for newcomers to relate this to a more familiar problem: standard statistical hypothesis testing for one-dimensional x_i , such as Student's t -test [1, ch. 8], can be viewed as a very simple kind of pattern recognition problem. Here, the hypotheses H_0 and H_1 correspond to classes $+1, -1$, and the familiar

$$g(x) = \begin{cases} H_0, & \text{if } x < \bar{x} \\ H_1, & \text{if } x > \bar{x} \end{cases},$$

where $\bar{x}$ is the mean of within-class means,

$$\bar{x} = \frac{1}{2} \left(\frac{1}{l_1} \sum_{i: y_1 = +1} x_i + \frac{1}{l_{-1}} \sum_{i: y_1 = -1} x_i \right),$$

$l_c = |\{i: y_1 = c\}|$ and $H_0 < H_1$, is called the *decision rule* or sometimes simply the *classifier*. We say that the decision rule g is induced from data x_i , in this case determined by computing $\bar{x}$.

However, real pattern recognition problems usually involve high-dimensional data (such as image data) and unknown underlying distributions. In this situation, it is nearly impossible to develop statistical tests like the preceding one. These problems are typically attacked with algorithms, such as artificial neural networks [2], decisions trees [3, ch. 18], Bayesian models [4], and recently SVMs [5], to which we will devote the rest of this article. Here we will only consider data that can be represented as vectors $x \in R^n$; other kinds of information can usually be changed to this form in some appropriate manner.

■ Support Vector Machines

SVMs attempt to find a hyperplane $\Pi_{w,b} = w \cdot x + b = 0$, $x \in R^n$ that separates the data points x_i (meaning that all x_i in a given class are on the same side of the plane), corresponding to a decision rule

$$g(x) = \text{sign}(w \cdot x + b).$$

In SVM literature, w is often referred to as the *weight vector*; b is called the *bias* (a term adopted from neural networks). This idea is not new; it dates back at least to R.A. Fisher and the theory of *linear discriminants* [6]. The novelty of SVMs lies in how this plane is determined: SVMs choose the separating hyperplane $w \cdot x + b = 0$ that is furthest away from the data points x_i , that is, that has maximal *margin* (Figure 1). The underlying idea is that a hyperplane far from any observed data points should minimize the risk of making wrong decisions when classifying new data. To be precise, in SVMs we maximize the distance to the closest data points. We solve

$$\max_{(w,b)} (\min_i d(\Pi_{w,b}, x_i)), \quad (1)$$

where $d(\Pi_{w,b}, x_i) = |w \cdot x_i + b| / \|w\|$ is the distance between data point i and the plane Π_w , subject to the constraint that this plane still separates the classes. The plane Π_w that solves (1) is called the *optimal separating hyperplane* and is unique [5]. *MathSVM* provides algorithms for determining this plane from data.

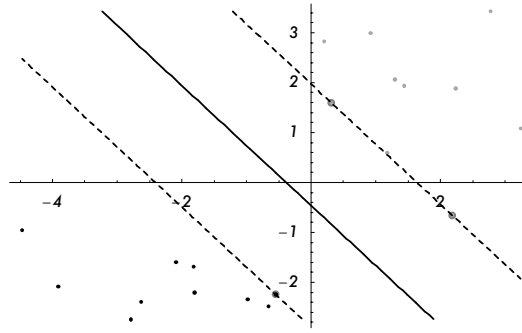


Figure 1. Two-class data (black and grey dots), their optimal separating hyperplane (continuous line), and support vectors (circled in blue). This is an example output of the `SVMPLOT` function in *MathSVM*. The width of the “corridor” defined by the two dotted lines connecting the support vectors is the margin of the optimal separating hyperplane.

■ Solving the Optimization Problem

□ The Primal Problem

It turns out that the optimal separating hyperplane solving (1) can be found as the solution to the equivalent optimization problem

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2} \|w\|^2 \\ \text{subject to} \quad & y_i(w^T x_i + b) \geq 1, \end{aligned} \quad (2)$$

referred to as the *primal* problem. Typically, only a small subset of the data points will attain equality in the constraint; these are termed *support vectors* since they are “supporting” (constraining) the hyperplane (Figure 1). In fact, the solution (w, b) depends only on these specific points. Therefore, the method also is a scheme for data compression, in the sense that the support vectors contain all the information necessary to derive the decision rule.

□ The Dual Problem

For reasons that will become clear later, w often has very high dimension, which makes the primal problem (2) intractable. Therefore, we attack (2) indirectly, by solving the *dual problem*

$$\begin{aligned} \min_{\alpha} \quad & \alpha^T Q \alpha - \alpha \\ \text{subject to} \quad & \alpha_i \geq 0, \quad y^T \alpha = 0, \end{aligned} \quad (3)$$

where $Q = (q_{ij}) = (y_i y_j x_j x_i)$. This is a quadratic programming (QP) problem, which has a unique solution whenever Q is positive semidefinite (as is the case here). It is solved numerically in *MathSVM* by the function `QPSolve`. (At this point we need to load the *MathSVM* package; see Additional Material.)

```

In[1]:= << MathSVM`
        << Statistics`NormalDistribution`

In[3]:= ? QPSolve

```

QPSolve[Q,p,a,b,c,y,τ] solves the quadratic programming problem $\min \alpha \cdot Q \cdot \alpha + p \cdot \alpha$, subject to $a \leq \alpha \leq b$ and $y \cdot \alpha = c$. QPSolve uses the GSDO algorithm described by Keerthi et al. τ is a solution tolerance parameter (0.01 or so is usually good enough for SVMs). Q must be a positive semidefinite matrix to guarantee convergence.

The variable α has $\dim \alpha = l$, the number of data points, so the matrix Q has l^2 elements, which may be quite large for large problems. Therefore, QPSolve employs a divide-and-conquer approach [7] that allows for solving (3) efficiently without storing the full matrix Q in memory.

Having solved the dual problem for α using QPSolve, we obtain the optimal weight vector w and bias term b , that is, the solution to the primal problem (2), using the identities

$$w = \sum_i \alpha_i y_i x_i \quad (4)$$

$$b = -\frac{1}{2} (w^T x_+ + w^T x_-), \quad (5)$$

where in (5) x_+ , x_- are any two support vectors belonging to class +1 and -1, respectively (there always exist at least two such support vectors) [5].

□ A Simple SVM Example

Enough theory—let us generate some data and solve a simple SVM problem using *MathSVM*.

```

In[4]:= len = 20;
        X = Join[
            RandomArray[NormalDistribution[-2, 1], {len/2, 2}],
            RandomArray[NormalDistribution[2, 1], {len/2, 2}]];
        y = Join[Table[1, {len/2}], Table[-1, {len/2}]];

```

For this elementary problem, we use the simple SVM formulation (2) provided in *MathSVM* by the SeparableSVM function.

```

In[7]:= τ = 0.01;
        α = SeparableSVM[X, y, τ]

Out[8]:= {0.405345, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0.102479, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.507824}

```

The returned vector α is the solution found by `QPSolve` for the dual formulation (3) of the SVM problem we just constructed. For this specific problem, the dual formulation used is exactly that described by (3), that is

$$\begin{aligned} Q &= (q_{ij}) = (y_i y_j x_j x_i) \\ p &= (-1, \dots, -1) \\ a &= (0, \dots, 0), \quad b = (0, \dots, 0) \\ c &= 0 \end{aligned}$$

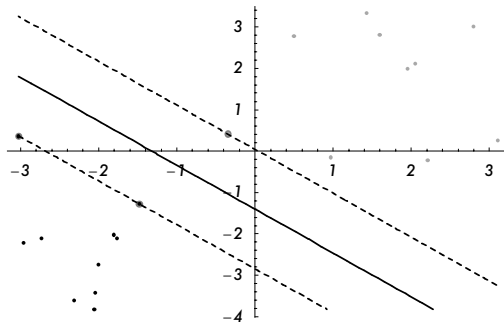
The support vectors are immediately identifiable as the nonzero α_i . (4) and (5) are implemented as

```
In[9]:= WeightVector[α, X, y]
Out[9]= {-0.73531, -0.692296}

In[10]:= Bias[α, X, y]
Out[10]= -0.973496
```

A plot similar to Figure 1 is produced by the `SVMPlot` function. As in Figure 1, the solid line marks the optimal hyperplane, and dotted lines mark the width of the corridor that joins support vectors (highlighted in blue).

```
In[11]:= SVMPlot[α, X, y]
```



□ Nonseparable Data

Often the assumption of separable training data is not reasonable (it may fail even for the preceding simple example). In such cases, `NonseparableSVM` should be used. This SVM variant takes a parameter C that determines how hard points violating the constraint in (2) should be penalized. This parameter appears in the objective function of the primal problem, which now is formulated as [5]

$$\begin{aligned} \min_{w, b, \xi} \quad & \frac{1}{2} \|w \cdot w\| + C \sum_i \xi_i \\ \text{subject to} \quad & y_i(x_i \cdot w + b) \geq 1 - \xi_i, \quad \xi_i \geq 0. \end{aligned}$$

Large C means high penalty, and in the limit $C \rightarrow \infty$ we obtain the separable case.

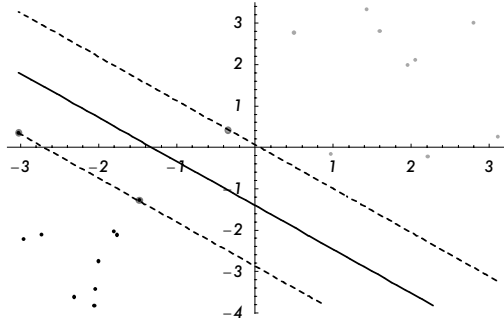
```

In[12]:=  $\tau = 0.01$ ;
 $\alpha = \text{NonseparableSVM}[X, y, 0.5, \tau]$ 

Out[13]:= {0.3991, 0, 0, 0, 0, 0, 0, 0, 0, 0.1009, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.5}

In[14]:=  $\text{SVMPLOT}[\alpha, X, y]$ 

```



□ Getting QP Formulations

Sometimes it is interesting to examine what the QP problem looks like for a given SVM formulation. Using the option `FormulationOnly`, we can inspect the various parameters instead of actually solving the QP. This can, for example, be used to study expressions analytically.

```

In[15]:= Clear[X, y,  $\alpha$ , len]

In[16]:=  $\tau = 0.01$ ;
NonseparableSVM[Array[x, {2, 2}],
  Array[y, {2}], C,  $\tau$ , FormulationOnly  $\rightarrow$  True]

Out[17]:= {{{(x[1, 1]2 + x[1, 2]2) y[1]2,
  (x[1, 1] x[2, 1] + x[1, 2] x[2, 2]) y[1] y[2]},
  {(x[1, 1] x[2, 1] + x[1, 2] x[2, 2]) y[1] y[2],
  (x[2, 1]2 + x[2, 2]2) y[2]2}},
  {-1, -1}, {0, 0}, {C, C}, 0, {y[1], y[2]}, 0.01}

```

The parameters are given in the order corresponding to the arguments of `QPSolve`.

■ Feature Space and Kernels

In all of the preceding examples, the separating surface is assumed to be linear (a hyperplane). This is often a serious limitation, as many pattern recognition problems are inherently nonlinear in the input data and require nonlinear separating surfaces. To overcome this obstacle, for each specific problem we devise some appropriate transformation $\phi(x)$ from *input space* X (the domain of the original data) to a *feature space* H . The function ϕ is chosen so that a hyper-

plane in H corresponds to some desirable class of surfaces in X . (Choosing this function for a specific problem is something of an art, but we can always try a few different ϕ known to have been successful in similar problems and simply hope for the best.)

As an example, consider a second-degree polynomial surface in R^2 , described by $w_1 + w_2 x_1 + w_3 x_2 + w_4 x_1^2 + w_5 x_1 x_2 + w_6 x_2^2 = 0$. We may represent such a surface by choosing a mapping $X \rightarrow H$ as

$$\phi(x_1, x_2) = (1, x_1, x_2, x_1^2, x_1 x_2, x_2^2) \quad (6)$$

and forming a hyperplane in H defined by $w \cdot z + b = 0$, where $z = \phi(x)$. If we now solve the problem in the new variables z , we obtain a wide-margin hyperplane in H corresponding to a second-degree surface in X .

The problem with this approach is that nonlinear ϕ -transformations may have huge dimensions. Even for the simple quadratic surface considered here, we obtain $\dim H = 1 + n + n^2$ when $\dim X = n$. For increasing polynomial degree, this number grows quickly, and there are even nonlinear functions that result in infinite-dimensional H . This is why we do not solve the primal problem (2) directly: w may not be a finite vector, but α always is.

This problem is elegantly solved by *kernels*. It turns out that there are many functions ϕ for which we can compute scalar products $\phi(x) \cdot \phi(y)$ in H implicitly, without actually calculating ϕ . And in fact, this is all we need for solving the SVM formulation (1). Thus we define the kernel function as

$$K(x, y) = \phi(x) \cdot \phi(y)$$

In the case of example (6), it turns out that $K(x, y) = \phi(x) \cdot \phi(y) = (1 + x \cdot y)^2$, which is why we chose that specific form of ϕ (explaining the two separate terms $x_1 x_2$ and $x_2 x_1$). It is not difficult to prove that this result holds for any polynomial degree d ; therefore, using the polynomial kernel

$$K_d(x, y) = (1 + x \cdot y)^d$$

we can obtain any polynomial separating surfaces.

□ A Nonlinear Example: Using Kernels

Let us see how kernels are handled in *MathSVM* to solve nonlinear problems. The second-degree kernel in (6) is provided by

```
In[18]:= PolynomialKernel[x, y, 2]
```

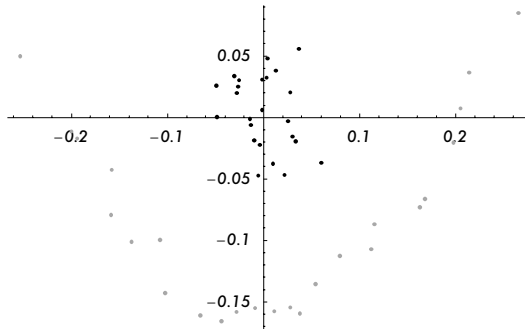
```
Out[18]= (1 + x . y)^2
```

Here is some data a linear classifier cannot possibly cope with.

```

In[19]:= len = 50;
X = Join[
  RandomArray[NormalDistribution[0, 0.03], {len/2, 2}],
  Table[
    {Random[NormalDistribution[i/len - 1/4, 0.01]],
     Random[NormalDistribution[(2 i/len - 1/2)^2 - 1/6, 0.01]]},
    {i, len/2}]];
y = Join[Table[1, {len/2}], Table[-1, {len/2}]];
SVMDDataPlot[X, y, PlotRange -> All]

```



Let us solve this problem using the polynomial kernel. This is done as before, by supplying the desired kernel (which can be any function accepting two arguments) using the `KernelFunction` option.

```

In[23]:= τ = 0.01;
pk = PolynomialKernel[#1, #2, 2] &;
α = SeparableSVM[X, y, τ, KernelFunction -> pk]

Out[25]:= {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  0, 0, 3826.52, 0, 0, 0, 0, 0, 0, 0, 1146.88, 0, 0, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 1644.97, 0, 0, 0, 0, 1034.67, 0}

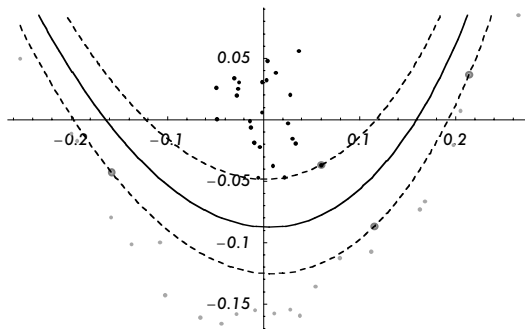
```

When visualizing the results, `SVMPLOT` can use the kernel functions to draw any nonlinear decision curves.

```

In[26]:= SVMPLOT[α, X, y, KernelFunction -> pk]

```



```

In[27]:= Clear[len, X, y, α, pk]

```

□ High-Dimensional Input Spaces

An interesting consequence of the kernel idea is that the dimensionality of the input space X does not matter for the time-complexity of the SVM algorithm. Since the solution is computed using only dot products between samples (in input space or some feature space), high-dimensional problems are solved equally fast (not considering the time used to precalculate the kernel matrix Q , which is usually not noticeable). As an example of this, consider a problem with dimension $n = 1000$.

```
In[28]:= len = 20; n = 1000;
X = Join[
  RandomArray[NormalDistribution[-2, 1], {len/2, n}],
  RandomArray[NormalDistribution[2, 1], {len/2, n}]];
y = Join[Table[1, {len/2}], Table[-1, {len/2}]];
```

The kernel matrix is still just $l \times l$.

```
In[31]:= KernelMatrix[IdentityKernel, X] // Dimensions
```

```
Out[31]= {20, 20}
```

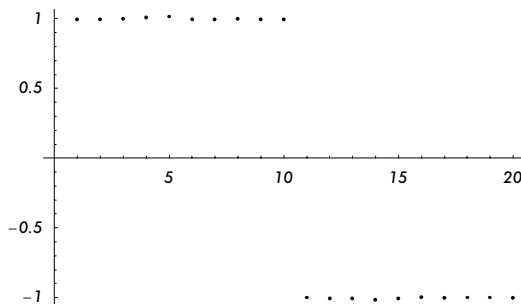
The SVM algorithm is still fast, although the problem is much harder due to extremely low sample density, which is reflected by more support vectors (the nonzero α_i).

```
In[32]:= τ = 0.01;
(α = SeparableSVM[X, y, τ]) // Timing
```

```
Out[33]= {0.79 Second, {0.0000185576, 9.32288 × 10-6,
  0, 0, 0, 0.000031162, 0.0000215259, 0, 0.0000267382,
  0.0000171359, 1.00351 × 10-6, 0.0000297867, 0.0000293384, 0,
  0.0000131243, 0, 0.000024641, 0.000020647, 0, 5.90165 × 10-6}}
```

The solution in this case may be viewed using the projection of data onto the weight vector w . The separation looks almost perfect, although there will be problems with *overfitting* (the solution may not work well when applied to new, unseen examples x). However, this problem is outside the scope of the present article.

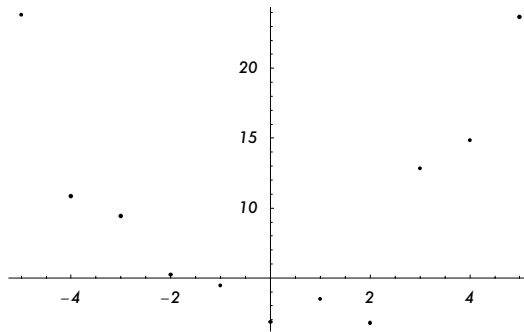
```
In[34]:= ListPlot[X.WeightVector[α, X, y]]
```



■ Regression Analysis with SVMs

So far we have considered SVMs as a tool for pattern recognition only. It is also possible to use the SVM framework for regression problems. Consider a function $y = f(x)$ to be approximated; for example, a quadratic.

```
In[35]:= X = Range[-5, 5]; len = Length[X];
y = Map[#^2 + Random[NormalDistribution[0, 2]] &, X];
ListPlot[Thread[{X, y}]]
```

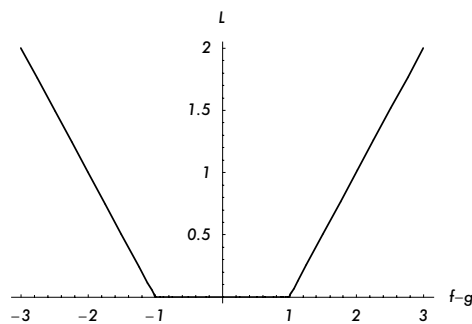


We can adapt the SVM method to the regression setting by using a ϵ -insensitive loss function

$$L_{\epsilon}(f(x), g(x)) = \begin{cases} 0, & \|f(x) - g(x)\| < \epsilon \\ \|f(x) - g(x)\| - \epsilon, & \text{otherwise} \end{cases}$$

where $g(x)$ is the SVM approximation to the regression function $f(x)$. This loss function determines how much a deviation from the true $f(x)$ is penalized; for deviations less than ϵ , no penalty is incurred. Here is what the loss function looks like.

```
In[38]:= Plot[If[Abs[x] < 1, 0, Abs[x] - 1], {x, -3, 3}, AxesLabel -> {"f-g", "L"}]
```



Using this idea, the regression problem is transformed to a classification problem: any x such that $L_{\epsilon}(f(x), g(x)) = 0$ may be considered “correctly classified.” *MathSVM* solves such problems using the *RegressionSVM* function, parameterized by ϵ and a penalty constant C . Here we again try a polynomial kernel.

```

In[39]:= pk = PolynomialKernel[#1, #2, 2] &;
          ε = 3; c = 0.5;
          τ = 0.01;
          α = RegressionSVM[X, y, c, ε, τ, KernelFunction → pk]

Out[42]= {0, 0, 0, 0, 0, 0.0252908, 0, 0, 0, 0,
          0, 0.0133908, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0.0119}

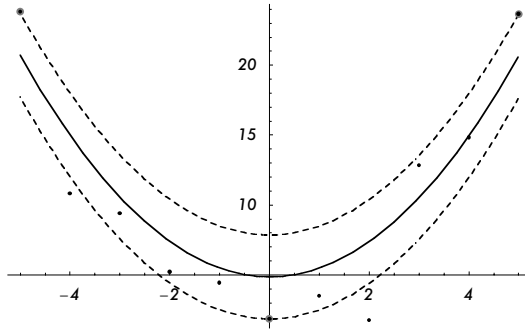
```

The function `RegressionSVMPlot` provides convenient plotting of the resulting regression function. As with `SVMPlot`, the kernel type used is supplied as a parameter. Note how support vectors in this case are chosen as the data points that are furthest away from the regression line.

```

In[43]:= RegressionSVMPlot[α, X, y, ε, KernelFunction → pk]

```



```

In[44]:= RegressionBias[α, X, y, ε, KernelFunction → pk]

Out[44]= 4.83561

```

We can, of course, also obtain the analytical expression of the estimated regression function.

```

In[45]:= RegressionFunction[α, X, y, ε, x, KernelFunction → pk]

Out[45]= 4.81032 + 0.0133908 (1 - 5 x)2 + 0.0119 (1 + 5 x)2

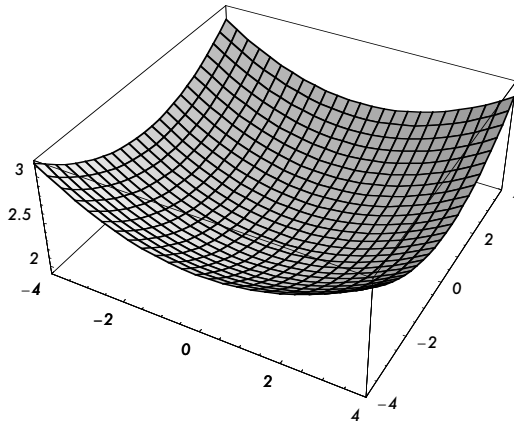
In[46]:= Clear[α, X, y, pk, ε, c, len]

```

□ Two-Dimensional Example

We can use SVM regression with domains of any dimension (that is the main advantage). Here is a simple two-dimensional example.


```
In[56]:= Plot3D[rf, {i, -4, 4}, {j, -4, 4}]
```



■ Conclusion

In this article, we have demonstrated the utility of the *MathSVM* package for solving pattern recognition and regression problems. This is an area of very active research and these algorithms are evolving quickly. In a rapidly moving field such as this, it is important to have a clear, well documented, high-level approach to implementation to minimize confusion. *Mathematica* provides an excellent solution here, due to its high-level programming language and symbolic capabilities.

MathSVM is currently 100% native *Mathematica* code, written with the emphasis on clarity. This does incur penalties in terms of computational speed. Some parts of the QP algorithm are therefore being ported to Java at this time to improve performance. This should not impair the clarity of the software in any way, since the *QPSolve* function is easily separable from the other parts of *MathSVM* in a “black box” fashion.

The *MathSVM* software is still in its infancy and will no doubt expand rapidly, as our group is currently involved in many projects in pattern recognition and high-dimensional data analysis in general, as well as in a biomedical context. We hope that this contribution will initiate other efforts to bring understandable implementations of machine learning algorithms to the *Mathematica* community.

■ References

- [1] G. Casella and R. L. Berger, *Statistical Inference*, 2nd ed., Belmont, CA: Duxbury Press, 2002.
- [2] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed., Englewood Cliffs, NJ: Prentice Hall, 1999.

- [3] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Englewood Cliffs, NJ: Prentice Hall, 1995.
- [4] N. Friedman, "Inferring Cellular Networks Using Probabilistic Graphical Models," *Science*, **303**, 2004 pp. 799–805.
- [5] V. N. Vapnik, *Statistical Learning Theory*, New York: John Wiley & Sons, 1998.
- [6] R. A. Fisher, "The Statistical Utilization of Multiple Measurements," *Annals of Eugenics*, **8**, 1938 pp. 376–386.
- [7] S. S. Keerthi and E. G. Gilbert, "Convergence of a Generalized SMO Algorithm for SVM Classifier Design," *Machine Learning*, **46**, 2002 pp. 351–360.

■ Additional Material

MathSVM.nb
MathSVM.m

Available at www.mathematica-journal.com/issue/v10i1/download.

About the Authors

Roland Nilsson is a graduate student at Linköping University working with machine learning algorithms in analysis of high-dimensional biomedical data.

Johan Björkegren is an associate professor in molecular medicine at Karolinska Institutet, Sweden, and cofounder of Clinical Gene Networks, a biotechnology company involved in system-level analysis of biomedical data in cardiovascular disease.

Jesper Tegnér is a professor of computational biology at Linköping University, Sweden, and cofounder of Clinical Gene Networks.

Roland Nilsson

Computational Biology
Linköping University
SE-58183 Linköping, Sweden
rolle@ifm.liu.se

Johan Björkegren

Center for Genomics and Bioinformatics
Karolinska Institutet
SE-17177 Stockholm, Sweden
johan.bjorkegren@ks.se

Jesper Tegnér

Computational Biology
Linköping University
SE-58183 Linköping, Sweden
jespert@ifm.liu.se

Configuration Analysis and Design by Using Optimization Tools in Mathematica

Frank J. Kampas
János D. Pintér

In engineering, economic, and scientific studies, decisions are frequently modeled by applying optimization concepts and techniques. This article discusses global optimization in multiextremal models and tools to handle such models in *Mathematica*. Since we assume that not all readers are familiar with optimization models and methods, a general modeling framework is presented. We also review several built-in *Mathematica* optimization functions and then introduce *MathOptimizer*, an application package for continuous nonlinear (convex and global) optimization. To illustrate the usage of *MathOptimizer*, several configuration analysis and design models are formulated and solved. We also provide some comparative notes related to current *Mathematica* optimization functionality (namely, the function `NMinimize`) and to the recently introduced *MathOptimizer Professional* package.

■ Introduction

Quantitative decisions related to various engineering, economic, and scientific investigations are frequently modeled by applying optimization concepts and tools. In mathematical terminology, the decision maker or modeler wants to find the “absolutely best” decision vector that corresponds to the minimum (or maximum) of a suitable objective function, while it satisfies a given collection of feasibility constraint functions. The objective function expresses an overall—scalar or scalarized—performance measure, such as profit, utility, loss, risk, or error. The model constraints originate from physical, technical, economic, or possibly some other, considerations.

A large number of such models naturally belong to the realm of “traditional” continuous optimization, notably, linear and convex nonlinear programming.

These subjects are covered by most operations research/management science texts [1–21].

There exist numerous practically important optimization problems in which the phenomena and processes modeled are highly nonlinear and/or are evaluated computationally. In many cases, the necessary structural (convexity) requirements of traditional optimization are not satisfied or may not be simply verifiable. Nonconvex objective functions often possess a multitude of local optima; nonconvex feasible sets also may lead to a similar situation, even for convex objective functions. The subject of global optimization is the theory and application of seeking the “absolutely best” solution in models that have a number of local optima. For discussions of highly nonlinear models and of global optimization, consult, for example, [22–44].

Global optimization (GO) models and strategies have been analyzed since the 1950s. For example, think of the origins of “naturally inspired” genetic and evolutionary searches, simulated annealing, or random search methods. To our best knowledge, the first systematic collection of English GO papers is found in the volumes edited by Dixon and Szegö [45]. Around the same time, several topical books in Russian also became available. See, for example, the related review and references in [35].

As of 2004, there exist well over one hundred books, as well as many thousands of research articles, that discuss GO theory, models, methods, software, and applications. In addition to the works listed earlier, we refer the reader to the *Handbook of Global Optimization* volumes edited by Horst and Pardalos [46] and by Pardalos and Romeijn [47]. The *Journal of Global Optimization*, which has been published since 1991, is devoted entirely to the subject.

□ The Continuous Global Optimization Model

For the purposes of this article, we assume that discrete optimization variables are absent and that all model functions are continuous. This leads to the corresponding specific case of the mathematical programming (MP) model, referred to as the continuous global optimization (CGO) problem. In order to introduce this model form, we use the following notations.

R^n	n -dimensional Euclidean vector space
x	vector of continuous decision variables ($x \in R^n$)
xl, xu	explicit, component-wise lower- and upper-bound vectors on x
$f(x)$	objective function of the model
$g(x)$	vector function of equality constraints
$h(x)$	vector function of inequality constraints

Applying this notation, the CGO model can be expressed as

$$\begin{array}{ll} \min f(x) & f : D_0 \rightarrow R^1 \\ g(x) = 0 & g : D_0 \rightarrow R^{m_e} \\ b(x) \leq 0 & b : D_0 \rightarrow R^{m_i} \\ x \in D_0 = [xl, xu] & xl, xu \in R^n \end{array}$$

The *set of feasible solutions* to this model is defined by

$$D = \{g(x) = 0, b(x) \leq 0, xl \leq x \leq xu\}.$$

Note that D could be empty: for example, one may think of a system of nonlinear equations $g(x) = 0$ that has no solutions within the range specified by $xl \leq x \leq xu$. In many practical models, however, D is nonempty, and this property is explicitly postulated in most theoretical analyses of optimization models.

The *globally best solution* of the CGO model is an $x^* \in D$, such that $f(x^*) \leq f(x)$ for all $x \in D$.

The *locally best solution* of CGO is an $x^{*l} \in D$, such that $f(x^{*l}) \leq f(x)$ for all $x \in D$, $\|x - x^{*l}\| < \delta$. Here $\delta > 0$ is a suitable constant defining the δ -neighborhood of x^{*l} . (Here typically the Euclidean norm is used, but other norms could be used as well.)

Obviously, $f(x^*) \leq f(x^{*l})$, and the relation becomes equality only when x^{*l} is also a globally optimal solution.

According to the classical theorem of Weierstrass, a continuous function $f(x)$ attains its minimum on a closed, bounded set D . This fact is often used to guarantee the existence of the global solution in various special cases of the CGO model. Observe at the same time that, in its full generality, this model encompasses a large class of difficult instances.

Figure 1 illustrates the potential difficulty of continuous optimization models. Although the feasible set is just a two-dimensional interval, the objective function is highly multimodal. Hence, purely local scope search methods, as a general rule, even in much simpler cases, will not find the best solution. Note that this model is due to Trefethen, and it has been used also in our software tests: consult, for example, [38].

```

In[1]:= Plot3D[Exp[Sin[50*x]] + Sin[60*Exp[y]] + Sin[70*Sin[x]] +
             Sin[Sin[80*y]] - Sin[10*(x+y)] +  $\frac{x^2+y^2}{4}$ , {x, -3, 3},
             {y, -3, 3}, PlotPoints -> 50, ColorFunction -> Hue]

```

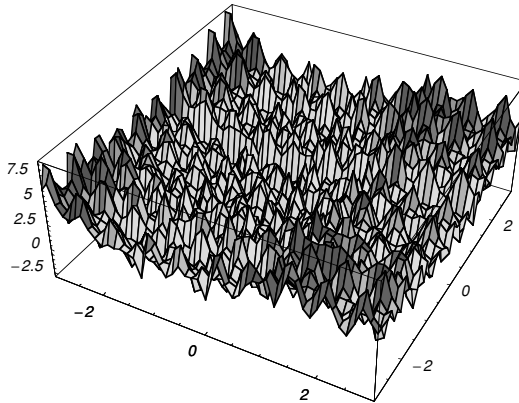


Figure 1. A box-constrained GO model in two dimensions.

■ Optimization in *Mathematica*

In this section, we briefly review `NMinimize`, a built-in general numerical optimization tool. This is followed by a somewhat more detailed summary of *MathOptimizer*, an application package.

Note that in addition to `NMinimize`, there are several other optimization-related functions in *Mathematica*, for example, `ConstrainedMin` and `ConstrainedMax`, which will also be discussed shortly. We entirely omit the discussion of the underlying theory and algorithms and refer the reader to *The Mathematica Book* [48], the current documentation [49, 50], and the optimization references listed earlier for details.

□ Built-in Optimization Tools

The following brief description of `NMinimize` is based on *Mathematica*'s online help system.

NMinimize

`NMinimize[{f, cons}, vars]` numerically finds (approximates) the global minimum of an arbitrary linear or nonlinear function *f* subject to the arbitrary constraints *cons*. `NMinimize` implements several algorithms for finding global optima. According to the documentation for `NMinimize`, “The methods are flexible enough to cope with functions that are not differentiable or continuous, and are

not easily trapped by local optima. Keep in mind, however, that finding a global optimum can be arbitrarily difficult, even without constraints, and so the methods used may fail. It may frequently be useful to optimize the function several times with different starting conditions and take the best of the results.” This caveat is of obvious relevance with respect to *all* general numerical (global) optimization approaches.

The following simple example illustrates the usage of this function. First, load the package if necessary (in *Mathematica* 5, `NMinimize` is autoloading).

```
In[2]:= If[$VersionNumber < 5,
        << NumericalMath`NMinimize`]

In[3]:= NMinimize[{Sin[5 x - 3 y] + x*y + (x - y)^2,
                  -2 ≤ x ≤ 5 && -3 ≤ y ≤ 4 && x + y ≤ 5}, {x, y}]

Out[3]= {-0.909737, {x → -0.268141, y → 0.0383059}}
```

Figure 2 shows the objective function over the interval defined by the explicit bounds. Note that the added $x + y \leq 5$ constraint cuts off a part of this box region.

```
In[4]:= Plot3D[Sin[5 x - 3 y] + x*y + (x - y)^2, {x, -2, 5}, {y, -3, 4},
             PlotPoints → 50, Mesh → False, ColorFunction → Hue]
```

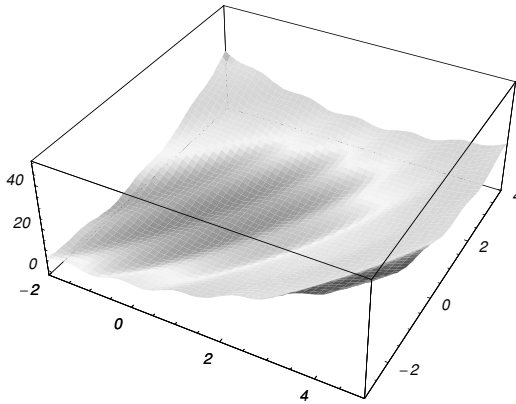


Figure 2. The function $\sin(5x - 3y) + x*y + (x - y)^2$ in the two-dimensional “box” region $x \in [-2, 5]$, $y \in [-3, 4]$.

□ *MathOptimizer*

MathOptimizer [38] is a third-party native *Mathematica* product that enables the global and local solution of the general class of optimization problems encompassed by the CGO model form. The key features of this package are summarized next. For theoretical background, refer to [35] and the other GO textbooks listed earlier.

MathOptimizer currently consists of two core solver packages and a solver integrator package. One of these solvers is used for the typically approximate global optimization of an aggregated “merit” (exact penalty) function on the given interval range $[xl, xu]$, in the possible presence of added constraints g and b . This package is based on an efficient adaptive stochastic search method, combined with a statistical bounding procedure. The latter provides an estimate of the global optimum value, based on the global sampling results.

The second solver package implements a convex optimization approach aimed at finding a solution vector that satisfies the Karush–Kuhn–Tucker local optimality conditions. This solver option can be used for “precise” local optimization, based on the initial solution produced by the global search phase or provided by the user. (Note that, at least theoretically, all CGO model functions should be smooth, in order to support the application of this solver component.)

The solver integrator package enables the individual or combined use of these two solver packages; the solvers can also be used separately. (Further solver modules are under development and will be made available.)

The *MathOptimizer* User Guide [38] is a *Mathematica* notebook that can be directly invoked through the online Help Browser. The user guide includes installation and technical notes and provides concise mathematical background information and modeling tips. In addition to a basic usage description, it also discusses a number of simple and more challenging test problems and a few detailed application examples.

MathOptimizer can be installed and used on all hardware platforms that are suitable to run *Mathematica* 3.0 or later. It is available for all Microsoft Windows (95 and later) operating systems, as well as all other platforms that have corresponding current *Mathematica* implementations. Several prominent OS examples are Linux, Macintosh (including Mac OS X), HP-UX, SGI, Solaris, and Sun (Unix flavor) platforms. *MathOptimizer* software installation and tests with identical file systems were successfully completed using over half a dozen different hardware platforms and operating systems (at the time of this writing, including all the aforementioned platforms).

MathOptimizer Usage Definitions and Options

Assuming that *MathOptimizer* is set up using the *Mathematica* standard directory structure, it can be invoked by the following statement.

```
In[5]:= Needs["MathOptimizer`Optimize`"]
```

Here is the concise information regarding how to use *Optimize*.

In[6]:= ?Optimize

Optimize[f, g, h, x, xinit, xlb, xub, opts] integrates the current solver options of the MathOptimizer system of packages. These packages together serve to find - that is, to numerically approximate - the solution of the following general constrained optimization problem: minimize $f(x)$ subject to $g(x)=0$ and $h(x)\leq 0$, x is a real vector from a given finite interval range [xlb, xub], xinit is a list of initial (nominal) values for x . All model functions are assumed to be continuous, g and h are lists of Mathematica expressions. MathOptimizer is based on a combination of global and local scope search procedures: these can be used via calling Optimize in a flexible manner, in suitable combinations or in a standalone mode, with corresponding sets of options. Optimize returns the solution vector, and additional model information (as described in detail with respect to the individual solver options). The symbolic argument opts denotes a list of usage possibilities: please use Options[Optimize] to display these. The current solver option packages are MS and CNLP: for further information on these, type ?MS or ?CNLP.

We will not go into further, quite extensive, details of the package options and refer the reader to the *MathOptimizer* User Guide [38] for more information. Instead, we shall illustrate the usage of the function Optimize on the last problem discussed in the preceding section.

Here are the main steps of the modeling and solution procedure. Comments set in blue serve only for explanation.

```
In[7]:= vars = {x, y}; (* decision variables *)
varnom = {0, 0}; (* nominal (initial) values of variables *)
varlb = {-2, -3}; (* lower bounds of variables *)
varub = {5, 4}; (* upper bounds of variables *)
objf = Sin[5 x - 3 y] + x * y + (x - y)^2;
(* objective function, to minimize *)
eqs = {}; (* equality constraints, absent *)
ineqs = {x + y - 5}; (* inequality constraints *)

In[8]:= Optimize[objf, eqs, ineqs, vars, varnom, varlb, varub,
GlobalSolverMode → 1, LocalSolverMode → 1, ReportLevel → 0]

Out[8]:= {{-0.268141, 0.0383059}, -0.909737,
{ }, {-5.22984}, {0., 2.03692 × 10-10, 0.}}
```

The output of *MathOptimizer* consists of the numerically estimated list of optimal solution components, the corresponding objective function value, the lists of the equality and inequality model function values at the solution (including an empty list of equality constraints in this example), the list of the maximal feasibility violation level at the solution, the violation level of the Karush–Kuhn–Tucker equation at the solution, and the violation level of the complementary slackness condition at the solution.

Without going into further details, let us note that the output indicates that the local optimality conditions are satisfied to a fairly high precision and that the solution agrees with the one found by `NMinimize` to at least 10^{-6} precision, in terms of both the optimized arguments and the optimum value.

In the next (Contact and Collision Problems) section, we shall illustrate the usage of *MathOptimizer* and `NMinimize` in the context of several nontrivial tests that point towards interesting application areas. An extensive collection of further numerical examples and illustrative applications formulated in *Mathematica* is discussed in [29, 39].

■ Contact and Collision Problems

□ Introductory Notes

There are many situations in engineering analysis and design in which one would like to know whether two stationary objects are in contact or, if not, what the minimum distance is between them. Similarly, for moving objects, one may wish to know whether or not they collide, the collision time if they do, or the minimum distance if they do not.

If each object can be analytically described as a set of points satisfying certain constraints, then finding the minimum distance can be formulated as a minimization problem in which one minimizes the distance between two points, each constrained to be in one of the objects. If one is only interested in whether or not the objects are in contact, the problem can be solved by determining if there are any points that satisfy the constraints describing both objects.

□ Intersecting Stationary Objects

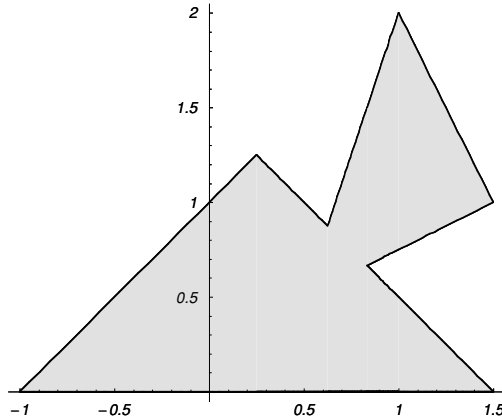
For a simple example of determining whether or not two stationary objects intersect, let us consider two triangles described by the following lists of constraints.

$$\begin{aligned} \text{tri1}[x_, y_] &= \{y \geq 0, \ x + y \leq \frac{3}{2}, \ y - x \leq 1\}; \\ \text{tri2}[x_, y_] &= \{y \geq \frac{x}{2} + \frac{1}{4}, \ y \leq -2x + 4, \ y \leq 3x - 1\}; \end{aligned}$$

The points satisfying one or the other list of constraints can be shown using the `InequalityPlot` function. It is easily seen that the two triangles do intersect.

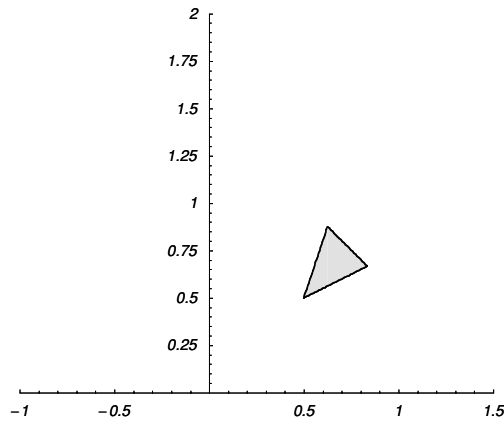
```
In[11]:= << Graphics`InequalityGraphics`
```

```
In[12]:= InequalityPlot[
  Apply[And, tri1[x, y]] || Apply[And, tri2[x, y]], {x}, {y}]
```



The shape of the intersection, when nonempty, can be easily produced by looking at points satisfying both lists of constraints.

```
In[13]:= InequalityPlot[ Apply[And, tri1[x, y]] && Apply[And, tri2[x, y]],
  {x}, {y}, PlotRange -> {{-1, 1.5}, {0, 2}}]
```



We can use the `Minimize` function to determine some target extremal point of the intersection of the triangles. In the following example, we want to determine the location of the point with the smallest $x + y$ value (see the preceding plot).

```
In[14]:= Minimize[x + y, Apply[And, tri1[x, y]] && Apply[And, tri2[x, y]], {x, y}]
```

```
Out[14]= {1, {x -> 1/2, y -> 1/2}}
```

Had the triangles not overlapped, there would have been no solution to the optimization model.

As another simple example, we can also determine the point in the intersection that has the maximal l_1 -norm (Manhattan) distance from the origin.

```
In[15]:= Maximize[x + y, Apply[And, tri1[x, y]] && Apply[And, tri2[x, y]], {x, y}]
```

```
Out[15]:= {3/2, {x -> 5/6, y -> 2/3}}
```

□ Nonintersecting Stationary Objects

To see an example of nonintersecting stationary objects, we modify triangle 1.

```
In[16]:= tri1[x_, y_] = {y ≥ 0, x + y ≤ 3/4, y - x ≤ 1};
```

In this case, `Minimize` gives no solution, as there are no points that satisfy the constraints of both triangles.

```
In[17]:= Minimize[x, Apply[And, tri1[x, y]] && Apply[And, tri2[x, y]], {x, y}]
```

```
Minimize::natt : The minimum is not attained  
at any point satisfying the given constraints. More ...
```

```
Out[17]:= {∞, {x -> Indeterminate, y -> Indeterminate}}
```

Next, we shall take a nonlinear optimization model form to analyze the current configuration of the triangles. `NMinimize` can be used to find the shortest line joining the two triangles by minimizing the distance between two points $\{x_1, y_1\}$ and $\{x_2, y_2\}$, with the first point constrained to be in triangle 1 and the second point constrained to be in triangle 2.

```
In[18]:= Timing[NMinimize[Join[{Sqrt[(x1 - x2)^2 + (y1 - y2)^2],  
tri1[x1, y1], tri2[x2, y2]}, {x1, y1, x2, y2}]]]
```

```
Out[18]:= {0.219 Second, {0.176777, {x1 -> 0.375, x2 -> 0.5, y1 -> 0.375, y2 -> 0.5}}}
```

MathOptimizer can be used as well to handle this optimization model. Note that in order to use *MathOptimizer*, it is necessary to rewrite the inequalities in the form $\text{ineq}(x, y) \leq 0$, and then give as input only a list of the left-hand side of the inequalities.

```
In[19]:= tri1[x_, y_] = {-y, x + y - 3/4, y - x - 1};  
tri2[x_, y_] = {x/2 - y + 1/4, 2x + y - 4, y - 3x + 1};
```

```
In[21]:= obj = Sqrt[(x1 - x2)^2 + (y1 - y2)^2];  
equalCons = {};  
inequalCons = Join[tri1[x1, y1], tri2[x2, y2]];  
vars = {x1, y1, x2, y2};  
inits = {1, 2, 2, 1};  
lbs = {-2, -2, -2, -2};  
ubs = {2, 2, 2, 2};
```

```
In[28]:= Timing[
  soln = Optimize[obj, equalCons, inequalCons, vars, inits, lbs, uba]]
```

```
FindMinimum::lstol :
```

```
The line search decreased the step size to within tolerance specified by
AccuracyGoal and PrecisionGoal but was unable to find a sufficient
decrease in the function. You may need more than MachinePrecision
digits of working precision to meet these tolerances. More...
```

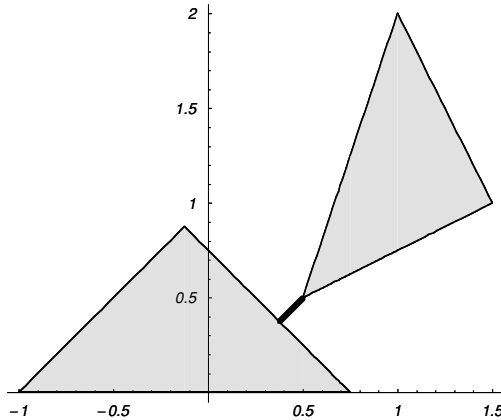
```
Out[28]:= {1.844 Second, {{0.375, 0.375, 0.5, 0.5}, 0.176777, {}},
  {-0.375, -2.62896 × 10-10, -1., 1.95283 × 10-10, -2.5, -3.92659 × 10-10},
  {1.95283 × 10-10, 2.88281 × 10-8, 1.31549 × 10-10}}
```

The shortest line can be shown superimposed on a plot of the triangles.

```
In[29]:= p1 = InequalityPlot[Apply[And, Thread[tri1[x, y] ≤ 0]] ||
  Apply[And, Thread[tri2[x, y] ≤ 0]],
  {x}, {y}, DisplayFunction → Identity];
```

```
In[30]:= p2 = Graphics[soln[[1]] /. {x1_, y1_, x2_, y2_} →
  {Thickness[.01], Line[{x1, y1}, {x2, y2}]}];
```

```
In[31]:= Show[p1, p2, DisplayFunction → $DisplayFunction]
```



□ Analysis of Moving Objects

In the following example, we shall analyze the configuration of two objects, one of them moving along a trajectory given analytically. Let us define

$$\text{In[32]:= } \text{sqr}[x_, y_] = \left\{y - \frac{1}{2}, -\frac{1}{2} - y, x - \frac{1}{2}, -\frac{1}{2} - x\right\};$$

Then $\text{Thread}[\text{sqr}[x, y] \leq 0]$ defines a square with edge 1 centered on the origin.

Next, we define the rotation $\text{rot}[\theta]$ that rotates two-dimensional points clockwise by angle θ .

```
In[33]:= rot[θ_][x_, y_] =
      {x := x*Cos[θ] + y*Sin[θ], y := -x*Sin[θ] + y*Cos[θ]};
```

The function `disp[dx,dy]` displaces a point by $\{-dx, -dy\}$.

```
In[34]:= disp[dx_, dy_][x_, y_] = {x := x - dx, y := y - dy};
```

Finally, `newsqr[dx,dy,θ]` takes variables x and y (specifically, denoting a point from the square), and rotates and displaces it back to the starting position so the original constraints can be applied.

```
In[35]:= newsqr[dx_, dy_, θ_][x_, y_] :=
      (sqr[x, y] /. rot[θ][x, y]) /. disp[dx, dy][x, y]
```

The other object in our illustrative example is a static disk centered on the origin with radius $1/2$.

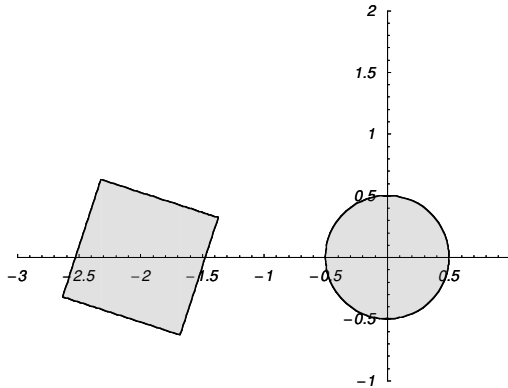
```
In[36]:= disk[x_, y_] = x^2 + y^2 - 1/4;
```

Now, as a starting position, we center the square at $\{-2, 0\}$, rotated clockwise by $\pi/10$ at $t = 0$, and then have it move with velocity $\{1.0, 0.5\}$ and rotate with angular velocity 0.3 . At any time moment t , the position of the disk and the square can be plotted using the following function.

```
In[37]:= p[t_] := InequalityPlot[
      And @@ Thread[newsqr[-2 + t, .5 * t, .3 t - π/10][x, y] ≤ 0] ||
      disk[x, y] ≤ 0, {x}, {y}, PlotRange → {{-3, 1}, {-1, 2}}];
```

Note that the function `p[t]` can be used to generate a series of plots, which can be animated. Here are the initial object positions.

```
In[38]:= p[0]
```



Now let us find the minimum distance between the square and the disk. First, we apply `NMinimize` and then `MathOptimizer`.

```

In[39]:= obj =  $\sqrt{(x1 - x2)^2 + (y1 - y2)^2}$  ;

equalCons = {};
inequalCons =
  Join[newsqr[-2 + t, .5 * t, .3 t -  $\frac{\pi}{10}$ ][x1, y1], {disk[x2, y2]}];

vars = {x1, y1, x2, y2, t};
inits = {0, 1, 1, 0, 1};
lbs = {-2, -2, -2, -2, 0};
ubs = {1, 1, 1, 1, 5};

In[46]:= Timing[solnNM = NMinimize[
  Join[{obj}, Thread[equalCons == 0], Thread[inequalCons ≤ 0]], vars]]
Out[46]:= {0.266 Second, {1.88758 × 10-16, {t → 1.17315,
  x1 → -0.312285, x2 → -0.312285, y1 → 0.106061, y2 → 0.106061}}}}

In[47]:= Timing @ (solnM0 =
  Optimize[obj, equalCons, inequalCons, vars, inits, lbs, ubs])

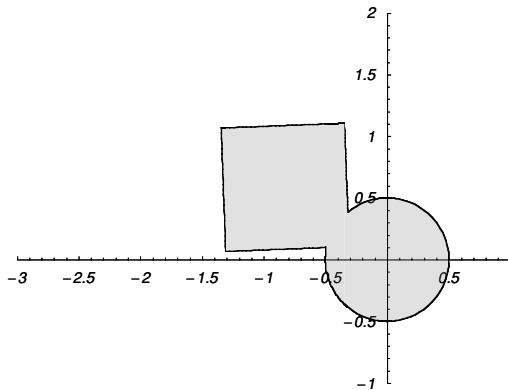
FindMinimum::lstol :
The line search decreased the step size to within tolerance specified by
AccuracyGoal and PrecisionGoal but was unable to find a sufficient
decrease in the function. You may need more than MachinePrecision
digits of working precision to meet these tolerances. More ...

Out[47]:= {3.812 Second,
  {{-0.340309, 0.35257, -0.340309, 0.35257, 1.67855}, 1.56419 × 10-15,
  {}}, {-0.97445, -0.02555, -0.610154, -0.389846, -0.00988422},
  {0., 1.41421, 0.}}}
```

The minimum distance is (numerically very close to) 0, so the objects must have collided. A graph of the solution confirms that the square and disk had collided, actually by the earlier of the two times found. (As indicated by the plot, at `tsoln` the objects already overlap.)

```

In[48]:= tsoln = t /. solnNM[[2]]
Out[48]:= 1.17315
In[49]:= p[tsoln]
```



Next, let us find the collision time by minimizing t with a point in the square constrained to be equal to a point in the disk. `NMinimize` and *MathOptimizer* give the same solution.

```
In[50]:= obj = t;
equalCons = {x1 - x2, y1 - y2};
inequalCons =
  Join[newsqr[-2 + t, .5 * t, .3 t -  $\frac{\pi}{10}$ ][x1, y1], {disk[x2, y2]}];
vars = {x1, y1, x2, y2, t};
inits = {0, 0, 0, 0, 1};
lbs = {-2, -2, -2, -2, 0};
ubs = {1, 1, 1, 1, 1.5};

In[57]:= Timing[NMinimize[
  Join[{obj}, Thread[equalCons == 0], Thread[inequalCons ≤ 0]], vars]]
```

`NMinimize::incst` :
 NMinimize was unable to generate any initial points satisfying
 the inequality constraints $\{-\frac{1}{4} + x^2 + y^2 \leq 0, \ll 3 \gg,$
 $-\frac{1}{2} - \ll 1 \gg + (-0.5 t + 1. y2) \text{Sin}[\ll 1 \gg] \leq 0\}$. The initial region specified
 may not contain any feasible points. Changing the initial region or
 specifying explicit initial points may provide a better solution. More ...

```
Out[57]= {0.172Second, {1.00612, {t → 1.00612, x1 → -0.499962,
  x2 → -0.499962, y1 → 0.00616099, y2 → 0.00616099}}}
```

```
In[58]:= Timing @
  (soln = Optimize[obj, equalCons, inequalCons, vars, inits, lbs, ubs])
```

`FindMinimum::lstol` :
 The line search decreased the step size to within tolerance specified by
 AccuracyGoal and PrecisionGoal but was unable to find a sufficient
 decrease in the function. You may need more than MachinePrecision
 digits of working precision to meet these tolerances. More ...

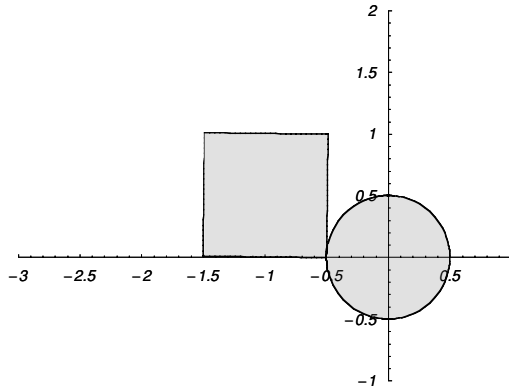
```
Out[58]= {4.89Second, {{-0.499962, 0.00616099, -0.499962, 0.00616099, 1.00612},
  1.00612, {9.34605 × 10-10, -2.03102 × 10-9},
  {-0.990777, -0.0092231, -6.49468 × 10-11, -1., 1.87505 × 10-8},
  {1.87505 × 10-8, 5.75711 × 10-8, 1.63765 × 10-8}}}
```

```
In[59]:= tsoln = soln[[1, 5]]
```

```
Out[59]= 1.00612
```

The next plot visually confirms that the solution found is a good numerical approximation of the collision time.

`In[60]:= p[tsoln]`



Obviously, these simple illustrative examples could be extended and modified to model issues of practical interest, for example, robot or machine movements in manufacturing and ergonomic contexts. Further related examples are discussed in [29].

■ Non-Uniform Size Circle Packings

□ Introductory Notes

Point arrangements and closely related *uniform* circle packings within given bodies (the latter most typically being n -dimensional intervals and circles for $n = 2, 3$) have been intensively studied by the mathematical community for several decades. Such models are not only of pure academic interest, but are also closely related to important problems in numerical mathematics (integration, experimental design), physics and chemistry (potential models, crystal and molecule structures), and biology (viral morphology), just to name a few application areas.

There is a considerable body of literature devoted to these subjects [51–59]. Several of these works provide extensive lists of further references.

The applicability of global optimization tools to analyze and solve instances of various point arrangement (potential) models has been demonstrated [60–62]. We emphasize here that the issue of *non-uniform size* circle (or more general object) packings is entirely outside of the scope of “traditional” (purely analytical) studies, while GO can still be applied to such, as well as to many other, model variants and extensions.

For illustration, in this section we use global optimization to pack different-size circles into the smallest possible circle. Since this model formulation typically has infinitely many solutions per se, we will additionally try to bring the circles as close together as possible. The primary objective is to find the embedding circle with the smallest radius; the secondary objective—bringing the circles close

together—is added as a second criterion function. A linear combination of the two objectives is used in an aggregate objective function.

Note that alternative formulations are also possible and that rotational symmetries of solutions can also be avoided by added constraints, thereby making the solution of a specific model formulation essentially unique.

□ Model Formulation and Numerical Examples Using NMinimize, MathOptimizer, and MathOptimizer Professional

The solution is demonstrated using three different optimizers: *NMinimize*, *MathOptimizer*, and *MathOptimizer Professional*. *MathOptimizer Professional* is a recently introduced product [39] in which the actual optimization is performed by an external executable—the Lipschitz Global Optimizer (LGO) solver engine—on a Windows dynamic link library (dll) model function. The latter is automatically generated from a model formulated in *Mathematica*.

Without going into details, which are outside of the scope of this article, we should mention that LGO is a global/local solver suite that has been discussed, for example, in [35–37]. The LGO engine can also be used to handle models set up in Excel [63] and GAMS [64], in addition to its C or Fortran compiler-based implementations. Let us note that [65] presents a detailed and fully reproducible comparative assessment of LGO versus several state-of-art local nonlinear solvers: this comparison is based on solving an extensive set of GAMS models, and it was done by the GAMS Development Corporation (thus it can be considered as reasonably objective). These tests demonstrate that in a significant percentage of nonlinear optimization (GAMS) models, global scope search is necessary indeed; furthermore, that LGO compares favorably to the solvers considered (CONOPT, MINOS, and SNOPT).

In the following model equations, the center of circle i is denoted by $\{x[i], y[i]\}$ and the radius of circle i is $r[i]$. The distance from the origin to the most distant point in a circle is given by the radius of the circle plus the distance of the center of the circle from the origin. For circle i , this distance is given by $dd[i]$, as formulated here.

$$\text{In}[61]:= \text{dd}[i_Integer] = r[i] + \sqrt{x[i]^2 + y[i]^2};$$

The distance between the centers of the circles i and j is given by $p[i, j]$. Two circles do not overlap, if the sum of their radii minus the distance between their centers is not greater than 0: the corresponding term is given by $dd[i, j]$ for circles i and j .

$$\begin{aligned} \text{In}[62]:= \text{p}[i_Integer, j_Integer] &= \sqrt{(x[i] - x[j])^2 + (y[i] - y[j])^2}; \\ \text{dd}[i_Integer, j_Integer] &= r[i] + r[j] - \text{p}[i, j]; \end{aligned}$$

The optimization variables for describing the position of n circles are the coordinates of the centers of the circles.

$$\text{In}[64]:= \text{Clear}[\text{vars}, \text{inequalCons}];$$

```
In[65]:= vars[n_] := Flatten[Table[{x[i], y[i]}, {i, n}]];
```

The first objective, defined by obj1, is to minimize the radius of a circle circumscribing the packed circles, which is the maximum of the values $dd[i]$ for the n circles. *MathOptimizer Professional* only evaluates Max for two arguments, requiring the use of a table of constraints to define the first objective.

```
In[66]:= obj1[n_] := Max[Table[dd[i], {i, n}]];
obj1Cons[rmax_, n_] := Table[rmax ≥ dd[i], {i, n}];
```

The second objective, defined by obj2, is to minimize the average distance between the centers of the n circles.

```
In[68]:= obj2[n_] :=  $\frac{2}{n(n-1)}$  *
Apply[Plus, Flatten[Table[p[i, j], {i, n-1}, {j, i+1, n}]]];
```

The overall objective function is defined now as the weighted average of the two objective functions obj1 and obj2, with $0 \leq \lambda \leq 1$ being the weight parameter. Note that the second objective function component is normalized to be the average distance between the centers of the circles, so that the two objective functions will remain of comparable magnitude. For $\lambda = 0$, the objective function is aimed at finding the smallest embedding circle. For $\lambda = 1$, the objective function expresses solely the average distance between the circle centers, providing the “tightest possible” packing.

```
In[69]:= obj[λ_, n_] := (1 - λ) obj1[n] + λ * obj2[n];
objMOP[λ_, n_, rmax_] := (1 - λ) * rmax + λ * obj2[n];
```

The requirement that the packed circles not overlap means that $dd[i, j] \leq 0$ for all pairs of i and j . The input form differs somewhat for the different optimizers. For *MathOptimizer Professional*, it is necessary to add the constraints that define rmax.

```
In[71]:= inequalCons[n_] := Flatten[Table[dd[i, j], {i, n-1}, {j, i+1, n}]];
inequalConsNM[n_] := Thread[inequalCons[n] ≤ 0];
inequalConsMOP[n_, rmax_] :=
Join[Thread[inequalCons[n] ≤ 0], obj1Cons[rmax, n]];
```

The problem as formulated has considerable symmetry. To reduce the symmetry, several constraints are added (to set the first circle object position and the relative location of the second and third circle objects), so that the results of the different optimizers can be more easily compared. Note that adding further similar constraints could increase the overall model complexity significantly, and thus it is avoided.

```
In[74]:= symbreakEqualConsNM := {x[1] == 0, y[1] + obj1[n] - r[1] == 0};
symbreakEqualConsMO := symbreakEqualConsNM[All, 1];
symbreakEqualConsMOP := {x[1] == 0, y[1] + rmax - r[1] == 0};
symbreakInequalCons :=
{x[1] - x[2] ≤ 0, y[1] - y[2] ≤ 0, x[3] - x[1] ≤ 0, y[1] - y[3] ≤ 0};
symbreakInequalConsMO := symbreakInequalCons[All, 1]
```

MathOptimizer and *MathOptimizer Professional* require bounds on the variables, which are expressed somewhat differently. Since the formulation for *MathOptimizer Professional* requires an extra variable, the variables with bounds are constructed in one function.

```
In[79]:= boundsM0[n_] := Apply[Sequence, Transpose[Table[{0, -3, 3}, {2*n}]]];
varswithboundsMOP[n_] := Join[{{rmax, 0, 3}},
  Transpose[{vars[n], Table[-3, {2*n}], Table[3, {2*n}]}]]];
```

The function `in[i]` produces a constraint describing circle *i* for use in the `InequalityPlot` function.

```
In[81]:= in[i_Integer] = (x - x[i])^2 + (y - y[i])^2 ≤ r[i]^2;
```

The results of the calculations are shown as the combination of three plots: `p1` uses `InequalityPlot` to show the circles with a table of inequalities defining the circles; `p2` is a plot of the circumscribing circle; `p3` is a plot of the centers of the circles. `show` combines the plots.

```
In[82]:= Clear[p1, p2, p3, show];
In[83]:= p1[solnrule_, n_] :=
  InequalityPlot[Apply[Or, Table[in[i], {i, n}]] /. solnrule,
    {x}, {y}, DisplayFunction → Identity];
p2[solnrule_, n_] := Graphics [Circle[{0, 0}, obj1[n] /. solnrule]];
p3[solnrule_, n_] := Graphics [
  Table[{PointSize[.025], Point[{x[i], y[i]}]}, {i, n}] /. solnrule];
show[solnrule_, n_] := Block[{rr = obj1[n] /. solnrule},
  Show[p1[solnrule, n], p2[solnrule, n],
    p3[solnrule, n], DisplayFunction → $DisplayFunction,
    PlotRange → {{-rr, rr}, {-rr, rr}}]]];
```

Having defined the variables, objective function, and constraints, we now define functions for performing the circle packing and displaying the results using *MathOptimizer*, *MathOptimizer Professional*, and `NMinimize`. The numerical outputs of the following three functions are the values of the two objective functions and the complete solution.

```
In[87]:= packMathOptimizer[n_Integer, λ_] :=
  Block[{soln, solnrule},
    soln = Optimize[obj[λ, n], symbreakEqualConsM0, Join[
      symbreakInequalConsM0, inequalCons[n], vars[n], boundsM0[n]];
    solnrule = Thread[vars[n] → soln[[1]]];
    show[solnrule, n];
    {{obj1[n], obj2[n]} /. solnrule, soln}]
In[88]:= << MathOptimizerPro`callLGO`
```

```

In[89]:= packMathOptimizerPro[n_Integer, λ_, opts___] :=
  Block[{soln},
    soln = callLGO[objMOP[λ, n, rmax],
      Join[symbreakEqualConsMOP, symbreakInequalCons,
        inequalConsMOP[n, rmax]], varswithboundsMOP[n], opts];
    show[soln[[2]], n];
    {{obj1[n], obj2[n]} /. soln[[2]], soln}

In[90]:= packNMinimize[n_Integer, λ_] :=
  Block[{soln},
    soln = NMinimize[Join[{obj[λ, n]}, inequalConsNM[n],
      symbreakInequalCons, symbreakEqualConsNM], vars[n],
      Method → {"DifferentialEvolution", "SearchPoints" →
        50 * (Length[vars[n]] + Length[inequalCons[n]])},
      Compiled → False];
    show[soln[[2]], n];
    {{obj1[n], obj2[n]} /. soln[[2]], soln}

```

In our illustrative examples included here, five circles generated with radii 0.2 to 1.0 are used, together with a λ value of 0.5, as the key problem input.

```

In[91]:= n = 5; Do[r[i] =  $\frac{n-i+1}{n}$ , {i, n}]

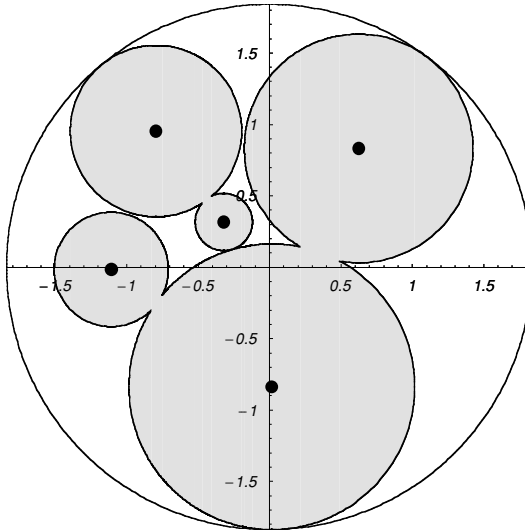
```

In[92]:= **Timing @ packMathOptimizer[n, 0.5]**

FindMinimum::lstol :

The line search decreased the step size to within tolerance specified by AccuracyGoal and PrecisionGoal but was unable to find a sufficient decrease in the function. You may need more than MachinePrecision digits of working precision to meet these tolerances. More ...

General::stop : Further output of FindMinimum::lstol will be suppressed during this calculation. More ...

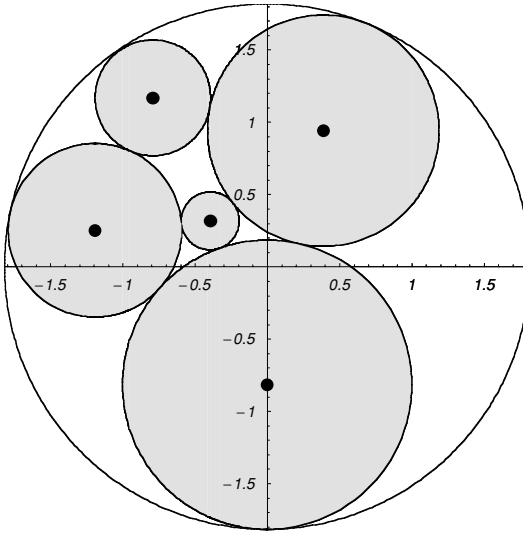


Out[92]= {79.609 Second,
 {{1.83936, 1.34321}, {{0.0180293, -0.839165, 0.625977, 0.829711,
 -0.792936, 0.951527, -1.10627, -0.0172762, -0.318036, 0.315019},
 1.59128, {0.0180293, 0.000193655}, {-0.607948, -1.66888,
 -0.810966, -1.79069, 0.0238389, -0.365768, 0.0073256,
 -0.00211565, -0.0241332, -0.728226, -0.0752063, -0.0182119,
 0.00585091, -0.25541}, {0.0238389, 0.707359, 0.00162173}}}}

□ Numerical Example Using *MathOptimizer Professional*

AbsoluteTiming is used in this example since Timing will not include the time required by the compiler and the LGO executable.

```
In[93]:= AbsoluteTiming @ packMathOptimizerPro[n, 0.5]
```



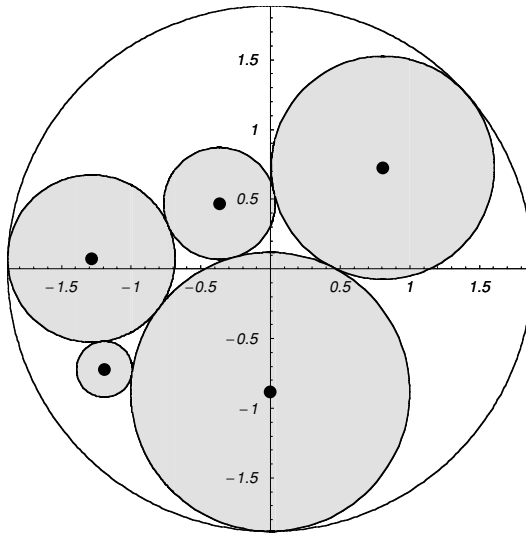
```
Out[93]= {2.7031250 Second,
  {{1.81725, 1.33983}, {1.57854, {rmax → 1.81725, x[1] → 0.,
    y[1] → -0.817247, x[2] → 0.387515, y[2] → 0.940545,
    x[3] → -1.19107, y[3] → 0.251097, x[4] → -0.790826, y[4] → 1.16751,
    x[5] → -0.393731, y[5] → 0.316321}, 1.4684 × 10-11}}}
```

Comparison of the *MathOptimizer* and *MathOptimizer Professional* results shows that they are very similar. The *MathOptimizer Professional* result has somewhat lower values for both objective functions, and it also leads to smaller constraint violations: the maximal overlap of the circles equals approximately 1×10^{-11} . The calculation time for *MathOptimizer Professional* is approximately 30 times smaller, due its use of a compiled executable for minimization and a compiled dll for objective function and constraint evaluation. In further numerical tests, we have used *MathOptimizer Professional* to solve similar models up to $n = 40$ circles, without any other prior circle object arrangement specifications: these results are reported elsewhere [66].

□ Numerical Example Using NMinimize

Next, we also completed a few experimental runs to see how `NMinimize` performs on the same circle packing test problems. The circle packing obtained was not as good as in the result produced by *MathOptimizer* (or by *MathOptimizer Professional*).

```
In[94]:= Timing[packNMinimize[n, 0.5]]
```



```
Out[94]= {124.985 Second, {{1.88473, 1.51038}, {1.69756,
{x[1] → 0, x[2] → 0.808008, x[3] → -1.28273, x[4] → -0.364223,
x[5] → -1.18903, y[1] → -0.884734, y[2] → 0.72372,
y[3] → 0.0716622, y[4] → 0.467059, y[5] → -0.722831}}}}
```

Note that we also did further extensive tests with respect to all three solvers. The corresponding—far more detailed—numerical results will be reported and discussed elsewhere, for example, in our forthcoming book [29].

■ Conclusion

In this article, the subject of optimization using *Mathematica* tools is discussed, with an emphasis on models that could have multiple local optima of various quality: in such cases, we wish to find the best possible (global) solution. After introducing a general optimization modeling framework, several built-in *Mathematica* optimization functions and the *MathOptimizer* package are reviewed. To illustrate the usage of *MathOptimizer*, several point and multibody configuration analysis and design models are formulated and solved. In an illustrative test related to “best” circle packings, we also used *NMinimize* and *MathOptimizer Professional*.

As our results demonstrate, the usage of proper GO tools is both necessary and possible in order to provide globally established numerical solutions to difficult models of theoretical interest and practical relevance. To illustrate the practical use of our optimizers, let us mention here that *MathOptimizer* has been recently used to design high-quality acoustic equipment: consult, for example, [67]. The LGO solver system embedded in *MathOptimizer Professional* has already been used in a variety of advanced applications, including, for example, cancer therapy

planning, laser equipment design, robotics design, industrial (shape) design, chemical product (material composition) design, water quality modeling, wastewater systems design and operations, computational chemistry, econometrics, financial modeling, and others. Most of these results are reported elsewhere: consult, for example, [29, 35–39]. We expect that a significant range of advanced optimization models developed using *Mathematica* will be successfully analyzed and solved using our packages.

■ Acknowledgments

The development of *MathOptimizer* (by the second author) has profited significantly from the quality software, literature, and advice provided by Wolfram Research staff in recent years, and from a project funded in part by Defence R&D Canada under Contract W7707-01-0746/001/HAL.

János Pintér also wishes to thank Christopher J. Purcell (DRDC Atlantic Region, Dartmouth, NS) for many useful comments related to the development of *MathOptimizer*. Pintér's research has also been partially supported by the National Research Council of Canada (NRC IRAP Project 362093) and by the Hungarian Scientific Research Fund (OTKA Grant T 034350).

We also wish to thank Mark Sofroniou at Wolfram Research, Inc. for his kind permission to use and customize the *Format.m Mathematica* package in our *MathOptimizer Professional* development work.

■ References

- [1] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty, *Nonlinear Programming: Theory and Algorithms*, New York: Wiley, 1993.
- [2] M. A. Bhatti, *Practical Optimization Methods with Mathematica Applications*, New York: Springer-Verlag, 2000.
- [3] D. P. Bertsekas, *Nonlinear Programming*, 2nd ed., Cambridge, MA: Athena Scientific, 1999.
- [4] D. Bertsimas and R. M. Freund, *Data, Models, and Decisions: The Fundamentals of Management Science*, Cincinnati: South-Western College Publishing, 2000.
- [5] J.R. Birge and K.G. Murty, eds., *Mathematical Programming: State of the Art 1994*, Ann Arbor: The University of Michigan, 1994.
- [6] M. W. Carter and C. C. Price, *Operations Research: A Practical Introduction*, Boca Raton, FL: CRC Press, 2001.
- [7] E. K. P. Chong and S. H. Zak, *An Introduction to Optimization*, 2nd ed., New York: Wiley, 2001.
- [8] G. B. Dantzig and M. N. Thapa, *Linear Programming: Introduction*, Vol. 1, New York: Springer-Verlag, 1997.
- [9] D. Hertog, *Interior Point Approach to Linear, Quadratic and Convex Programming*, Dordrecht: Kluwer Academic Publishers, 1994.

- [10] J. E. Dennis and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, NJ: Prentice-Hall, 1983.
- [11] A. V. Fiacco and G. P. McCormick, *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*, New York: Wiley, 1968.
- [12] R. Fletcher, *Practical Methods of Optimization*, 2nd ed., Chichester, UK: Wiley, 1987.
- [13] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*, London: Academic Press, 1981.
- [14] F. S. Hillier and G. J. Lieberman, *Introduction to Operations Research*, 4th ed., Oakland, CA: Holden Day, 1986.
- [15] T. M. Liebling and D. de Werra, eds., *Lectures on Mathematical Programming (ISMP97)*, Amsterdam: Elsevier Science, 1997.
- [16] D. G. Luenberger, *Linear and Nonlinear Programming*, 2nd ed., Reading, MA: Addison-Wesley, 1984.
- [17] K. G. Murty, *Linear and Combinatorial Programming*, New York: Wiley, 1976.
- [18] S. G. Nash and A. Sofer, *Linear and Nonlinear Programming*, New York: McGraw-Hill, 1996.
- [19] J. Nocedal and S. J. Wright, *Numerical Optimization*, New York: Springer-Verlag, 1999.
- [20] T. Terlaky, ed., *Interior Point Methods of Mathematical Programming*, Dordrecht: Kluwer Academic Publishers, 1996.
- [21] H. P. Williams, *Model Building in Mathematical Programming*, 4th ed., New York: Wiley, 1999.
- [22] R. Aris, *Mathematical Modeling: A Chemical Engineer's Perspective*, San Diego, CA: Academic Press, 1999.
- [23] J. L. Casti, *Searching for Certainty*, New York: William Morrow & Co., 1990.
- [24] T. F. Edgar, D. M. Himmelblau, and L. S. Lasdon, *Optimization of Chemical Processes*, 2nd ed., New York: McGraw-Hill, 2001.
- [25] M. Eigen and R. Winkler, *Das Spiel*, München: Piper & Co., 1975.
- [26] N. A. Gershenfeld, *The Nature of Mathematical Modeling*, Cambridge, UK: Cambridge University Press, 1999.
- [27] I. E. Grossmann, ed., *Global Optimization in Engineering Design*, Dordrecht: Kluwer Academic Publishers, 1996.
- [28] C. Jacob, *Illustrating Evolutionary Computation with Mathematica*, San Francisco: Morgan Kaufmann Publishers, 2001.
- [29] F. J. Kampas and J. D. Pintér, *Advanced Optimization: Scientific, Engineering, and Economic Applications with Mathematica Examples*, Amsterdam: Elsevier Science, forthcoming.
- [30] B. B. Mandelbrot, *The Fractal Geometry of Nature*, New York: Freeman & Co., 1983.
- [31] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd ed., Berlin: Springer-Verlag, 1996.
- [32] J. D. Murray, *Mathematical Biology*, New York: Springer-Verlag, 1989.

- [33] P. Y. Papalambros and D. J. Wilde, *Principles of Optimal Design*, 2nd ed., Cambridge, UK: Cambridge University Press, 2000.
- [34] P. M. Pardalos and M. G. C. Resende, eds., *Handbook of Applied Optimization*, Oxford: Oxford University Press, 2002.
- [35] J. D. Pintér, *Global Optimization in Action: Continuous and Lipschitz Optimization: Algorithms, Implementations and Applications*, Dordrecht: Kluwer Academic Publishers, 1996.
- [36] J. D. Pintér, *Computational Global Optimization in Nonlinear Systems: An Interactive Tutorial*, Atlanta: Lionheart Publishing, 2001.
- [37] J. D. Pintér, "Global Optimization: Software, Test Problems, and Applications." *Handbook of Global Optimization* (P. M. Pardalos and H. E. Romeijn, eds.), Vol. 2, Dordrecht: Kluwer Academic Publishers, 2002 pp. 519–569.
- [38] J. D. Pintér, *MathOptimizer: An Advanced Modeling and Optimization System for Mathematica Users: User Guide*, Halifax, NS: Pintér Consulting Services, Inc., 2002.
- [39] J. D. Pintér and F. J. Kampas, *MathOptimizer Professional: An Advanced Modeling and Optimization System for Mathematica, Using the LGO Solver Engine: User Guide*, Halifax, NS: Pintér Consulting Services, Inc., 2003.
- [40] F. Rothlauf, *Representations for Genetic and Evolutionary Algorithms*, Heidelberg: Physica-Verlag, 2002.
- [41] M. Schroeder, *Fractals, Chaos, Power Laws*, New York: Freeman & Co., 1991.
- [42] S. P. Sethi and G. L. Thompson, *Optimal Control Theory: Applications to Management Science and Economics*, 2nd ed., Dordrecht: Kluwer Academic Publishers, 2000.
- [43] M. Tawarmalani and N. V. Sahinidis, *Convexification and Global Optimization in Continuous and Mixed-Integer Nonlinear Programming*, Dordrecht: Kluwer Academic Publishers, 2002.
- [44] S. Wolfram, *A New Kind of Science*, Champaign: Wolfram Media, Inc., 2002.
- [45] L. C. W. Dixon and G. P. Szegö, eds., *Towards Global Optimization*, Vols. 1–2, Amsterdam: North-Holland, 1975, 1978.
- [46] R. Horst and P. M. Pardalos, eds., *Handbook of Global Optimization*, Vol. 1, Dordrecht: Kluwer Academic Publishers, 1995.
- [47] P. M. Pardalos and H. E. Romeijn, eds., *Handbook of Global Optimization*, Vol. 2, Dordrecht: Kluwer Academic Publishers, 2002.
- [48] S. Wolfram, *The Mathematica Book*, 4th ed., Champaign: Wolfram Media, Inc., 1999.
- [49] *Mathematica*, Release Version 4.2, Champaign: Wolfram Research, Inc., 2002.
- [50] *Mathematica*, Release Version 5.0, Champaign: Wolfram Research, Inc., 2003.
- [51] J. H. Conway and N. J. A. Sloane, *Sphere Packings, Lattices and Groups*, 2nd ed., Berlin: Springer-Verlag, 1988.
- [52] B. Hartke, "Global Cluster Geometry Optimization by a Phenotype Algorithm with Niches: Location of Elusive Minima, and Low-order Scaling with Cluster Size," *Journal of Computational Chemistry*, **20**, 1999 pp. 1752–1759.
- [53] H. Melissen, "Packing and Covering with Circles," Ph.D. diss., The Netherlands: University of Utrecht, 1997.
- [54] A. Neumaier, "Molecular Modeling of Proteins and Mathematical Prediction of Protein Structure," *SIAM Review*, **39**, 1997 pp. 407–460.

- [55] R. H. Pain, ed., *Mechanisms of Protein Folding*, Oxford, UK: Oxford University Press, 1994.
- [56] P. M. Pardalos, D. Shalloway, and G. Xue, *Global Minimization of Nonconvex Energy Functions: Molecular Conformation and Protein Folding*, DIMACS Series, Vol. 23, Providence: American Mathematical Society, 1996.
- [57] E. B. Saff and A. B. J. Kuijlaars, "Distributing Many Points on a Sphere," *The Mathematical Intelligencer*, **19**, 1997 pp. 5–11.
- [58] S. Smale, "Mathematical Problems for the Next Century," *The Mathematical Intelligencer*, **20**, 1998 pp. 7–15.
- [59] D. J. Wales and H. A. Scheraga, "Global Optimization of Clusters, Crystals, and Biomolecules," *Science*, **285**, 1999 pp. 1368–1372.
- [60] J. D. Pintér, "Extremal Energy Models and Global Optimization," *Computing Tools for Modeling, Optimization and Simulation* (M. Laguna and J. L. González-Velarde, eds.), Dordrecht: Kluwer Academic Publishers, 2000 pp. 145–160.
- [61] J. D. Pintér, "Globally Optimized Spherical Point Arrangements: Model Variants and Illustrative Results," *Annals of Operations Research*, **104**, 2001 pp. 201–213.
- [62] W. J. H. Stortelder, J. J. B. de Swart, and J. D. Pintér, "Finding Elliptic Fekete Point Sets: Two Numerical Solution Approaches," *Journal of Computational and Applied Mathematics*, **130**, 2001 pp. 205–216.
- [63] Frontline Systems and Pintér Consulting Services, Inc., *Premium Solver Platform Solver Engines User Guide: LGO Global Solver*, Incline Village, NV: Frontline Systems, 2001 www.solver.com/xlslgoeng.htm.
- [64] GAMS Development Corporation and Pintér Consulting Services, Inc., *GAMS/LGO Solver Engine*, Washington, DC: GAMS Development Corporation, 2003 www.gams.com/dd/docs/solvers/lgo.pdf.
- [65] J. D. Pintér, "GAMS/LGO Nonlinear Solver Suite: Key Features, Usage, and Numerical Performance," (Submitted for publication.) www.gams.com/solvers/solvers.htm#LGO.
- [66] J. D. Pintér and F. J. Kampas, "Nonlinear Optimization in Mathematica with MathOptimizer Professional," *Mathematica in Education and Research*, **10**(2), 2005 pp. 1–18.
- [67] J. D. Pintér and C. J. Purcell, "Optimization of Finite Element Models with MathOptimizer and ModelMaker," Presented at the Mathematica Developers Conference (Champaign, IL, April 10–12, 2003).

About the Authors

Frank J. Kampas is a senior developer at WAM Systems, Inc., where he is responsible for adding optimization capabilities to the company's supply-chain management software. He holds a Ph.D. in physics from Stanford University and an M.B.A. from Temple University.

János D. Pintér is a consultant (owner of Pintér Consulting Services, Inc.); he is also an adjunct professor at Dalhousie University. He has authored books, articles, and professional software in the area of advanced nonlinear optimization. He holds a Ph.D. (Moscow State University) and a D.Sc. (Hungarian Academy of Sciences), both in mathematics. For further information, visit myweb.dal.ca/jdpinter.

Frank J. Kampas

WAM Systems, Inc.
600 West Germantown Pike, Suite 230
Plymouth Meeting, Pennsylvania 19462
fkampas@wamsystems.com

János D. Pintér

Pintér Consulting Services, Inc.
129 Glenforest Drive
Halifax, Nova Scotia
Canada, B3M 1J2
jdpinter@hfx.eastlink.ca
www.pinterconsulting.com

The Xmath Calculator

Odd Bringslid

Xmath [1] and dMath [2] are projects supported by the European Commission through the Minerva Action (Xmath) [3] and the Leonardo da Vinci program (dMath) [4]. These programs seek to promote European cooperation in the fields of open and distance learning (Xmath) and vocational training (dMath). A pedagogic calculator [5] is being developed using web*Mathematica* [6, 7] in connection with these programs. This calculator gives intermediate steps for calculations in a wide range of applications (integration, differentiation, algebra, equation solving, and so on). An expression is broken down and analyzed by *Mathematica* packages that have been developed to work with web*Mathematica*. Rules that are familiar from hand calculation are given and so are the intermediate results. The user may scroll to look at a number of levels (the first level is the first hint) and then continue by hand. In the same way as an expression may be broken down into different levels, so can the calculation steps. This gives structured output, as a professor would do it on a blackboard, stating the rules at each level. The output in web*Mathematica* may be written as MathMLForm, giving a non-image format that can be rendered in Internet Explorer using proper style sheets. One of the main ideas of Xmath is implementing MathML [8] in general web pages containing mathematical expressions. The dMath project creates a database of mathematical modules using the SciLas system developed in the project. The Xmath calculator is connected to the database modules and will be further developed in dMath.

■ **The Xmath Project**

The Xmath project is part of the Minerva Action. The Minerva Action's purpose is to increase European cooperation in information and communication technology (ICT) and open and distance learning (ODL) in education. The action seeks to promote understanding of the implications of ICT and the critical and responsible use of ICT for educational purposes among teachers, learners, decision makers, and the public at large.

At European universities and colleges the numbers of students in mathematical courses and passing the exams are dangerously low. To meet this challenge, mathematics teaching needs to undergo an innovative process, making new technology important. This will create more efficient, direct teaching and allow an explorative and personal way of working [9, 10].

Xmath has built a framework making it possible both to use MathML in mathematical education on the internet and to evaluate the pedagogical advantages of

net-based education in mathematics. Development and research into MathML's use in e-learning applications has been one of the main purposes of Xmath. Using MathML in Internet Explorer requires special style sheets and plug-ins (e.g., MathPlayer). MathML is a standard supported by the World Wide Web Consortium (W3C) [8]. They have supported the Xmath project as well. *webMathematica* [6] and *Mathematica* [11] are important parts of this framework.

A pilot course [1] has been developed to test and demonstrate the main ideas of Xmath. The Xmath project has been completed and reports on the pedagogical aspects of ODL in Xmath [12] and the Xmath project [13] have been written. The Xmath project arranged The First European Workshop on MathML & Scientific e-Contents [14]. The second workshop was arranged by the follow-up project, dMath [15].

■ The dMath Project

The dMath project is a follow-up to Xmath and is intended to be an industrialization of the Xmath project. This project will develop a European database of mathematical e-learning modules. These modules are independent but may be combined in an arbitrary way to set up a course in web-based mathematics.

The system contains three main parts:

1. The authoring suite (working title SciLas) has two pieces: an environment to write and store mathematical content trees, which are a collection of reusable learning objects organized in modules, and an environment to publish and download these learning objects with the capability to reorganize them into a course. The content trees are accessible through a web browser using predefined style sheets. Storage requires preliminary authorization and approval (dMath development server).
2. The dMathArchive, based on this system, is a collection of modules (see Figure 1).
3. The Xmath calculator, based on *webMathematica*, is to be used by developers, publishers, and course students to do calculations on the web. The calculator gives rules and steps in a calculation as a professor would do it on the blackboard.

The module design is important for adoption of local conditions and to enhance flexibility. Publishers may download modules from the archive in an arbitrary way and link them into a course in web-based mathematics. This collection of modules may be used as a mathematical dictionary as well.

In the dMath project we will focus on the needs for upgrading and furthering the education of teachers and engineers in mathematics. This is in accordance with national and European strategies for these target groups especially and for life-long learning in general. Because dMath is strongly innovative and uses new technologies (MathML, *webMathematica*, and innovative database design) other

potential users may be publishers, people in computer science, content providers of scientific e-learning material, and researchers and developers.

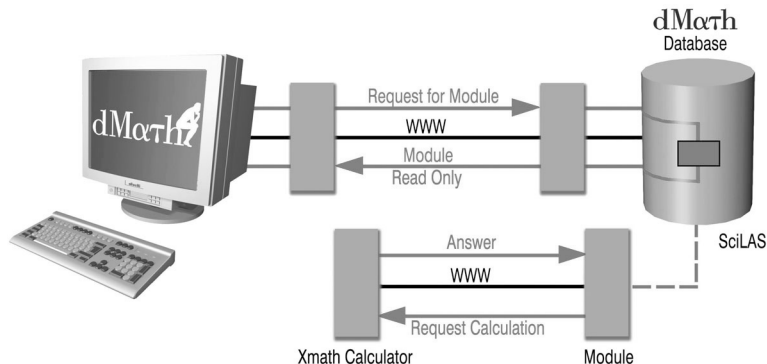


Figure 1. Information server component of the dMathArchive.

■ Xmath Step-by-Step Calculator

The Xmath calculator uses *Mathematica* packages loaded into *webMathematica*. The page design and menus are contained in XML files. Expressions are broken down into different levels using the *Mathematica* object *TreeForm*. All levels are identified in the output by Level n . On each level different mathematical rules (e.g., the substitution rule) may apply and the rule is identified by the package. Users can determine the number of levels in the input. Level 1 indicates the first hint for eventual further calculation by hand.

This is quite similar to what a professor would do on the blackboard; all rules and intermediate calculations are given. This is quite convenient for ODL applications where a professor is not nearby for face-to-face discussion. This has been tested in distance learning in mathematics for school teachers in Norway; a report may be ordered from anne.norstein@hisf.no.

The Xmath calculator contains a variety of mathematical topics, but not all of them give intermediate results. Some topics are still being developed. The algebra topic includes collecting terms, simplifying expressions, expanding expressions, polynomial factorization, partial fractions, step-by-step partial fractions, equation solving, step-by-step equation solving, solved problems, applications, assessments (mathematical comparison of given and answered expressions), case studies, and student projects.

The function topics include plotting, exact solving, numeric solve limits, step-by-step limits, differentiation, step-by-step differentiation, difference quotients, Newton's method, Taylor series, integration, step-by-step integration, definite integrals, Riemann sums, trapezoidal rule, Simpson's rule, solved problems, applications, assessments, case studies, and student projects.

Other topics will be plane curves, functions with several variables, and differential equations. Many of these will have step-by-step facilities.

□ Examples from the Calculator

The calculator can integrate, differentiate, solve equations, and so on for a wide range of expressions. Different levels are shown by scrolling and focusing only on the first level will give the user the first hint in a hand calculation. The program has discovered expressions the calculator can solve for integration but *Mathematica* cannot (certain substitutions).

Level 1

Find the integral $\int \frac{1}{(-3+x)^2} dx$

Substitution Rule, Composite function

Substitute $u = -3 + x$

This gives: $dx = du$, $\int \frac{1}{(-3+x)^2} dx = \int \frac{1}{u^2} du$

Level 2

Find the integral $\int \frac{1}{u^2} du$

Power Rule

$\int u^n du = \frac{1}{n+1} u^{n+1}$, Here $n = -2$

$\int \frac{1}{u^2} du = -\frac{1}{u}$

Substituting back $u = -3 + x$: $-\frac{1}{u} = -\frac{1}{-3+x}$

Result, Substitution Rule Composite function (Answer)

Substitute $u = -3 + x$

$\int \frac{1}{(-3+x)^2} dx = \int \frac{1}{u^2} du = -\frac{1}{u} = -\frac{1}{-3+x}$

Figure 2. web*Mathematica* output for integration.

Here is another example focusing on the first level.

Level 1

Find the integral $\int \frac{x^2}{\sqrt{1-x^6}} dx$

Substitution Rule, Composite function

Substitute $w = x^3$, $dx = \frac{dw}{3x^2}$, $\int \frac{x^2}{\sqrt{1-x^6}} dx = \int \frac{1}{3\sqrt{1-w^2}} dw$

Figure 3. Integration with the first hint for further calculation by hand.

The third example from Xmath uses partial fractions.

Find the Partial Fractions expansion of $\frac{x+2}{x(x^2+4)}$

Task 1

Finding Roots

Finding the roots of the denominator to be used in the denominators of the partial fractions, solving: $x(x^2+4)=0$

Solution: $x \in \{0, -2i, 2i\}$

Task 2

Partial Fractions

Divide into partial fractions:

$$\frac{x+2}{x(x^2+4)} = \frac{C}{x} + \frac{B+Ax}{x^2+4} = \frac{x(B+Ax)+C(x^2+4)}{x(x^2+4)} = \frac{(A+C)x^2+Bx+4C}{x(x^2+4)}$$

This gives: $(A+C)x^2+Bx+4C = x+2$

Task 3

Undetermined Coefficients

Using the method of undetermined coefficients (identical powers of free variable), we get the following system of linear equations:

$$4C = 2$$

$$B = 1$$

$$A + C = 0$$

Solving: $\{B \rightarrow 1, C \rightarrow \frac{1}{2}, A \rightarrow -\frac{1}{2}\}$

Answer

$$\frac{x+2}{x(x^2+4)} = \frac{1}{2x} + \frac{1-\frac{x}{2}}{x^2+4}$$

Figure 4. webMathematica output for partial fractions.

■ References

- [1] "Xmath Project." (Jan 24, 2005) dmath.hibu.no/xmath.
- [2] "dMath Project." (Jan 24, 2005) dmath.hibu.no.
- [3] "Minerva Action." (Nov 11, 2004) europa.eu.int/comm/education/programmes/socrates/minerva/ind1a_en.html.
- [4] "Leonardo da Vinci Community Vocational Training Action Programme." (Jan 24, 2005) leonardo.cec.eu.int.
- [5] "Xmath Calculator." xmath.hibu.no:8080/webMathematica/Xmath/Xcalc.msp.
- [6] "webMathematica." (Jan 24, 2005) www.wolfram.com/news/webm21.html.

- [7] O. Bringslid, "webMathematica: Interactive Mathematics on the Web," speech at the First European Workshop on MathML & Scientific e-Contents, Trieste, Italy 2002 www4mail.org/mathml2002/index.html.
- [8] "W3C Math Home." (Nov 15, 2004) www.w3.org/Math.
- [9] O. Bringslid, "Multimedia Books in the Mathematical Education of Engineers," *European Journal of Engineering Education*, **24**(2), 1999 pp. 189–201.
- [10] O. Bringslid, "Mathematical e-learning Using Interactive Mathematics on the Web," Mathematical Education of Engineers, *European Journal of Engineering Education*, **27**(3), 2002 pp. 249–255.
- [11] S. Wolfram, *The Mathematica Book*, 5th ed., Champaign, Oxford: Wolfram Media/Cambridge University Press, 2003.
- [12] Evaluation Report on Xmath in ODL (order from anne.norstein@hisf.no).
- [13] Full Report on Xmath. EU Commission, 2002 (order from odd.bringslid@hibu.no).
- [14] "First European Workshop on MathML & Scientific e-Contents." (Nov 21–22, 2002) www-dft.ts.infn.it/mathml.
- [15] "Second European Workshop on MathML & Scientific e-Contents." (Sep 16–18, 2004) dmath.savonia-amk.fi/secondWorkshop/servlet/Home/index.

About the Author

Odd Bringslid is an associate professor at Buskerud University College, Norway. He has also been the leader of the EU projects Xmath (2002–2004) and dMath (2003–2006). From 1981–1983, he was the leader of governmental educational projects, and a third-party developer for Hewlett Packard calculators (1992–1994). He has also received several scholarships including a 1995–1997 post-doctoral scholarship.

Odd Bringslid

*Department of Technology
Buskerud University College
Frogs vei 41 3603 Kongsberg, Norway
odd.bringslid@hibu.no*

Elements—An Object-Oriented Approach to Industrial Software Development

Gerd Baumann
Michał Mruk

This article discusses an object-oriented approach to industrial software development using *Mathematica*. We present the package *Elements* for structured representation of physical, engineering, and mathematical objects. This package introduces object-oriented paradigms into *Mathematica* and is used to develop a modeling environment built on a knowledge base where class and object properties and relations are maintained in a consistent, transparent, and extensible way. We show how this tool can be applied to design models parametrized by structured objects instead of just simple values.

■ Introduction

Models in engineering as well as in physics, chemistry, finance, mathematics, and economics are characterized by the following two main features.

- First, any model is based on methods representing an algorithm with which the problem is solved. These methods usually depend on each other or are at best completely independent.
- The second feature of any model is the occurrence of parameters determining physical, engineering, economic, or other properties and thus the behavior of the system.

The parameters are of central importance when variations of the model are examined. For example these quantities determine bifurcations to different states of the system. By *variations* we mean not only numeric changes but also symbolic ones.

However, modeling is a sophisticated endeavor of traversing between reality, imagination, and feasibility. It eventually converges to a mathematical descrip-

tion which captures the most essential aspects important for a particular purpose and, at the same time, is simple enough to be treated by formal methods.

We start with an observation that every modeling process involves working with objects and its primary goal is to select features relevant to a specific problem together with relations between them. Different problems in one specific area usually involve a rather limited number of objects. The problem itself is then reflected in the relations between them.

From this point of view it is natural to build a knowledge base containing descriptions of basic objects which often appear in modeling. Such a database would significantly speed up the modeling and provide a basis for rapid prototyping. It is not only a tedious and error-prone task to type in things over and over again, but, in many cases, parametrized solutions for recurring problems could be computed and stored in advance.

Our primary motivation is to facilitate the modeling process in engineering by providing a transparent and easy-to-use knowledge base of objects. In this article we stipulate and analyze requirements on such a system and present a *Mathematica* package to store and retrieve information about objects.

The proposed system is designed to be, in its nature, primarily static. It is not supposed to include “automatic solvers”, although internal and external algorithms will be utilized to perform computation whenever this appears appropriate. Extensions of the knowledge base will always be checked by a human. The reason is that the vast majority of practical problems do not provide easy answers. Even though today’s computer systems are very powerful, solutions obtained automatically still need to be post-processed by experts. This makes it impossible to simply add them to an existing knowledge base since sooner or later its consistency would inevitably be destroyed, making it useless.

□ Objects in Modeling

When we analyze requirements on the design of a system for object representation, we naturally come across some principles already used for years in software engineering: object-oriented programming. It is not surprising that the same or very similar principles will govern the design here.

Object-oriented programming introduced the concepts of classes and inheritance. It can be observed that solutions are often simpler when a hierarchical structure is imposed on the problems.

Another aspect of object-oriented programming may be even more important. Classes and their objects are encapsulated, stand-alone, and well-specified entities that behave the same way in any program—this is a key property of reusable components. This makes it possible to publish code which can be used in various settings without change.

It is clear that these paradigms apply to modeling as well. A hierarchical structure is already in the nature of all mathematical and physical objects while the closeness of used entities is a prerequisite for the use of objects in different environ-

ments. These two concepts will play a primary role in the design of a modeling environment.

□ Object-Oriented Methods in Computer Algebra

The fact that principles of object-oriented programming are compatible with structures in scientific computing led to adopting those methods into computer algebra systems. A proper combination of these two fields yields the key to a solution of our problem. On the one hand, object-oriented design provides for encapsulation and reuse of objects. On the other hand, computer algebra offers a variety of algorithms to work with them.

However, up to now there is no single system that contains these features in a well-balanced symbiosis. We encounter either systems offering implementation of the class concept (C++, Java, SML, Aldor, and so forth), or single- or multi-purpose computer algebra systems providing more or less large sets of algorithms (Maple, *Mathematica*, GAP, and so forth).

In spite of this fact, a lot of successful work has already been done using object-oriented methods in computer algebra systems. There is an extensive literature on object-oriented numerical methods [1, 2, 3, 4]. Numerical algorithms benefit from the fact that most of the data types they work on are largely supported by classical programming languages so there are quite a few choices. However, numerics do not belong to areas that would primarily benefit from object-oriented paradigms.

Another approach integrates object-oriented programming languages and computer algebra in one application [5, 6, 7, 8]. However, these solutions are mostly aimed at a few specialized fields.

The last option is to stay solely within a computer algebra system. For a long time, Maple lacked a proper concept for data encapsulation. It was finally included in Version 7. An example of object-oriented programming using numerical computation in Maple can be found in [9]. In spite of this, Maple's imperative nature is not well suited for the complex manipulations on data and code necessary when implementing object-oriented features.

Probably the first implementation of classes in *Mathematica* was done by R. Maeder in [10]. Even though this code was just a proof of concept, it demonstrated the strength of the language and proved that *Mathematica* provides powerful means for building structures and for data encapsulation.

Applications in engineering require an extensive collection of algorithms which are currently provided only by general computer algebra systems like *Mathematica* or Maple. In this article we present the package *Elements*, which provides an implementation of object-oriented features to facilitate modeling directly within *Mathematica* [11, 12]. The package, which is available from the authors' company [13], goes beyond Maeder's work and has proved to be well suited for the purpose of building a knowledge base of objects from which models are built. In the next section, we postulate and analyze requirements for a working environment that allows rapid development of models in engineering.

■ Examples

□ Car on a Bumpy Road

The problem considered here is a practical application originating from the early phase of a car design process. Engineers in the automotive business have to know in advance how a new car will react under certain environmental conditions. In the sequel, we discuss a procedure for building a model that provides basic information on the behavior of a car at the very beginning of its design. In this early stage of the process, very little information on the new car is available and thus it is sufficient to reduce the model to the most essential properties of the car. These properties can include the mass distribution on a chassis, that is, the location of the engine and other heavy components like the fuel tank, batteries, gear box, and so on. Another essential feature in this early design phase is the undercarriage. In our model, these components are combined into a multibody system interacting by forces with each other and with the environment. Knowing the preliminary behavior of the new car, the next step is to improve the model over several iterations to incorporate more details about the interacting components. Here, we restrict our considerations to the very beginning of the design process and demonstrate how an object-oriented approach can be used to model different components in a transparent way.

Figure 1 shows the simplified one-dimensional mechanical model of a car moving on a bumpy road. In principle, the car consists of three main components: the two axes and the car body. We assume that the two axes are coupled to the chassis by a dashpot and a spring. In addition to the translation, we allow a narrow rotation of the body. The wheels are in contact with the road (this interaction is modeled by introducing springs and dampers representing the two tires). The symbols shown in the figure have the following meaning: k_i , d_i with ($i = 1, 2, 3, 4$), m_k with ($k = 1, 2, 3$), a , and b are the physical and geometric parameters of the system. The k_i and d_i denote force and damping constants, m_k are the wheel and chassis masses, and a and b determine the center of the mass of the chassis. $J_{C2} = I$ is the inertia moment of the chassis. The dynamical coordinates denoted by z_1 , z_2 , z_3 , and β describe the vertical and angular motion of the car. The acting external forces on the car are due to bumps in the road and gravity. Now, the whole system can be described by Newton's equations.

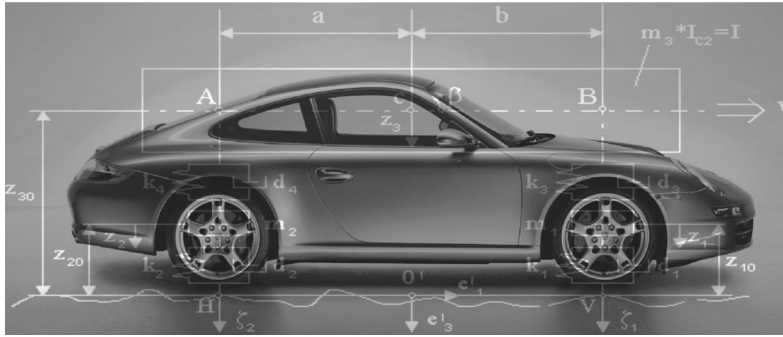


Figure 1. A simplified version of a real car. The model uses a multi-body system with spring and damper interactions. The components of the car are the axis, the car body, wheels, and the road as elements.

These equations are given for the coordinates by

$$m_1 z_1'' = -F_1 + F_3 \quad (1)$$

$$m_2 z_2'' = -F_2 + F_4 \quad (2)$$

$$m_3 z_3'' = F_3 \quad (3)$$

$$I \beta'' = b F_3 - a F_4, \quad (4)$$

where F_i are the acting forces. One of the main problems in car design is to estimate how the total system will react under different forces, different damping parameters, different velocities, different environment conditions, and so on. These questions are partially answered in the following subsections. The complete examination of all these inquiries would exceed the space available for this article. Hence, we will restrict our consideration to a few examples.

To attack the physical and mathematical aspects we first identify classes that would capture the essence of this problem. If we look at the equations of motion and at Figure 1, it seems natural to divide the car into three main classes: the body, the axes, and the environment.

Class Definitions

The problem can be split into the following classes: the general setup class, the car body, the axes, the car itself, and some tools needed to carry out the simulation. The simulation class that provides graphical output refers to other classes that serve as a basis for the computation required to produce the output.

General Setup

The general setup is concerned with geometric properties and the influence of the environment, such as the external force characterized here by four bumps in the road.

```

In[1]:= Setup = Class["Setup", Class["Element"],
  {{a = 1.2, Description → "distance front axis"},
   {b = 1.2, Description → "distance rear axis"},
   {v = 5, Description → "velocity of the car"},
   {stepWidth = 0.1, Description → "step width"},
   {stepHeight = 0.07, Description → "step height"}},
  {extForce[t_] :=
    Fold[Plus, 0, Table[(Tanh[100*v*(t - n stepWidth)] -
      Tanh[100*(v*(t - n stepWidth) - stepWidth)])*
      stepHeight/2, {n, 1, 4}]]}
]

```

```
Out[1]= <Class Setup>
```

Body

The class Body consists solely of geometric and physical parameters of the car body. This class also includes the initial conditions for displacement, angle, and velocity.

```

In[2]:= Body = Class["Body", Setup,
  {description = "Body",
   {m = 1200, Description → "mass"},
   {z0 = 0, Description → "initial displacement"},
   {zp0 = 0, Description → "initial velocity"},
   {h = 0.75, Description → "geometric quantity"},
   {β0 = 0, Description → "initial value for β"},
   {βp0 = 0, Description → "initial value for β prime"},
   {zFunName = z, Description → "name of the z function"},
   {betaFunName = β, Description → "name of the β function"}},
  {}]

```

```
Out[2]= <Class Body>
```

Axes

The common properties of the two axes are collected in the class CarAxis. This class contains only parameters: force and damping constants together with initial conditions for the equations of motion.

```

In[3]:= CarAxis = Class["CarAxis", Setup,
  {description = "Axis",
   {m = 42.5, Description → "mass of the axis"},
   {k1 = 150000, Description → "wheel-body force constant"},
   {d1 = 700, Description → "wheel-body damping constant"},
   {k2 = 4000, Description → "wheel-road force constant"},
   {d2 = 1800, Description → "wheel-road damping constant"},
   {z0 = 0, Description → "initial displacement"},
   {zp0 = 0, Description → "initial velocity"},
   {zFunName = z, Description → "name of the function z"}},
  {}]

```

```
Out[3]= <Class CarAxis>
```

In the next step we define classes for the front and rear axes. The front axis consists solely of a method to generate the equation of motion for the axis by considering the acting forces.

```
In[4]:= FrontAxis = Class["FrontAxis", CarAxis,
  {},
  {getEquations[body_ /; ObjectOfClassQ[body, Body]] :=
    Block[{f1 = k1 * (zFunName[t] - extForce[t]) +
      d1 * (zFunName'[t] - D[extForce[t], t]),
      f3 = k2 * ((body ° zFunName)[t] - b * (body ° betaFunName)[t]
        - zFunName[t]) + d2 * (D[(body ° zFunName)[t] -
          b * (body ° betaFunName)[t], t] - zFunName'[t])},
    {m * zFunName''[t] + (-f1 + f3) * -1 == 0,
      zFunName[0] == z0, zFunName'[0] == zp0}]
  }
]
```

```
Out[4]= <Class FrontAxis>
```

The rear axis is generated in the same way by taking the relations specific to this component into account.

```
In[5]:= RearAxis = Class["RearAxis", CarAxis,
  {},
  {getEquations[body_ /; ObjectOfClassQ[body, Body]] :=
    Block[{f2 = k1 * (zFunName[t] - extForce[t - (a + b) / v]) +
      d1 * ((body ° zFunName)'[t] - D[extForce[t - (a + b) / v], t]),
      f4 = k2 * ((body ° zFunName)[t] + a * (body ° betaFunName)[t]
        - zFunName[t]) + d2 * (D[(body ° zFunName)[t] +
          a * (body ° betaFunName)[t], t] - zFunName'[t])},
    {m * zFunName''[t] + (-f2 + f4) * -1 == 0,
      zFunName[0] == z0, zFunName'[0] == zp0}]
  }
]
```

```
Out[5]= <Class RearAxis>
```

Car

Finally, after setting up classes for the body and axes, we define a class to describe the car. This class incorporates parameters as well as methods to combine information from different components.

```
In[6]:= Car = Class["Car", Setup,
  {description = "Car",
    {body = Null, Description → "body"},
    {frontAxis = Null, Description → "front axis"},
    {rearAxis = Null, Description → "rear axis"}},
  {Ic[a0_, b0_, h0_, m03_] := Block[{}, m03 / 12 * ((a0 + b0)^2 + h0^2)],
  getEquations[] := Block[
    {f3 = frontAxis ° k2 *
      ((body ° zFunName)[t] -
        b * (body ° betaFunName)[t] - (frontAxis ° zFunName)[t]) +
      frontAxis ° d2 *
        (D[(body ° zFunName)[t] - b * (body ° betaFunName)[t], t] -
```

```

        (frontAxis◦zFunName)'[t]),
f4 = rearAxis◦k2*
    ((body◦zFunName)[t] +
     a*(body◦betaFunName)[t] - (rearAxis◦zFunName)[t]) +
    rearAxis◦d2*
    (D[(body◦zFunName)[t] + a*(body◦betaFunName)[t], t] -
     (rearAxis◦zFunName)'[t])),
Union[{body◦m*(body◦zFunName)''[t] + 2 f3 == 0,
      (body◦zFunName)[0] == body◦z0,
      (body◦zFunName)'[0] == body◦zp0, Ic[a, b, body◦h, body◦m] *
      (body◦betaFunName)''[t] + (b*f3 - a*f4)*-1 == 0,
      (body◦betaFunName)[0] == body◦β0,
      (body◦betaFunName)'[0] == body◦βp0},
      frontAxis◦getEquations[body],
      rearAxis◦getEquations[body]]
],
getVariables[] := {frontAxis◦zFunName, rearAxis◦zFunName,
                  body◦zFunName, body◦betaFunName}}
]

```

Out[6]= <Class Car>

Simulation

Having the classes for all physical components available, we need a tool to carry out the simulation. The following classes provide an efficient simulation of the model and generate a graphical representation of the results.

```

In[7]:= MakeSimulation = Class["MakeSimulation", Class["Element"],
  {description = "Simulation of the motion",
   {car = Null, Description → "Car"},
   {maxSteps = 100000,
    Description → "Maximum number of steps in NDSolve"}},
  {simulate[tstart_, tend_] :=
   Block[{nsol = getNSolution[tstart, tend]},
    $TextStyle = {FontFamily → "Arial", FontSize → 12};
    MapThread[Plot[Evaluate[{{car◦extForce[t]}, #1 /. nsol}],
      {t, tstart, tend}, PlotRange → All,
      PlotStyle → {RGBColor[0.25098, 0, 0.25098],
        {RGBColor[0.9, 0, 0], AbsoluteThickness[3]}},
      AxesLabel → {"t [s]", ""}, PlotLabel → #2,
      GridLines → Automatic, Frame → True] &,
    {Map[Apply[#, {t}] &, car◦getVariables[]], {"Front Axis",
      "Rear Axis", "Body Movement", "Rotation Angle"}];],
   getNSolution[tstart_, tend_] :=
    NDSolve[car◦getEquations[],
      car◦getVariables[], {t, tstart, tend},
      MaxSteps → maxSteps]}
]

```

Out[7]= <Class MakeSimulation>

If we wish to see the motion of the car, it is convenient to declare a special class for making animations.

```

In[8]:= << Graphics`Shapes`

In[9]:= MakeAnimation = Class["MakeAnimation", Class["Element"],
  {description = "Animate the motion of the car"
   {path = "C:/Mma/WORK/TUMObjects/Elements05/Library/Cars/
      EEVCMedium/Generic", Description → "Path of the car model"
   }
  },
  {animate[sim1_, tstart_, tend_, δt_] :=
    Block[{names, l1, tire1, tire2, window1, window2, window3, body,
      sol = sim1°getNSolution[tstart, tend]},
      SetDirectory[path];
      names = FileNames["*."];
      l1 = Map[ReadList[#, Number, RecordLists → True,
        RecordSeparators →
          {If[StringMatchQ[$0operatingSystem, "Windows*"],
            "\n", "\r"}] &, names];
      tire1 = {GrayLevel[0.752941], Polygon[
        l1[[1]] /. {x_, y_} → {x, y + z1[t]}]};
      tire2 = {GrayLevel[0.752941], Polygon[
        l1[[2]] /. {x_, y_} → {x, y + z2[t]}]};
      window1 = {GrayLevel[0.752941], Polygon[
        l1[[3]] /. {x_, y_} → {x Cos[-β[t]] + y Sin[-β[t]],
          y Cos[-β[t]] - x Sin[-β[t]] + z3[t]}]};
      window2 = {GrayLevel[0.752941], Polygon[
        l1[[4]] /. {x_, y_} → {x Cos[-β[t]] + y Sin[-β[t]],
          y Cos[-β[t]] - x Sin[-β[t]] + z3[t]}]};
      window3 = {GrayLevel[0.752941], Polygon[
        l1[[5]] /. {x_, y_} → {x Cos[-β[t]] + y Sin[-β[t]],
          y Cos[-β[t]] - x Sin[-β[t]] + z3[t]}]};
      body = {RGBColor[0, 0, 0.25098], Polygon[
        l1[[6]] /. {x_, y_} → {x Cos[-β[t]] + y Sin[-β[t]],
          y Cos[-β[t]] - x Sin[-β[t]] + z3[t]}]};
      Table[Show[Graphics[{body, tire1, tire2, window1, window2,
        window3} /. sol], AspectRatio → Automatic, Axes → False,
        Frame → True, PlotRange → {{-.1, 4.4}, {0, 1.4}},
        FrameTicks → None], {t, tstart, tend, δt}];
    ]}
  ]
}

Out[9]:= <Class MakeAnimation>

```

Objects

Up to this point, nothing more than a framework was defined in which the computation can be carried out. The classes represent essentially only templates for creating the real and calculation objects. In order to set up the simulation, we have to generate objects and specify their properties. The car is generated by a modular process incorporating the distinguished objects. At this point, it is apparent that we have a great advantage in designing a specific car compared to traditional approaches in simulation. For example, we can define different objects for axes and use them in different simulations as parameters of the underlying

model. Components (objects) of the same type can be exchanged without causing any harm to the system. In the sequel, we demonstrate how this process is carried out.

Body

First, we create the body by calling the method `new` of the class `Body`. All properties except `zFunName` and `betaFunName` will have their default values specified in the class declaration.

```
In[10]:= b = Body◦new[{zFunName → z3, betaFunName → β}]
```

```
Out[10]= <Object of Body>
```

The actual properties of the body can be checked by

```
In[11]:= GetPropertiesForm[b]
```

```
Out[11]//DisplayForm=
```

Property	Value
description	Body
a	1.2
b	1.2
betaFunName	β
h	0.75
m	1200
stepHeight	0.07
stepWidth	0.1
v	5
z0	0
zFunName	z3
zp0	0
β 0	0
β p0	0

Axes

Next, objects for the axes are created. The front axis is defined with a mass of 45.5 kg; the function representing the displacement of the center of the wheel attached to this axis is denoted by z_1 .

```
In[12]:= fa = FrontAxis◦new[{m → 45.5, zFunName → z1}]
```

```
Out[12]= <Object of FrontAxis>
```

The properties of the axis can be printed by

```
In[13]:= GetPropertiesForm[fa]
```

```
Out[13]//DisplayForm=
```

Property	Value
description	Axis
a	1.2
b	1.2
d1	700
d2	1800
k1	150000
k2	4000
m	45.5
stepHeight	0.07
stepWidth	0.1
v	5
z0	0
zFunName	z1
zp0	0

Since no values for the damping and the force constant were given in the call to `new`, these properties are initialized with values specified in the class declaration.

A second version of the front axis with a stiffer damping is derived in the following line. Here, k_1 and k_2 were given different initial values.

```
In[14]:= fas = FrontAxis ◦ new[{m → 45.5,
    d1 → 5000, d2 → 5000,
    k1 → 150000, k2 → 150000,
    zFunName → z1}]
```

```
Out[14]= <Object of FrontAxis>
```

For the rear axis, we just use the default settings. The coordinate of elongation is denoted by z_2 .

```
In[15]:= ra = RearAxis ◦ new[{zFunName → z2}]
```

```
Out[15]= <Object of RearAxis>
```

A check of parameters shows the default values.

```
In[16]:= GetPropertiesForm[ra]
```

```
Out[16]//DisplayForm=
```

Property	Value
description	Axis
a	1.2
b	1.2
d1	700
d2	1800
k1	150000
k2	4000
m	42.5
stepHeight	0.07
stepWidth	0.1
v	5
z0	0
zFunName	z2
zp0	0

Car

Having all necessary components available, it is easy to generate the car by incorporating the defined objects into a common model. The car object is defined by

```
In[17]:= c = Car◦new[{frontAxis → fas, rearAxis → ra, body → b}];
```

It consists of the front axis, the rear axis, and the body. We want to stress that object-oriented design allows the behavior of the model to be modified very easily by changing different components or their properties. Hence, if an engineer has access to a collection of different bodies, axes, and other objects, he would be able to create and test many different car designs without needing to adapt the model.

Simulation

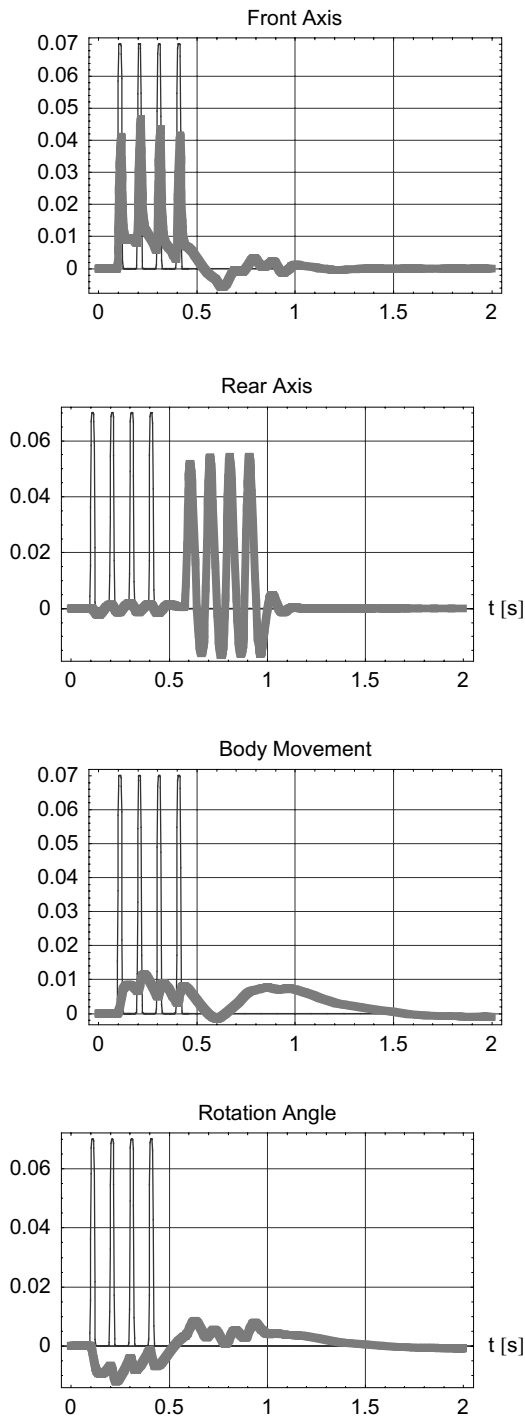
A simulation object is created from the class `MakeSimulation`. The simulation can be performed with any object of the class `Car` that is assigned to the property `car` in the call to the method `new`.

```
In[18]:= sim = MakeSimulation◦new[{car → c}]
```

```
Out[18]= <Object of MakeSimulation>
```

Finally, the actual simulation is initiated by calling the `simulate` method of the class `MakeSimulation`. The inputs for this function are just the starting and end points of the simulation interval. The result is four plots for the coordinates of the axes, the body, and the rotation angle of the body. All quantities are plotted versus the simulation time.

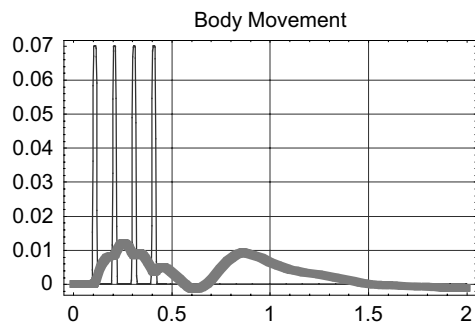
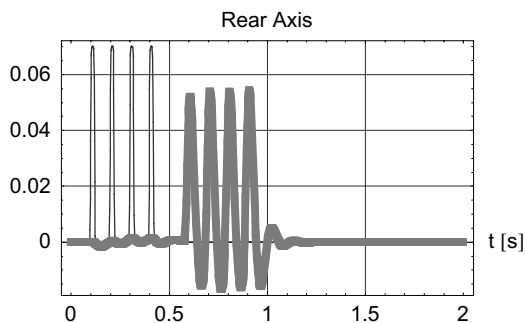
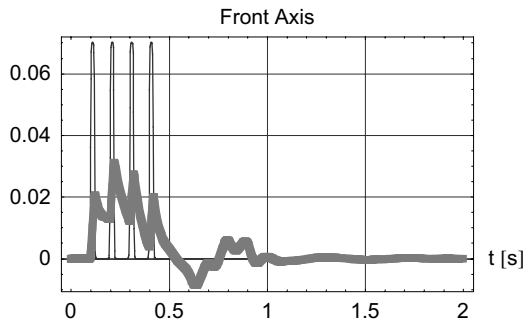
```
In[19]:= sim◦simulate[0, 2]
```

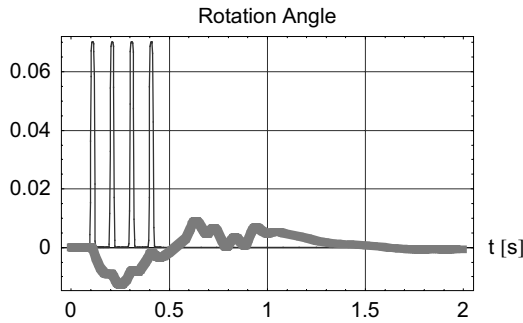


Now the hard work for an engineer starts. If he is interested in an optimization of the elongations, he must change the properties of the car in the simulation.

How to achieve the best result and how to select the best strategy depends on the experience of the individual engineer. However, within an object-oriented simulation environment, he has the flexibility to incorporate his thoughts in a quick and transparent way. For example, he can change some parameters of the front axis to achieve a better performing car.

```
In[20]:= sim◦car◦frontAxis◦d1 = 700;  
         sim◦car◦frontAxis◦k1 = 100000;  
  
In[22]:= sim◦simulate[0, 2]
```





Or he can replace the whole front axis by one with a smoother damper.

```
In[23]:= sim◦car◦frontAxis = fa
```

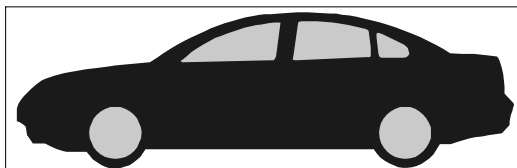
```
Out[23]= <Object of FrontAxis>
```

```
In[24]:= sim◦simulate[0, 2]
```

And so on (the output of the previous line was suppressed due to space limits). However, a clever engineer will resort to some optimization strategies that can be carried out by an additional class in the development environment (not shown here). In this article, we considered only manual computations to demonstrate the simulation steps.

Finally, if the results are to be presented at meetings, the behavior of the optimized car can be shown online as an animation using different simulation models or strategies. The generation of an animation object and the animation itself is started with

```
In[25]:= an = MakeAnimation◦new[] ;
          an◦animate[sim, 0, 2, 0.05]
```



The flip chart movie is suppressed in the printed version. However, the notebook version of this article shows you the movement of the car body and the wheels.

This example demonstrated that the object-oriented approach to modeling is very efficient and transparent to the user. It divides the process of modeling into two phases. In the first phase, the model and its components are implemented. In the optimization phase, the parameters of the model satisfying certain criteria are found. Both phases benefit from the availability of predefined components (objects). Using the classical approach, minor structural changes (e.g., using a different axis) require modifications by the model builder. This, in turn, may have made a subsequent verification step necessary. In contrast, the object-

oriented approach allows this to be done much more easily. A new component (e.g., an axis) can be added or exchanged without affecting the soundness of the model. This fact is of crucial importance in the industrial environment where models contain a huge number of parameters. It is necessary to have a clear and well-founded procedure to handle them. If the process to manage and work with models is clearly specified, results can be obtained more quickly, they are more predictable, and comprehensible.

□ Dynamics in Poisson–Hamilton Manifolds

Another example of an application of *Elements* in a mathematical context is the definition of operators and manifolds for Poisson–Hamiltonian systems. The problem here can be stated as follows: A set of generalized coordinates and momenta in connection with a Hamiltonian defines a Hamiltonian manifold. This manifold is also known as the phase space governed by the Hamiltonian. We assume that in addition to coordinates and momenta an additional function generating the dynamics exists. This function is usually the Hamiltonian of the system. Having the coordinates and the generating function available, the manifold acquires an algebraic structure. The algebra on the Hamiltonian manifold is generated by the Poisson’s bracket. The total Poisson–Hamilton manifold is thus equipped with coordinates, momenta, the Hamiltonian, and the Poisson’s bracket.

The question is: How do these entities relate to objects and classes in an object-oriented environment? From a practical point of view, it is necessary to define the Poisson manifold given by the coordinates and momenta associated with the algebraic structure. Thus, one class in the Poisson–Hamilton manifold represents the Poisson bracket with its properties for a given set of coordinates and momenta. The basic algebraic properties which a Poisson bracket has to satisfy are discussed next.

Let us consider two functions f and g depending on the phase space variables. Using these functions in the Poisson bracket, we can derive some general properties for this kind of bracket defined by

$$\{f, g\}_{(q,p)} = \sum_{k=1}^s \frac{\partial f}{\partial q_k} \frac{\partial g}{\partial p_k} - \frac{\partial f}{\partial p_k} \frac{\partial g}{\partial q_k}. \quad (5)$$

In the following, we use the shorthand notation $\{f, g\}$ to represent the Poisson bracket $\{f, g\}_{(q,p)}$. This notation is used whenever no confusion about the phase space variables can arise. The Poisson bracket possesses the following properties:

$$\{f, g\} = -\{g, f\} \quad \text{antisymmetry.} \quad (6)$$

If one of the functions f or g is a constant, the Poisson bracket vanishes

$$\{f, c\} = 0 = \{c, g\}. \quad (7)$$

If we have three functions f , g , and h living in the phase space, then the following properties hold:

$$\{f + h, g\} = \{f, g\} + \{h, g\} \quad \text{linearity} \quad (8)$$

$$\{f h, g\} = f \{h, g\} + h \{f, g\} \quad \text{Leibniz's rule} \quad (9)$$

$$\frac{\partial}{\partial t} \{f, g\} = \left\{ \frac{\partial f}{\partial t}, g \right\} + \left\{ f, \frac{\partial g}{\partial t} \right\} \quad \text{differentiation rule.} \quad (10)$$

If one of the two functions f or g reduces to a phase space variable, the Poisson bracket reduces to the partial derivative of the function with respect to the conjugate coordinate. For example, if g is equal to the generalized coordinate q_k or the generalized momentum p_k , the result of the Poisson bracket is

$$\{f, q_k\} = - \frac{\partial f}{\partial p_k} \quad (11)$$

$$\{f, p_k\} = \frac{\partial f}{\partial q_k}. \quad (12)$$

If we chose the coordinates of the phase space for both f and g , we obtain the fundamental Poisson brackets

$$\{q_i, q_j\} = 0 \quad (13)$$

$$\{p_i, p_j\} = 0 \quad (14)$$

$$\{q_i, p_i\} = \sum_{k=1}^s \frac{\partial q_i}{\partial q_k} \frac{\partial p_j}{\partial p_k} - \frac{\partial q_i}{\partial p_k} \frac{\partial p_j}{\partial q_k} = \sum_{k=1}^s \delta_{ik} \delta_{jk} = \delta_{ij}. \quad (15)$$

These relations of the fundamental Poisson brackets represent the algebraic structure and are the basis of quantum mechanics. Any three functions of the phase space satisfy a special relation, the so-called Jacobi identity

$$\{f, \{g, h\}\} + \{g, \{h, f\}\} + \{h, \{f, g\}\} = 0. \quad (16)$$

These properties determine the algebraic quality of the Poisson bracket. Especially, linearity, antisymmetry, and the Jacobi identity define the related Lie algebra of the bracket.

Another important property of the Poisson bracket is the ability to derive from two conserved quantities $\mathcal{J}_1$ and $\mathcal{J}_2$ another conserved quantity

$$\{\mathcal{J}_1, \mathcal{J}_2\} = \text{const.} \quad (17)$$

This behavior is known as Poisson's theorem. A direct proof is feasible if we assume that $\mathcal{J}_1$ and $\mathcal{J}_2$ are independent of time. By replacing the third function in the Jacobi identity by the Hamiltonian of the system, we get

$$\{H, \{\mathcal{J}_1, \mathcal{J}_2\}\} + \{\mathcal{J}_1, \{\mathcal{J}_2, H\}\} + \{\mathcal{J}_2, \{H, \mathcal{J}_1\}\} = 0. \quad (18)$$

Since $\{\mathcal{J}_2, H\} = 0$ and $\{H, \mathcal{J}_1\} = 0$, we find

$$\{H, \{\mathcal{J}_1, \mathcal{J}_2\}\} = 0. \quad (19)$$

Thus, the bracket $\{J_1, J_2\}$ is also a conserved quantity. We note that the application of Poisson's theorem will not always provide new conserved quantities because the number of conserved quantities of a standard mechanical system is finite. It is known that the total number of conserved quantities is given by $2n - 1$, if n is the degree of freedom in the phase space. Thus, Poisson's theorem sometimes delivers trivial constants or the resulting conserved quantity is a function of the original conserved quantities J_1 and J_2 . If both cases fail, we gain a new conserved quantity.

It is obvious from the previous definitions and properties that the pure algebraic structure is independent of the phase space variables. Thus we can separate the problem into the algebraic properties (equations (6) through (10)) and the coordinate-specific definition of the PB given by equation (5). This separation is the basis of our class for the Poisson bracket PB.

Poisson Brackets

The following definition of the class PB represents a general definition of the Poisson bracket. This class is determined by the algebraic properties and does not incorporate the coordinates on which it operates. The class definition expresses properties, such as bilinearity, the product relation for each argument of the Poisson bracket, and the calculation of the Poisson expression. The properties implemented represent the algebraic structure of a Lie algebra.

```
In[1]:= PB = Class["PB", Class["Element"],
  {description = "Poisson Bracket",
    {P = Null, Description → "momenta"},
    {Q = Null, Description → "coordinates"},
    {T = t, Description → "independent variable"}},
  (* --- constant factor extraction --- *)PoissonBracket[a_ f_, g_] :=
    a PoissonBracket[f, g] /; (Apply[And, Map[FreeQ[a, #] &, Q]] &
      Apply[And, Map[FreeQ[a, #] &, P]]),
  (* --- constant factor extraction --- *)
  PoissonBracket[f_, a_ g_] :=
    a PoissonBracket[f, g] /; (Apply[And, Map[FreeQ[a, #] &, Q]] &
      Apply[And, Map[FreeQ[a, #] &, P]]),
  (* --- Linearity --- *)
  PoissonBracket[a_ + f_, g_] :=
    PoissonBracket[a, g] + PoissonBracket[f, g],
  (* --- Linearity --- *)
  PoissonBracket[f_, a_ + g_] :=
    PoissonBracket[f, a] + PoissonBracket[f, g],
  (* --- product relation --- *)
  PoissonBracket[f_ h_, g_] :=
    f PoissonBracket[h, g] + h PoissonBracket[f, g],
  (* --- product relation --- *)
  PoissonBracket[g_, f_ h_] :=
    f PoissonBracket[g, h] + h PoissonBracket[g, f],
  (* --- Calculation of the bracket --- *)
  PoissonBracket[f_, g_] := Block[{f},
    Fold[Plus, 0, MapThread[(∂#1 f ∂#2 g - ∂#2 f ∂#1 g) &, {Q, P}]]],
```

```
PB[Pnew_List, Qnew_List, Tnew_Symbol] :=
  Block[{}, P = Pnew; Q = Qnew; T = Tnew]
]
```

Out[1]= <Class PB>

Our aim here is to use a Poisson bracket within *Mathematica* in the same way as in a textbook. This requires that we first define the Poisson manifold as an object and use this object in our calculations. The following line generates a template for the Poisson bracket combining the Poisson manifold as an object and the algebraic properties of the bracket.

```
In[2]:= Notation[{f_, g_}obj_ ⇔ obj_◦PoissonBracket[f_, g_]];
```

This allows us to proceed in the same way as in mechanics textbooks.

A Simple Poisson Bracket

In the following, we give a few examples of Poisson manifolds and their application to a function defined on them.

The first example describes a two-dimensional manifold in generalized coordinates p and q . The manifold (object $\mathbf{pk}$) is derived from the class PB.

```
In[3]:= pk = PB◦new[{P → {p[t]}, Q → {q[t]}}]
```

Out[3]= <Object of PB>

The properties of this manifold can be printed using

```
In[4]:= GetPropertiesForm[pk]
```

Out[4]//DisplayForm=

Property	Value
description	Poisson Bracket
P	{p[t]}
Q	{q[t]}
T	t

We get the coordinates q and p depending by default on time t .

Now, let us apply the Poisson manifold $\mathbf{pk}$ to two functions f and g depending on p and q .

```
In[5]:= {f[p[t], q[t]], g[p[t], q[t]]}pk
```

Out[5]= $-g^{(0,1)}[p[t], q[t]] f^{(1,0)}[p[t], q[t]] + f^{(0,1)}[p[t], q[t]] g^{(1,0)}[p[t], q[t]]$

The result is the definition of the Poisson bracket given in equation (5) for $s = 1$.

A second example is concerned with the Hamiltonian of a single particle moving in a potential $V = V(q)$. The application to this Hamiltonian provides us with the momentum p .

```
In[6]:= {  $\frac{p[t]^2}{2} + v[q[t]]$ , p[t] }pk
```

Out[6]= $V'[q[t]]$

Another example of a general function H depending on both coordinates yields for the q coordinate

$$\text{In}[7]:= \{H[p[t], q[t]], q[t]\}_{pk}$$

$$\text{Out}[7]= -H^{(1,0)}[p[t], q[t]]$$

An additional example deals with a general Hamiltonian H and an arbitrary function f depending on both coordinates of the manifold.

$$\text{In}[8]:= \{\alpha H[q[t], p[t]], f[q[t], p[t]]\}_{pk}$$

$$\text{Out}[8]= \alpha (-H^{(0,1)}[q[t], p[t]] f^{(1,0)}[q[t], p[t]] + f^{(0,1)}[q[t], p[t]] H^{(1,0)}[q[t], p[t]])$$

This result demonstrates that linearity is recognized by the manifold. The next relation represents Jacobi's identity for three functions H , f , and g depending on both coordinates.

$$\begin{aligned} \text{In}[9]:= & \{\alpha H[q[t], p[t]], \{f[q[t], p[t]], g[q[t], p[t]]\}_{pk}\}_{pk} + \\ & \{f[q[t], p[t]], \{g[q[t], p[t]], \alpha H[q[t], p[t]]\}_{pk}\}_{pk} + \\ & \{g[q[t], p[t]], \{\alpha H[q[t], p[t]], f[q[t], p[t]]\}_{pk}\}_{pk} // \text{Simplify} \end{aligned}$$

$$\text{Out}[9]= 0$$

Higher-Dimensional Manifolds

The class PB is defined in such a way that it allows creating a general Poisson manifold in $2n$ -dimensions. To make things easier, let us define a four-dimensional Poisson manifold with coordinates q_i and p_i ($i = 1, 2$). The manifold is derived from the class PB and stored in the object pb2.

$$\text{In}[10]:= \text{pb2} = \text{PB} \circ \text{new}[\{P \rightarrow \{p1[t], p2[t]\}, Q \rightarrow \{q1[t], q2[t]\}\}]$$

$$\text{Out}[10]= \langle \text{Object of PB} \rangle$$

The properties of this Poisson manifold are

$$\text{In}[11]:= \text{GetPropertiesForm}[\text{pb2}]$$

$$\text{Out}[11]//\text{DisplayForm}=$$

Property	Value
description	Poisson Bracket
P	{p1[t], p2[t]}
Q	{q1[t], q2[t]}
T	t

Again, the momenta and coordinates are defined in terms of the independent variable t .

Global relations incorporating all coordinates for a two-particle Hamiltonian can be obtained by the following line

$$\text{In}[12]:= \text{Map}\left[\left\{\frac{\mathbf{p1}[\mathbf{t}]^2}{2} + \frac{\mathbf{p2}[\mathbf{t}]^2}{2} + \mathbf{v}[\mathbf{q1}[\mathbf{t}], \mathbf{q2}[\mathbf{t}]], \# \right\}, \# \right] \&, \\ \{\mathbf{p1}[\mathbf{t}], \mathbf{p2}[\mathbf{t}], \mathbf{q1}[\mathbf{t}], \mathbf{q2}[\mathbf{t}]\}$$

$$\text{Out}[12]= \{V^{(1,0)}[\mathbf{q1}[\mathbf{t}], \mathbf{q2}[\mathbf{t}]], V^{(0,1)}[\mathbf{q1}[\mathbf{t}], \mathbf{q2}[\mathbf{t}]], -\mathbf{p1}[\mathbf{t}], -\mathbf{p2}[\mathbf{t}]\}$$

The function used as the first argument of the Poisson bracket is a Hamiltonian consisting of two terms for the kinetic energy and a general interaction potential V . For another two-dimensional Hamiltonian with a different potential V , we find

$$\text{In}[13]:= \text{Map}\left[\left(\left\{\frac{\mathbf{p1}[\mathbf{t}]^2}{2} + \frac{\mathbf{p2}[\mathbf{t}]^2}{2} + \mathbf{v}[\mathbf{q1}[\mathbf{t}]], \# \right\}, \# \right), \# \right] \&, \\ \{\mathbf{p1}[\mathbf{t}], \mathbf{p2}[\mathbf{t}], \mathbf{q1}[\mathbf{t}], \mathbf{q2}[\mathbf{t}]\}$$

$$\text{Out}[13]= \{V'[\mathbf{q1}[\mathbf{t}]], 0, -\mathbf{p1}[\mathbf{t}], -\mathbf{p2}[\mathbf{t}]\}$$

The following example demonstrates that the Poisson bracket with two identical arguments, here a Hamiltonian, vanishes

$$\text{In}[14]:= \left\{\frac{\mathbf{p1}[\mathbf{t}]^2}{2} + \frac{\mathbf{p2}[\mathbf{t}]^2}{2} + \mathbf{v}[\mathbf{q1}[\mathbf{t}]], \frac{\mathbf{p1}[\mathbf{t}]^2}{2} + \frac{\mathbf{p2}[\mathbf{t}]^2}{2} + \mathbf{v}[\mathbf{q1}[\mathbf{t}]]\right\}_{\text{pb2}}$$

$$\text{Out}[14]= 0$$

So far, we defined the algebraic structure and derived certain types of Poisson manifolds by specifying the phase space variables. However, the knowledge of these manifolds allows us to go a step further to a real application and to introduce Hamilton's equations. Knowing the Poisson manifold, it is straightforward to write down these equations.

Definition of Hamilton's Equations

For a set of generalized coordinates (q_k, p_k) , Hamilton's equations are defined by

$$\dot{q}_k = \frac{\partial H}{\partial p_k} \quad (20)$$

$$\dot{p}_k = -\frac{\partial H}{\partial q_k}. \quad (21)$$

This set of equations is related to Poisson's bracket by considering a function f in the phase space depending on the phase space coordinates q_k, p_k , and t as

$$f = f(q_k, p_k, t). \quad (22)$$

The structure of the phase space is governed by Poisson's bracket which can be seen by differentiating f with respect to t

$$\frac{df}{dt} = \frac{\partial f}{\partial t} + \sum_{k=1}^p \frac{\partial f}{\partial q_k} \dot{q}_k + \frac{\partial f}{\partial p_k} \dot{p}_k. \quad (23)$$

By inserting Hamilton's equations of motion into this expression we gain

$$\frac{df}{dt} = \frac{\partial f}{\partial t} + \sum_{k=1}^{\rho} \frac{\partial f}{\partial q_k} \frac{\partial H}{\partial p_k} - \frac{\partial f}{\partial p_k} \frac{\partial H}{\partial q_k}. \quad (24)$$

Let us abbreviate by

$$\sum_{k=1}^{\rho} \frac{\partial f}{\partial q_k} \frac{\partial H}{\partial p_k} - \frac{\partial f}{\partial p_k} \frac{\partial H}{\partial q_k} = \{f, H\}_{\{q,p\}} \quad (25)$$

the Poisson's bracket defined on the phase space spanned by the coordinates q_k and p_k . The subscript $\{q, p\}$ denotes the set of variables of the phase space. Using this definition, the temporal change of f becomes

$$\frac{df}{dt} = \frac{\partial f}{\partial t} + \{f, H\}_{\{q,p\}}. \quad (26)$$

Note that this relation allows us to calculate the temporal changes of any function depending on the phase space variables.

We already encountered conserved quantities which have the property that their temporal change vanishes. This vanishing can be expressed by the Poisson bracket in a very convenient way. The conservation of the quantity f implies

$$\frac{df}{dt} = 0, \quad (27)$$

which is identical with

$$\frac{\partial f}{\partial t} + \{f, H\}_{\{q,p\}} = 0. \quad (28)$$

The previous relations define a connection between a Poisson manifold and a Hamilton system. Thus, the two properties connected by the manifold are denoted as a Poisson–Hamilton (PH) manifold. The equations of motion for any kind of function in this manifold can be derived by means of the Poisson bracket via equation (26).

It is convenient here to define a class for Hamilton's equations which inherits the properties of the Poisson bracket. The properties of the Poisson manifold are equivalent to the properties of the Hamilton manifold.

The following lines define the class `HamiltonEquations` based on equation (28).

```
In[15]:= HamiltonEquations = Class["HamiltonEquations", PB,
  {description = "Hamilton's equations"},
  {HamEqs[H_, V_ /; FreeQ[V, List]] := ∂_t V == PoissonBracket[H, V],
   HamEqs[H_, V_List] := Map[HamEqs[H, #] &, V],
   HamEqs[H_] := Map[HamEqs[H, #] &, Flatten[{P, Q}]]},
  HamiltonEquations[Pnew_List, Qnew_List, Tnew_Symbol] :=
    Block[{P = Pnew; Q = Qnew; T = Tnew}]
]

Out[15]= <Class HamiltonEquations>
```

To handle the class for Hamiltonian equations and the derived objects in the same way as in textbooks or in the case of Poisson brackets, we introduce the notation

$$\text{In}[16]:= \text{Notation}[\mathcal{H}_{Eq}^{\text{obj-}}[\mathbf{f_}] \Leftrightarrow \text{obj_} \circ \text{HamEqs}[\mathbf{f_}]]$$

and define the corresponding palette.

$$\text{In}[17]:= \begin{array}{|c|} \hline \{\square, \square\}_{\square} \\ \hline \mathcal{H}_{Eq}^{\square}[\square] \\ \hline \end{array}$$

$$\text{Out}[17]= \left\{ \left\{ \square, \square \right\}_{\square}, \left\{ \mathcal{H}_{Eq}^{\square}[\square] \right\} \right\}$$

Having these tools available, we can apply the classes to specific problems.

Hamilton's Equations

As a first example, let us examine a PH manifold with a single coordinate and a single momentum. The object defining the PH manifold is created by

$$\text{In}[18]:= \text{ham1} = \text{HamiltonEquations} \circ \text{new}[\{P \rightarrow \{p[t]\}, Q \rightarrow \{q[t]\}\}]$$

$$\text{Out}[18]= \langle \text{Object of HamiltonEquations} \rangle$$

Specifying a single particle Hamiltonian by kinetic and potential energies, we can derive the set of Hamilton's equations by applying the manifold to the Hamiltonian.

$$\text{In}[19]:= \mathcal{H}_{Eq}^{\text{ham1}} \left[\frac{p[t]^2}{2} + v[q[t]] \right]$$

$$\text{Out}[19]= \{p'[t] == v'[q[t]], q'[t] == -p[t]\}$$

The result is a system of equations defining the dynamic motion of this particle.

A second example is concerned with a four-dimensional PH manifold. The generalized coordinates and the momenta are primarily given by q_1 , q_2 , p_1 , and p_2 .

$$\text{In}[20]:= \text{ham2} = \text{HamiltonEquations} \circ \text{new}[\{P \rightarrow \{p1[t], p2[t]\}, Q \rightarrow \{q1[t], q2[t]\}\}]$$

$$\text{Out}[20]= \langle \text{Object of HamiltonEquations} \rangle$$

As an example, let us consider the double pendulum. The Hamiltonian for this system reads

$$\begin{aligned} \text{In}[21]:= \text{HamDoublePendulum} = & \\ & \frac{1}{2 l_1^2 l_2^2 m_2 (m_1 + m_2 \sin[\theta_1[t] - \theta_2[t]]^2)} (l_2^2 m_2 p_1[t]^2 + l_1^2 (m_1 + m_2) p_2[t]^2 - \\ & 2 m_2 l_1 l_2 p_1[t] p_2[t] \cos[\theta_1[t] - \theta_2[t]]) - \\ & m_2 g l_2 \cos[\theta_2[t]] - (m_1 + m_2) g l_1 \cos[\theta_1[t]] \\ \text{Out}[21]= & -g \cos[\theta_2[t]] l_2 m_2 - g \cos[\theta_1[t]] l_1 (m_1 + m_2) + \\ & \frac{l_2^2 m_2 p_1[t]^2 - 2 \cos[\theta_1[t] - \theta_2[t]] l_1 l_2 m_2 p_1[t] p_2[t] + l_1^2 (m_1 + m_2) p_2[t]^2}{2 l_1^2 l_2^2 m_2 (m_1 + \sin[\theta_1[t] - \theta_2[t]]^2 m_2)} \end{aligned}$$

where p_i ($i = 1, 2$) are the generalized momenta, l_i , m_i are the inertia momenta and the masses of the particles, and θ_i are the angles of deviation. The PH manifold `ham2` defined earlier does not exactly correspond to the variables used in the Hamiltonian. However, we are able to change the coordinate names by setting the properties of the PH manifold using

```
In[22]:= SetProperty[ham2, {P -> {p1[t], p2[t]}, Q -> {\theta1[t], \theta2[t]}}
```

Now the PH manifold is defined for the coordinates

```
In[23]:= GetPropertiesForm[ham2]
```

```
Out[23]//DisplayForm=
```

Property	Value
description	Hamilton's equations
P	{p1[t], p2[t]}
Q	{\theta1[t], \theta2[t]}
T	t

The four equations of motion can then be gained by

```
In[24]:= equationsOfMotion = NEqham2[HamDoublePendulum]
```

$$\begin{aligned} \text{Out[24]} = \left\{ \begin{aligned} p_1'[t] &= g \sin[\theta_1[t]] l_1 (m_1 + m_2) + \\ &\frac{1}{2 l_1^2 l_2^2 m_2} \left(\frac{2 \sin[\theta_1[t] - \theta_2[t]] l_1 l_2 m_2 p_1[t] p_2[t]}{m_1 + \sin[\theta_1[t] - \theta_2[t]]^2 m_2} - \right. \\ &\quad (2 \cos[\theta_1[t] - \theta_2[t]] \sin[\theta_1[t] - \theta_2[t]] m_2 (l_2^2 m_2 p_1[t]^2 - \\ &\quad 2 \cos[\theta_1[t] - \theta_2[t]] l_1 l_2 m_2 p_1[t] p_2[t] + l_1^2 (m_1 + m_2) p_2[t]^2)) / \\ &\quad \left. (m_1 + \sin[\theta_1[t] - \theta_2[t]]^2 m_2)^2 \right), p_2'[t] = g \sin[\theta_2[t]] l_2 m_2 + \\ &\frac{1}{2 l_1^2 l_2^2 m_2} \left(- \frac{2 \sin[\theta_1[t] - \theta_2[t]] l_1 l_2 m_2 p_1[t] p_2[t]}{m_1 + \sin[\theta_1[t] - \theta_2[t]]^2 m_2} + \right. \\ &\quad (2 \cos[\theta_1[t] - \theta_2[t]] \sin[\theta_1[t] - \theta_2[t]] m_2 \\ &\quad (l_2^2 m_2 p_1[t]^2 - 2 \cos[\theta_1[t] - \theta_2[t]] l_1 l_2 m_2 p_1[t] p_2[t] + \\ &\quad l_1^2 (m_1 + m_2) p_2[t]^2)) / (m_1 + \sin[\theta_1[t] - \theta_2[t]]^2 m_2)^2 \Big), \\ \theta_1'[t] &= \frac{-2 l_2^2 m_2 p_1[t] + 2 \cos[\theta_1[t] - \theta_2[t]] l_1 l_2 m_2 p_2[t]}{2 l_1^2 l_2^2 m_2 (m_1 + \sin[\theta_1[t] - \theta_2[t]]^2 m_2)}, \\ \theta_2'[t] &= \\ &\frac{2 \cos[\theta_1[t] - \theta_2[t]] l_1 l_2 m_2 p_1[t] - 2 l_1^2 (m_1 + m_2) p_2[t]}{2 l_1^2 l_2^2 m_2 (m_1 + \sin[\theta_1[t] - \theta_2[t]]^2 m_2)} \Big\} \end{aligned} \right. \end{aligned}$$

They represent the dynamics of the double pendulum in the PH manifold. This example demonstrates the fact that using an object-oriented approach, the knowledge of the Hamiltonian, and the coordinates for the manifold suffice to derive the equations of motion in a portion of a second. The ease of use and the close connection to textbook presentations allows for fast manipulation and

reliable calculation of results. Besides these examples, many other applications of *Elements* to similar subjects are ahead.

■ Conclusions

In this article we discussed basic guidelines for building a sophisticated object-oriented modeling environment based on a knowledge base containing predefined components. We presented the package *Elements* that implements the core functionality of such a system that is suitable to be used for both industrial and mathematical applications. We demonstrated that combining the computational power of computer algebra with object-oriented paradigms provides a firm foundation for speeding up the work flow in industrial modeling processes. The soundness and consistency of the stored information ensures its practical applicability in a variety of settings.

■ References

- [1] P. F. Dubois, *Object Technology for Scientific Computing: Object-Oriented Numerical Software in Eiffel and C*, Englewood Cliffs, NJ: Prentice Hall, 1997.
- [2] K. Dowd, *High Performance Computing*, A Nutshell Handbook, Sebastopol, CA: O'Reilly & Associates, Inc., 1993.
- [3] G. Buzzi-Ferraris, *Scientific C++: Building Numerical Libraries the Object-Oriented Way*, Englewood Cliffs, NJ: Prentice Hall, 1993.
- [4] D. Yang, *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*, New York: Springer-Verlag, 2000.
- [5] M. Berth, F.-M. Moser, and A. Triulzi, "Implementing Computational Services Based on OpenMath," in Ganzha et al., pp. 49–60.
- [6] M. Göbel, W. Küchlin, S. Müller, and A. Weber, "Extending a Java Based Framework for Scientific Software-Components," in *Computer Algebra in Scientific Computing (CASC'99), Proceedings of the Second International Workshop on Computer Algebra in Scientific Computing*, Munich (V. G. Ganzha, E. W. Mayr, and E. V. Vorozhtsov, eds.), New York: Springer-Verlag, 1999 pp. 207–223.
- [7] *Modelica—A Unified Object-Oriented Language for Physical Systems Modeling Language Specification Version 2.0*, Linköping, Sweden: Modelica Association, (Feb 2002) www.modelica.org.
- [8] P. Fritzson, J. Gunnarson, and M. Jirstrand, "MathModelica: An Extensible Modeling and Simulation Environment with Integrated Graphics and Literate Programming," in *The Proceedings of the Second International Modelica Conference*, DLR, Oberpfaffenhofen, Germany (M. Otter, ed.), 2002 pp. 41–54 www.mathcore.com/documents.
- [9] V. G. Ganzha, D. Chibisov, and E. V. Vorozhtsov, "GROOME—Tool Supported Graphical Object-Oriented Modeling for Computer Algebra and Scientific Computing," in *Computer Algebra in Scientific Computing (CASC 2001), Proceedings of the Fourth International Workshop on Computer Algebra in Scientific Computing*, Konstanz (V. G. Ganzha, E. W. Mayr, and E. V. Vorozhtsov, eds.), New York: Springer-Verlag, 2001 pp. 213–23.

- [10] R. Maeder, *The Mathematica Programmer*, Boston: AP Professional, 1994.
- [11] M. Mnuk and G. Baumann, "Elements: A Package for Object Oriented Programming in Mathematica," in *Computer Algebra in Scientific Computing (CASC 2002), Proceedings of the Fifth International Workshop on Computer Algebra in Scientific Computing*, Yalta, Ukraine (V. G. Ganzha, E. W. Mayr, and E. V. Vorozhtsov, eds.), Munich: Technical University, 2002 pp. 227–236.
- [12] A. Weber, G. Simon, W. Küchlin, and J. Hoss, "Lessons Learned from Using CORBA for Components in Scientific Computing," in Ganzha et al., pp. 409–422.
- [13] *Elements*, KnowledgeGeneration, Edelweiss-St. 1, D-82211 Herrsching a.A., Germany (contact Gerd.Baumann@KnowledgeGeneration.de).
- [14] V. G. Ganzha, E. W. Mayr, and E. V. Vorozhtsov, eds., *Computer Algebra in Scientific Computing (CASC 2000), Proceedings of the Third International Workshop on Computer Algebra in Scientific Computing*, Samarkand, New York: Springer-Verlag, 2000.

About the Authors

Gerd Baumann is a professor in the Mathematical Physics department at the University of Ulm. He is Head of the Mathematics Department at the German University in Cairo (GUC) and also associated with the faculty of Informatics at the Technical University of Munich where he lectures on symbolic computing. Baumann is the author of *Mathematica in Theoretical Physics* and *Symmetry Analysis of Differential Equations with Mathematica*.

Michal Mnuk is associated with the faculty of Informatics at the Technical University of Munich. He holds a Ph.D. degree in mathematics from the Research Institute for Symbolic Computation, Hagenberg, and is currently completing his Habilitation at the Technical University of Munich.

Gerd Baumann

Department of Mathematical Physics
University of Ulm
Albert-Einstein-Allee 11
D-89069 Ulm
Gerd.Baumann@KnowledgeGeneration.de

Michal Mnuk

Institute of Informatics
Technical University of Munich
Boltzmannstrasse 3
85748 Garching b. München
mnuk@scimod.de

MathModelica—An Object-Oriented Mathematical Modeling and Simulation Environment

Peter Fritzson

MathModelica is an integrated interactive development environment for advanced object-oriented system modeling and simulation. The environment integrates Modelica-based modeling and simulation with graphic design, advanced scripting facilities, integration of program code, test cases, graphics, documentation, mathematical typesetting, and symbolic formula manipulation provided via *Mathematica*. The user interface consists of a graphical Model Editor and *Mathematica* notebooks. The Model Editor is a graphical user interface in which models can be assembled using components from a number of standard libraries representing different physical domains or disciplines, such as electrical, mechanics, block-diagram and multibody systems. The accessible *MathModelica* internal form allows the user to extend the system with new functionality, as well as to perform queries on the model representation and write scripts for automatic model generation. Furthermore, extensibility of syntax and semantics provides additional flexibility in adapting to unforeseen user needs.

■ 1. Background

Traditionally, simulation and accompanying activities [1] have been expressed using heterogeneous media and tools, with a mixture of manual and computer-supported activities:

- A simulation model is traditionally designed on paper using traditional mathematical notation.
- Simulation programs are written in a low-level programming language and stored on text files.

- Input and output data, if stored at all, are saved in proprietary formats needed for particular applications and numerical libraries.
- Documentation is written on paper or in separate files that are not integrated with the program files.
- The graphical results are printed on paper or saved using proprietary formats.

When the result of the research and experiments, such as a scientific paper, is written, the user normally gathers together input data, algorithms, and output data and its visualizations, as well as notes and descriptions. One of the major problems in simulation development environments is that gathering and maintaining correct versions of all these components from various files and formats is difficult and error-prone.

Our vision of a solution to this set of problems is to provide integrated computer-supported modeling and simulation environments that enable the user to work effectively and flexibly with simulations. Users would then be able to prepare and run simulations as well as investigate simulation results. Several auxiliary activities accompany simulation experiments: requirements are specified, models are designed, documentation is associated with appropriate places in the models, and input and output data, as well as possible constraints on such data, are documented and stored together with the simulation model. The user should be able to reproduce experimental results. Therefore input data and parts of output data as well as the experimenter's notes should be stored for future analysis.

□ 1.1. Integrated Interactive Programming Environments

An integrated interactive modeling and simulation environment is a special case of programming environment with applications in modeling and simulation. Thus, it should fulfill the requirements from both general integrated programming environments and the application area of modeling and simulation mentioned in the previous section.

The main idea of an integrated programming environment in general is that a number of programming support functions should be available within the same tool in a well integrated way. This means that the functions should operate on the same data and program representations and exchange information when necessary, resulting in an environment that is both powerful and easy to use. An environment is interactive and incremental if it gives quick feedback (e.g., without recomputing everything from scratch) and maintains a dialogue with the user, including preserving the state of previous interactions with the user. Interactive environments are typically both more productive and more fun to use.

There are many things that a programming environment should accomplish for the programmer, particularly if it is interactive. What functionality should be included? Comprehensive software development environments are expected to provide support for the major development phases, such as

- requirements analysis

- design
- implementation
- maintenance

A programming environment can be somewhat more restrictive and need not necessarily support early phases such as requirements analysis, but it is an advantage if such facilities are also included. The main point is to provide as much computer support as possible for different aspects of software development and to free the developer from mundane tasks so that more time and effort can be spent on the essential issues. The following is a partial list of integrated programming environment facilities, some of which are already mentioned in [2], that should be provided for the programmer:

- Administration and configuration management of program modules and classes, and different versions of these.
- Administration and maintenance of test examples and their correct results.
- Administration and maintenance of formal or informal documentation of program parts, and automatic generation of documentation from programs.
- Support for a given top-down or bottom-up programming methodology. For example, if a top-down approach should be encouraged, it is natural for the interactive environment to maintain successive composition steps and mutual references between those.
- Support for the interactive session. For example, previous interactions should be saved in an appropriate way so that the user can refer to previous commands or results, go back and edit those, and possibly re-execute.
- Enhanced editing support, performed by an editor that knows about the syntactic structure of the language. It is advantageous if the system allows editing of the program in different views. For example, editing of the overall system structure can be done in the graphical view, whereas editing of detailed properties can be done in the textual view.
- Cross-referencing and query facilities, to help the user understand interdependences between parts of large systems.
- Flexibility and extensibility (e.g., mechanisms to extend the syntax and semantics of the programming language representation and the functionality built into the environment).
- Accessible internal representation of programs. This is often a prerequisite to the extensibility requirement. This means that there is a well-defined representation of programs as data structures in the programming language itself, so that user-written programs may inspect the structure and generate new programs. This property is also known as the principle of program-data equivalence.

□ 1.2. Vision of Integrated Interactive Environment for Modeling and Simulation

Our vision for the *MathModelica* integrated interactive environment is to fulfill essentially all the requirements for general integrated interactive environments combined with the specific needs for modeling and simulation environments, such as

- specification of requirements, expressed as documentation and/or mathematics
- design of the mathematical model
- symbolic transformations of the mathematical model
- a uniform general language for model design, mathematics, and transformations
- automatic generation of efficient simulation code
- execution of simulations
- evaluation and documentation of numerical experiments
- graphical presentation

The design and vision of *MathModelica* is to a large extent based on our earlier experience in research and development of integrated incremental programming environments (e.g., the DICE system [3] and the ObjectMath environment [4, 5]) and many years of intensive use of advanced integrated interactive environments such as the Interlisp system [2, 6, 7], and *Mathematica* [8, 9]. The Interlisp system was actually one of the first really powerful integrated environments and still exceeds most current programming environments in terms of powerful facilities available to the programmer. It was also the first environment that used graphical window systems in an effective way [10] (e.g., before the Smalltalk environment [11] and the Macintosh window system appeared).

Mathematica, which has been available since 1988, is an integrated interactive programming environment with many similarities to Interlisp, containing comprehensive programming and documentation facilities, accessible intermediate representation with program-data equivalence, graphics, and support for mathematics and computer algebra. *Mathematica* is more developed than Interlisp in several areas (e.g., syntax, documentation, and pattern matching), but less so in programming support facilities.

□ 1.3. *Mathematica*, *Modelica*, and *MathModelica*

It turns out that *Mathematica* is an integrated programming environment that fulfils many of our requirements. However, it lacks object-oriented modeling and structuring facilities as well as generation of efficient simulation code needed for effective modeling and simulation of large systems. These modeling and simulation facilities are provided by the object-oriented modeling language *Modelica* [12, 13, 14, 15, 16, 17, 18, 19].

Our solution to the problem of a comprehensive modeling and simulation environment is to combine *Mathematica* and *Modelica* into an integrated interactive environment called *MathModelica*. This environment provides an internal representation of *Modelica* that builds on and extends the standard *Mathematica* representation, which makes it well integrated with the rest of the *Mathematica* system.

The realization of the general goal of a *uniform general language* for model design, mathematics, and symbolic transformations is based on an integration of the two languages *Mathematica* and *Modelica* into an even more powerful language. This language is *Modelica* in *Mathematica* syntax, extended with a subset of *Mathematica*. Only the *Modelica* subset of *MathModelica* can be used for object-oriented modeling and simulation, whereas the *Mathematica* part of the language can be used for interactive scripting and symbolic transformations.

Mathematica provides representation of mathematics and facilities for programming symbolic transformations, whereas *Modelica* provides language elements and structuring facilities for object-oriented component based modeling, including a strong type system for efficient code and engineering safety. However, this language integration is not yet realized to its full potential in the current release of *MathModelica*, even though the current level of integration provides many impressive capabilities. Future improvements of the *MathModelica* language integration might include making the object-oriented facilities of *Modelica* also available for ordinary *Mathematica* programming, as well as making some of the *Mathematica* language constructs available within code for simulation models.

The current *MathModelica* system builds on experience from the design of the ObjectMath [4, 5] modeling language and environment, early *Modelica* prototypes [20, 21], as well as results from object-oriented modeling languages and systems such as Dymola [22, 23] and Omola [24, 25], which together with ObjectMath and a few other object-oriented modeling languages (e.g., [26, 27, 28, 29, 30]) have provided the basis for the design of *Modelica*.

ObjectMath was originally designed as an object-oriented extension of *Mathematica* augmented with efficient code generation and a graphic class browser. The ObjectMath effort was initiated in 1989 and concluded in the fall of 1996 when the Modelica Design Group was started and later renamed the Modelica Association. At that time, instead of developing a fifth version of ObjectMath, we decided to join forces with the originators of a number of other object-oriented mathematical modeling languages in creating the *Modelica* language, with the ambition of eventually making it an international standard. In many ways the *MathModelica* product can be seen as a logical successor to the ObjectMath research prototype.

■ 2. The *Mathematica*-Style *Modelica* Language for Simulation

The details of the *Mathematica*-style *Modelica* language briefly mentioned in the previous section, will be described using an example of an electric circuit model that is given in the form of *Modelica* expressions in *Mathematica* syntax. The subset of the *Modelica* language described in this section can be used in the

simulation models, *not* in general *Mathematica* programming. Note that here we only describe modeling in terms of textually programming *Modelica*. The *Math-Modelica* environment also includes a graphical modeling tool and standardized graphic model representation based on the *Modelica* language, which is briefly described in Section 3 of this article. Visual constructs in the graphical environment have a one-to-one correspondence with constructs or classes in the textual *Modelica* language.

Modelica models are built from classes. Like other object-oriented languages, a class contains variables (i.e., class attributes representing data). The main difference compared to traditional object-oriented languages is that instead of functions (methods) we use equations to specify behavior. Equations can be written explicitly, like $a=b$, or can be inherited from other classes. Equations can also be specified by the `Connect` equation construct. The equation (although it has function call syntax) `Connect[v1,v2]` expresses coupling between the variables $v1$ and $v2$. These variables are instances of connector classes and are attributes of the connected objects. This gives a flexible way of specifying topology of physical systems described in an object-oriented way using *Modelica*.

In the following sections we introduce some basic and distinctive syntactical and semantic features of *Modelica*, such as connectors, encapsulation of equations, inheritance, declaration of parameters and constants. Powerful parameterization capabilities, which are advanced features of *Modelica*, are discussed in Section 2.4.

□ 2.1. Connection Diagrams

As an introduction to the *Modelica* language we will present a model of a simple electrical circuit shown in Figure 1.

The circuit can be broken down into a set of standard connected electrical components. We have a voltage source, two resistors, an inductor, a capacitor and a ground point. Models of such standard components are available in *Modelica* class libraries.

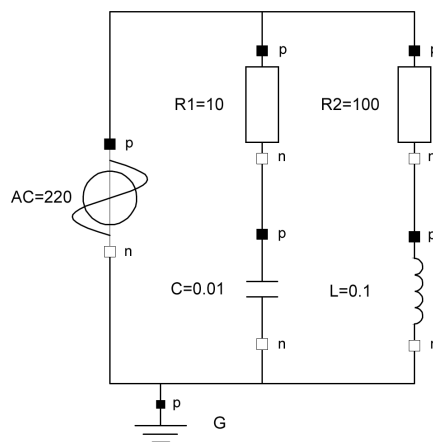


Figure 1. Connection diagram of the electric circuit.

A declaration like the following specifies R1 to be an object or instance of the class `Resistor` and sets the default value of the resistance parameter, `R`, to 10, shown in *Mathematica*-style *Modelica* syntax.

```
Resistor R1[{R==10}];
```

A *Mathematica*-style *Modelica* description of the complete circuit appears as follows.

```
Model[Circuit, (* Circuit model in Mathematica-style syntax *)
Resistor R1[{R == 10}];
Capacitor C[{C == 0.01}];
Resistor R2[{R == 100}];
Inductor L[{L == 0.1}];
VsourceAC AC;
Ground G;
Equation[
  Connect[AC.p, R1.p]; (* Capacitor circuit*)
  Connect[R1.n, C.p];
  Connect[C.n, AC.n];
  Connect[R1.p, R2.p]; (* Inductor circuit*)
  Connect[R2.n, L.p];
  Connect[L.n, C.n];
  Connect[AC.n, G.p]; (* Ground *)
]
]
```

The same Circuit model in standard *Modelica*-style syntax appears in the following. The relation between these two syntactic forms is explained in Section 3.4, including a discussion regarding our choice of *Mathematica* syntax for certain *Modelica* language features. For the rest of this article, we will primarily use *Mathematica*-style syntax for examples and *Modelica* language constructs.

```
model Circuit /* Circuit model in Modelica-style syntax */
  Resistor R1(R=10);
  Capacitor C(C=0.01);
  Resistor R2(R=100);
  Inductor L(L=0.1);
  VsourceAC AC;
  Ground G;
equation
  connect(AC.p, R1.p); // Capacitor circuit
  connect(R1.n, C.p);
  connect(C.n, AC.n);
  connect(R1.p, R2.p); // Inductor circuit
  connect(R2.n, L.p);
  connect(L.n, C.n);
  connect(AC.n, G.p); // Ground
end Circuit;
```

A composite model like the circuit model described earlier specifies the system topology (i.e., the components and the connections between the components). The connections specify interactions between the components. In some previous

object-oriented modeling languages, connectors are referred to as cuts, ports, or terminals. The `Connect` construct is a special equation form that generates standard equations taking into account what kind of interaction is involved as explained in Section 2.3.

Variables declared within classes are public by default if they are not preceded by the keyword `Protected`, which has the same semantics as in Java. Additional `Public` or `Protected` sections can appear within a class, preceded by the corresponding keyword.

□ 2.2. Type Definitions

The *Modelica* language is a strongly typed language with both predefined and user-defined types. The built-in “primitive” data types support boolean, integer, real, and string values. These primitive types contain data that *Modelica* understands directly. The type of every variable must be stated explicitly. The primitive data types of *Modelica* are listed in Table 1.

Type	Description
Boolean	Either <code>True</code> or <code>False</code>
Integer	Corresponding to the C <code>int</code> data type, usually 32-bit two’s complement
Real	Corresponding to the C <code>double</code> data type, usually 64-bit floating-point
String	String of 8-bit characters
enumeration	Enumeration literal constants of the corresponding enumeration type

Table 1. Predefined data types in *Modelica*.

It is possible to define new user-defined types.

```
Type[name, annotation, type]
```

Here is an example defining a temperature measured in Kelvin, `K`, which is of type `Real` with the minimum value zero.

```
Type[Temperature,  
      "temperature measured in Kelvin",  
      Real[{Unit == "K", Min == 0}]]
```

Here the user-defined types of `Voltage` and `Current` are defined.

```
Type[Voltage, Real[{Unit == "V"}]]
```

This defines the symbol `Voltage` to be a specialization of the type `Real` that is a basic predefined type. Each type (including the basic types) has a collection of default attributes such as unit of measure, initial value, minimum, and maximum value. These default attributes can be changed when declaring a new type. In the previous case, the unit of measure of `Voltage` is changed to "V". Here is a corresponding definition made for `Current`.

```
Type[Current, Real[{Unit == "A"}]]
```

In *Modelica*, the basic structuring element is a class. The general keyword `Class` is used for declaring classes. There are also seven restricted class categories with specific keywords, such as `Type` (a class that is an extension of built-in classes, such as `Real`, or of other defined types) and `Connector` (a class that does not have equations and can be used in connections). For a *valid model*, replacing the `Type` and `Connector` keywords by the keyword `Class` still keeps the model semantically equivalent to the original because the restrictions imposed by such a specialized class are already fulfilled by a valid model. Other specific class categories are `Model`, `Record`, and `InOutBlock`. Moreover, functions and packages are regarded as special kinds of restricted and enhanced classes, denoted by the keywords `ModelicaFunction` for functions and `Package` for packages.

The idea of restricted classes is advantageous because the modeler does not have to learn several different concepts, but just one—the *class* concept. All basic properties of a class, such as syntax and semantics of definition, instantiation, inheritance, and generic properties are identical to all kinds of restricted classes. Furthermore, the construction of *MathModelica* translators is simplified considerably because only the syntax and semantics of the *class* construct have to be implemented along with some additional checks on restricted classes. The basic types, such as `Real` or `Integer`, are built-in type classes (i.e., they have all the properties of a class). The previous definitions have been expressed using the keyword `Type`, which is equivalent to `Class`, but limits the defined type to be an extension of a built-in type, a record type, or an array type. Note however that the restricted classes that are packages and functions have some special properties that are not present in general classes.

□ 2.3. Connector Classes

When developing models and model libraries for a new application domain, it is good to start by defining a set of connector classes that are used as templates for interfaces between model instances. A common set of connector classes used by all models in the library supports compatibility and connectability of the component models.

Pin

The following is a definition of an electrical connector class `Pin`, used as an interface class for electrical components. The voltage, v , is defined as a “potential” nonflow variable, and the current, i , as a flow variable by being prefixed by the keyword `Flow`. This implies that voltages will be set equal when two or more components are connected (i.e., $v_1 = v_2 = \dots = v_n$) and currents are summed to zero at the connection point, $i_1 + i_2 + \dots + i_n = 0$.

```
Connector[Pin,
  Voltage v;
  Flow Current i
]
```

`Connect` equations are used to connect instances of connector classes. A `Connect` equation `Connect[Pin1,Pin2]`, with the instances `Pin1` and `Pin2` of connector

class `Pin`, connects the two pins so that they form one node (in this case one electrical connection). This implies two standard equations, namely:

```
Pin1.v == Pin2.v
Pin1.i + Pin2.i == 0
```

The first equation says that the voltages of the connected wire ends are the same (i.e., $v_1 = v_2 = \dots = v_n$). The second equation corresponds to Kirchhoff's current law saying that the currents sum to zero at a connection point (assuming positive value while flowing into the component), $i_1 + i_2 + \dots + i_n = 0$. The sum-to-zero equations are generated when the prefix `Flow` is used in the declaration. Similar laws apply to flow rates in a piping network and to forces and torques in mechanical systems.

□ 2.4. Partial (Abstract) Classes

A useful strategy for reuse in object-oriented modeling is to try to capture common properties in superclasses that can be inherited by more specialized classes. For example, a common property of many electrical components such as resistors, capacitors, inductors, and voltage sources, is that they have *two pins*. This means that it is useful to define a generic “template” class, or superclass, that captures the properties of all electric components with two pins. This class is *partial* (i.e., *abstract* in standard object-oriented terminology) since it does not specify all properties needed to instantiate the class.

```
Partial
Model[TwoPin, "Superclass of elements with two electrical pins",
  Pin {p, n};
  Voltage v;
  Current i;
  Equation[
    v == p.v - n.v;
    0 == p.i + n.i;
    i == p.i
  ]
]
```

The class (or model) `TwoPin` has two pins, `p` and `n`, a quantity, `v`, that defines the voltage drop across the component, and a quantity, `i`, that defines the current into the pin `p`, through the component and out from the pin `n`. This can be summarized in the following points.

- Classes that inherit `TwoPin` have at least two pins, `p` and `n`.
- The voltage, `v`, is calculated as the potential at pin `p` minus the potential at pin `n` (i.e., $v = p.v - n.v$).
- The current at the negative pin of a component equals the current at the positive pin, only with a different sign (i.e., $p.i + n.i = 0$).
- The current, `i`, through a component is defined as the current at the positive pin (i.e., $i = p.i$).

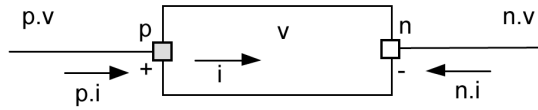


Figure 2. Structure of a `TwoPin` class with two pins.

The equations define generic relations between quantities of a simple electrical component. In order to be useful a constitutive equation must be added. The keyword `Partial` indicates that this model class is incomplete. The keyword is optional and is meant as an indication to a user that it is not possible to use the class as it is to instantiate components.

The string after the class name is a comment that is a part of the language (i.e., these comments are associated with the definition and are normally displayed by dialogues and forms presenting details about class definitions).

□ 2.5. Equations and Acausal Modeling

Acausal modeling means modeling based on equations instead of assignment statements. Equations do not specify which variables are inputs and which are outputs, whereas in assignment statements variables on the left-hand side are always outputs (results) and variables on the right-hand side are always inputs. Thus, the causality of equation-based models is unspecified and fixed only when the equation systems are solved. This is called acausal modeling.

The main advantage of acausal modeling is that the solution direction of equations will adapt to the data flow context in which the solution is computed. The data flow context is defined by specifying which variables are needed as outputs and which are external inputs to the simulated system.

The acausality of *MathModelica* (*Modelica*) library classes makes these more reusable than traditional classes containing assignment statements where the input-output causality is fixed.

Consider for example the constitutive equation from the following `Resistor` class.

$$R \cdot i == v$$

This equation can be used in three ways. The variable v can be computed as a function of i , the variable i can be computed as a function of v , or the resistance R can be computed as a function of v and i , as shown in these three statements.

$$\begin{aligned} i &= v/R \\ v &= R \cdot i \\ R &= v/i \end{aligned}$$

In the same way, consider the following equation from the class `TwoPin`.

$$v == p.v - n.v$$

This equation gives rise to one of the following three assignment statements, depending on the data flow context where the equation appears.

```
v = p.v - n.v
p.v = v + n.v
n.v = p.v - v
```

□ 2.6. Inheritance, Parameters, and Constants

We will use the following Resistor example to explain *inheritance*, *parameters*, and *constants*.

The Resistor inherits TwoPin using the Extends construct (inspired from extends in Java). A model parameter, R, is defined for the resistance and is used to state the constitutive equation for an ideal resistor, namely *Ohm's Law*: $v = R * i$. We add a definition of a parameter for the resistance and Ohm's law to define the behavior of the Resistor class in addition to what is inherited from TwoPin.

```
Model [Resistor, "Ideal electrical resistor",
  Extends [TwoPin];
  Parameter Real R [{Unit == "ohm"}] "Resistance";
  Equation[
    R * i == v
  ]
]
```

The keyword Parameter specifies that the variable is constant during a simulation run, but can change values between runs. This means that a model parameter is a special kind of constant, which is implemented as a static variable that is initialized once and never changes its value during a specific execution. A parameter is a variable that makes it simple for a user to modify the behavior of a model by changing the parameter value. There are also *Modelica* constants that never change and can be substituted inline, which are specified by the keyword Constant. Here are additional examples of constants and parameters with default values defined via so-called declaration equations that appear in declarations.

```
Constant Real c0 == 2.99792458 108;
Constant String redcolor == "red";
Constant Integer population == 1234;
Parameter Real speed == 25;
```

There are several predefined constants in the Modelica.Constants package (e.g., Planck, Boltzmann, and molar gas constants). In contrast to constants, parameters can be defined later via input to a model. Thus a parameter can be declared *without* a declaration equation. For example:

```
Parameter Real {mass, velocity};
```

The keyword Extends specifies inheritance from a parent class. All variables, equations, and connects are inherited from the parent. Multiple inheritance is supported in *Modelica*.

Just like in C++, the parent class cannot be replaced in a subclass. In *Modelica* similar restrictions also apply to equations and connections.

In C++ and Java a virtual function can be replaced/specialized by a function with the same name in the child class. In *Modelica* 2.1 equations in an `Equation` section cannot be directly named (but indirectly using a local class for grouping a set of equations) and therefore we cannot directly replace equations. When classes are inherited into a class, equations from those classes are copied into the class. This makes the equation-based semantics of the child classes consistent with the semantics of the parent class since the equation constraints of the parent class are fulfilled.

□ 2.7. Time and Model Dynamics

Models of dynamic systems are models where behavior evolves as a function of time. We use a predefined *Modelica* variable `time`, which steps forward during system simulation.

The following classes defined for electric voltage sources, capacitors, and inductors have dynamic time-dependent behavior and can also reuse the `TwoPin` superclass. In the differential equations in the classes `Capacitor` and `Inductor`, the forms v' and i' denote the time derivatives of v and i , respectively.

During system simulation the variables i and v evolve as functions of time. The differential equations solver will compute the values of $i(t)$ and $v(t)$ (t is time) so that $C v'(t) = i(t)$ for all values of t .

VsourceAC

A class for the voltage source can be defined as follows. This `VsourceAC` class inherits `TwoPin` since it is an electric component with two connector attributes, `n` and `p`. A parameter, `VA`, is defined for the amplitude, and a parameter, `f`, for the frequency. Both are given default values, 220 V and 50 Hz, respectively, that can easily be modified by the user when running simulations (e.g., through the graphical user interface). A constant `PI` is also declared using the value for π defined in the *Modelica* Standard Library, just to demonstrate the declaration of a constant. The input voltage v is defined by $v = VA * \sin(2 \pi f \text{ time})$. Note that `time` is a built-in *Modelica* primitive.

```
Model[VsourceAC, "Sine-wave voltage source",
  Extends[TwoPin];
  Parameter Real VA == 220 "Amplitude [V]";
  Parameter Real f == 50 "Frequency [Hz]";
  Protected[
    Constant Real PI == 3.141592
  ];
  Equation[
    v == VA sin[2 PI f time]
  ]
]
```

Capacitor

The `Capacitor` inherits `TwoPin` using `Extends`. A parameter, `C`, is defined for the capacitance and is used to state the constitutive equation for an ideal capacitor; namely, $\frac{dv}{dt} = \frac{i}{C}$.

```
Model[Capacitor, "Ideal electrical capacitor",
  Extends[TwoPin];
  Parameter Real C[{Unit == "F"}] "Capacitance";
  Equation[
    v' ==  $\frac{i}{C}$ 
  ]
]
```

Inductor

The `Inductor` inherits `TwoPin` using `Extends`. A parameter, `L`, is defined for the inductance and is used to state the constitutive equation for an ideal inductor; namely, $L * \frac{di}{dt} = v$.

```
Model[Inductor, "Ideal electrical inductor",
  Extends[TwoPin];
  Parameter Real L[{Unit == "H"}] "Inductance";
  Equation[
    L i' == v
  ]
]
```

Ground

Finally, we define a `Ground` class, which in the circuit model is instantiated as a ground point that serves as a reference value for the voltage levels.

```
Model[Ground, "Ground",
  Pin p;
  Equation[
    p.v == 0
  ]
]
```

□ 2.8. Definition and Simulation of the Complete Circuit Model

After all the component classes have been defined, it is possible to construct a circuit. First the *components* are *declared*, then the parameter *values* are *set*, and finally the components are *connected* using `Connect`.

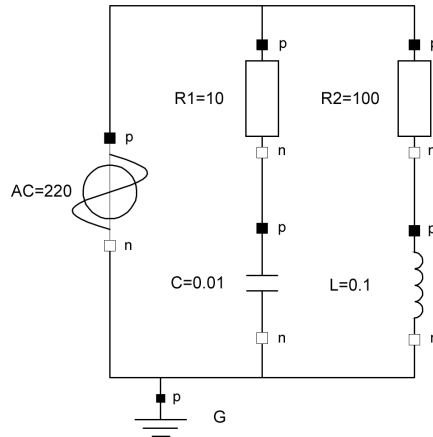


Figure 3. Connection diagram of the electric circuit (repeat of Figure 1).

Here we reproduce the `Circuit` model.

```
Model[Circuit,
  Resistor R1[{R == 10}];
  Capacitor C[{C == 0.01}];
  Resistor R2[{R == 100}];
  Inductor L[{L == 0.1}];
  VsourceAC AC;
  Ground G;
  Equation[
    Connect[AC.p, R1.p];
    Connect[R1.n, C.p];
    Connect[C.n, AC.n];
    Connect[R1.p, R2.p];
    Connect[R2.n, L.p];
    Connect[L.n, C.n];
    Connect[AC.n, G.p]
  ]
]
```

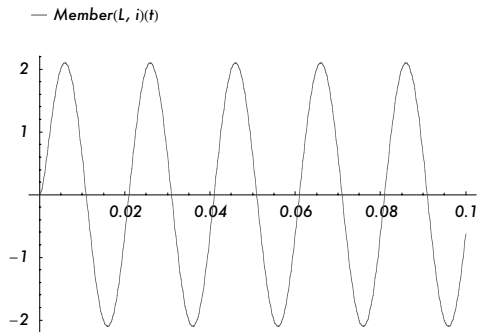
Simulation

We simulate the model with the default initial values and parameter settings in the range $0 \leq t \leq 0.1$. The status bar in the lower-left corner of the notebook shows the status of the simulation. Since this is the first time we simulate the circuit model, `Simulate` will generate C code and compile the code before the simulation.

```
Simulate[Circuit, {t, 0, 0.1}];
```

Let us plot the current in the inductor for the first 0.1 seconds.

```
PlotSimulation[{L.i[t]}, {t, 0, 0.1}]
```

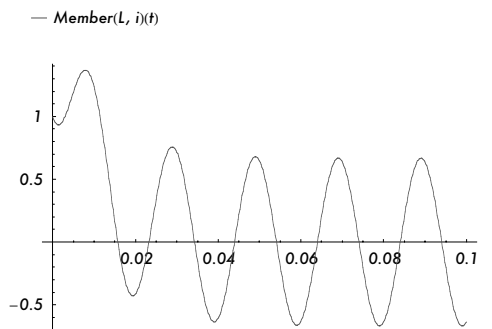


Note that the current starts at 0 Ampere, which is the default initial value. Let us change the initial values for the inductor current and the inductance using the options `InitialValues` and `ParameterValues`, respectively. This time `Simulate` will use the compiled code from the previous simulation as we have only changed initial and parameter values, and not the structure of the problem.

```
Simulate[Circuit, {t, 0, 0.1},
  InitialValues → {L.i == 1}, ParameterValues → {L.L == 1}];
```

Here is a plot that shows the result. Note the differences in initial current and amplitude due to the changed inductance.

```
PlotSimulation[{L.i[t]}, {t, 0, 0.1}]
```



□ 2.9. The *MathModelica* Notion of Subtypes

The notion of subtyping in *Modelica* is influenced by a type theory of Abadi and Cardelli [31]. The notion of inheritance in *Modelica* is independent of the notion of subtyping. According to the definition, a class A is a subtype of a class B *if and only if* the class A contains all the public variables declared in the class B, and the

types of these variables are subtypes of types of corresponding variables in B. The main benefit of this definition is additional flexibility in the definition and usage of types. For instance, the class `TempResistor` is a subtype of `Resistor`, without being a subclass of `Resistor`.

```
Model [TempResistor,
  Extends [TwoPin];
  Parameter Real {R, RT, Tref};
  Real T;
  Equation[
    v == i (R + RT * (T - Tref));
  ]
]
```

Subtyping compatibility is checked, for example, in class instantiation, redeclarations, and function calls. If a variable `a` is of type `A`, and `A` is a subtype of `B`, then `a` can be initialized by a variable of type `B`. Redeclaration is a way of modifying inherited classes, as discussed in the next section.

Note that `TempResistor` does not inherit the `Resistor` class. There are different definitions for the evaluation of `v`. If equations are inherited from `Resistor`, then the set of equations will become inconsistent in `TempResistor`, since there would be two definitions of `v`. For example, the following specialized equation from `TempResistor`:

$$v == i * (R + RT * (T - Tref))$$

and the general equation from class `Resistor`:

$$v == R * i$$

are incompatible. *MathModelica* currently does not support explicitly named equations and replacement of equations, except for the cases when the equations are collected into local class, or when a declaration equation occurs as part of a variable declaration.

□ 2.10. Class Parameterization

A distinctive feature of object-oriented programming languages and environments is the ability to reuse classes from standard libraries for particular needs. Obviously, this should be done without modification of the library code. The two main mechanisms that serve for this purpose are:

- *Inheritance*. This is essentially “copying” class definitions and adding additional elements (variables, equations, and functions) to the inheriting class.
- *Class parametrization* (also called generic classes or types). This mechanism replaces a generic type identifier in a whole class definition by an actual type.

In *Modelica* we can use *redeclaration* to control class parametrization. Assume that a library class is defined as follows.

```

Model[SimpleCircuit,
  Resistor {R1[{R == 100}], R2[{R == 200}], R3[{R == 300}]};
  Equation[
    Connect[R1.p, R2.p];
    Connect[R1.p, R3.p]
  ]
]

```

Assume also that in our particular application we would like to reuse the definition of `SimpleCircuit`: we want to use the parameter values given for `R1.R` and `R2.R` and the circuit topology, but exchange `Resistor` for the previously mentioned temperature-dependent resistor model, `TempResistor`.

This can be accomplished by redeclaring `R1` and `R2` as in the following type definition that defines `RedefinedSimpleCircuit` to be a special variant of `SimpleCircuit`.

```

Type[RedefinedSimpleCircuit,
  SimpleCircuit[{
    Redeclare[TempResistor R1],
    Redeclare[TempResistor R2]
  }]
]

```

Since `TempResistor` is a subtype of `Resistor`, it is possible to replace the *ideal* resistor model by a more specific *temperature-dependent* model. Values of the additional parameters of `TempResistor` can also be added in the redeclaration:

```

Redeclare[TempResistor R1[{RT==0.1, Tref==20.0}]]

```

Replacing `Resistor` by `TempResistor` is a very strong modification. However, it should be noted that all equations that are defined in the previous `Circuit` example model are still valid.

□ 2.11. Discrete and Hybrid Modeling

Macroscopic physical systems in general evolve continuously as a function of time, obeying the laws of physics. This includes the movements of parts in mechanical systems, current and voltage levels in electrical systems, chemical reactions, etc. Such systems are said to have *continuous-time* dynamics.

On the other hand, it is sometimes beneficial to make the approximation that certain system components display *discrete-time* behavior (i.e., changes of values of system variables over time may occur instantaneously and discontinuously). In a real physical system the change can be very fast, but not instantaneous. Examples are collisions in mechanical systems (e.g., a bouncing ball that almost instantaneously changes direction), switches in electrical circuits with quickly changing voltage levels, valves and pumps in chemical plants, etc. The reason for making the discrete approximation is to simplify the mathematical model of the system, making the model more tractable and usually speeding up the simulation of the model by several orders of magnitude.

Since the discrete approximation can only be applied to certain subsystems, we often arrive at system models consisting of interacting continuous and discrete components. Such a system is called a *hybrid system* and the associated modeling techniques *hybrid modeling*. The introduction of hybrid mathematical models creates new difficulties for their solution, but the disadvantages are far outweighed by the advantages.

Modelica provides two kinds of constructs for expressing hybrid models—*conditional expressions* and *conditional equations*—to describe discontinuous and conditional models. *When-equations* are a particular kind of conditional equation, here instantaneous equations, that express equations that are only valid at instants in time—at discontinuities—when certain conditions become true. *If[cond, then-part, else-part]* is the *Modelica* form of conditional expressions that allows modeling of phenomena with different expressions in different operating regions, as seen in the following equation describing a limiter.

$$y == \text{If}[u > \text{limit}, \text{limit}, u]$$

A more complete example of a conditional model is the model of an ideal diode. The characteristic of a real physical diode is depicted in Figure 4, and the ideal diode characteristic in parameterized form is shown in Figure 5.

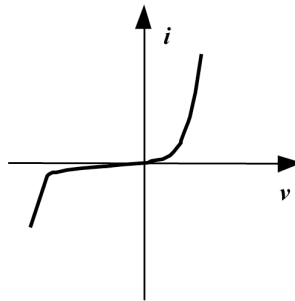


Figure 4. Real diode characteristic.

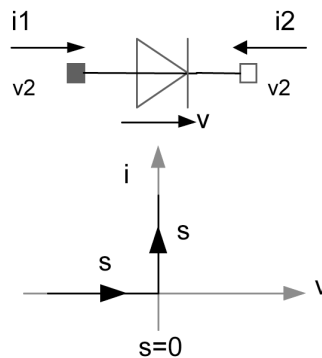


Figure 5. Ideal diode characteristic.

Since the voltage level of the ideal diode would go to infinity in an ordinary voltage-current diagram, a parameterized description is more appropriate, where both the voltage u and the current i , here the same as $i1$, are functions of the parameter s . When the diode is *off*, no current flows and the voltage is negative; whereas, when it is *on*, there is no voltage drop over the diode and the current flows.

```

Model[Diode, "Ideal diode",
  Extends[TwoPin];
  Real s;
  Boolean off;
  Equation[
    off == s < 0;
    If[off, (* two conditional equations *)
      u == s,
      u == 0
    ];
    i == If[off, 0, s] (* conditional expression *)
  ]
]

```

When-equations have been introduced in *MathModelica* to express instantaneous equations (i.e., equations that are valid only at certain points (e.g., at discontinuities)) when specific conditions become **True**. The syntax of **When**-equations for the case of a vector of conditions is shown as follows. The equations in the **When**-equation are activated when at least one of the conditions become **True**. A single condition is also possible.

```

When[ {condition1, condition2, []},
  <equations>
]

```

A bouncing ball is a good example of a hybrid system for which the **When**-equation is appropriate when modeled. The motion of the ball is characterized by the variable **height** above the ground and the vertical **velocity**. The ball moves continuously between bounces, whereas discrete changes occur at bounce times, as depicted in Figure 6. When the ball bounces against the ground, its velocity is reversed. An ideal ball would have an elasticity coefficient of 1 and would not lose any energy at a bounce. A more realistic ball, as the following modeled one, has an elasticity coefficient of 0.9, making it keep 90 percent of its speed after the bounce.

```

Model[BouncingBall, "Simple model of a bouncing ball",
  Constant Real g == 9.81 "Gravity constant";
  Parameter Real c == 0.9 "Coefficient of restitution";
  Parameter Real radius == 0.1 "Radius of the ball";
  Real height[{Start == 1}] "height of the ball center";
  Real velocity[{Start == 0}] "Velocity of the ball";
  Equation[
    height' == velocity;
    velocity' == -g;
    When[height ≤ radius,
      Reinit[velocity, -c Pre[velocity]]
    ]
  ]
]

```

The bouncing ball model contains the two basic equations of motion relating height and velocity as well as the acceleration caused by the gravitational force. At the bounce instant, the velocity is suddenly reversed and slightly decreased (i.e., $\text{velocity (after bounce)} = -c \cdot \text{velocity (before bounce)}$) which is accomplished by the special syntactic form of instantaneous equation: `Reinit[velocity, -c*Pre(velocity)]`.

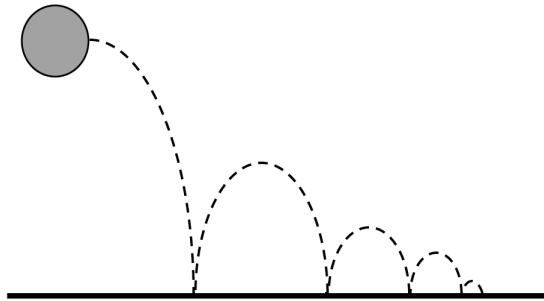


Figure 6. A bouncing ball.

Example simulations of the bouncing ball model can be found in Section 4.

□ 2.12. Discrete Events

In the previous section on hybrid modeling we briefly mentioned the notion of *discrete events*. But what is an *event*? Using everyday language an event is simply *something that happens*. This is also true for events in the abstract mathematical sense. An event in the real world (e.g., a music performance) is always associated with a point in time. However, abstract mathematical events are not always associated with time but they are usually ordered (i.e., an event ordering is defined). By associating an event with a point in time, as in Figure 7, we will automatically obtain an ordering of events to form an event history. Since this is also the case for events in the real world, we will in the following always associate

a point in *time* to each event. However, such an ordering is *partial* since several events can occur at the same point in time. To achieve a total ordering, we can use causal relationships between events, priorities of events, or, if these are not enough, simply pick an order based on some other event property.

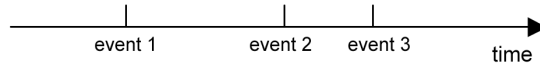


Figure 7. Events are ordered in time and form an event history.

The next question is whether the notion of event is a useful and desirable abstraction, i.e., do events fit into our overall goal of providing an object-oriented declarative formalism for modeling the world? There is no question that events, such as a cocktail party event, a car collision event, or a voltage transition event in an electrical circuit, actually exist. A set of events without structure can be viewed as a rather low-level abstraction—an unstructured mass of small low-level items that just happen.

The trick to arriving at declarative models about *what is*, rather than imperative recipes of how things *are done*, is to focus on *relations* between events, and between events and other abstractions. Relations between events can be expressed using declarative formalisms such as equations. The object-oriented modeling machinery provided by *Modelica* can be used to bring a high-level model structure and grouping of state variables affected by events, relations between events, conditions for events, and behavior in the form of equations associated with events. This brings order into what otherwise could become a chaotic mess of low-level items.

Our abstract “mathematical” notion of an event is an approximation compared to real events. For example, events in *Modelica* take *no time*—this is the most important abstraction of the *synchronous* principle to be described later. This abstraction is not completely correct with respect to our cocktail party event example since there is no question that a cocktail party actually takes some time. However, experience has shown that abstract events that take no time are more useful as a modeling primitive than events that have duration. Instead, our cocktail party should be described as a model class containing state variables, such as the number of guests that are related by equations active at primitive events like opening the party, the arrival of a guest, ending the party, serving the drinks, etc.

To conclude, an event in *Modelica* is something that happens that has the following four properties.

- A *point* in time that is instantaneous, i.e., has zero duration.
- An *event condition* that switches from `False` to `True` for the event to happen.
- A set of *variables* that are associated with the event, i.e., are referenced or explicitly changed by equations associated with the event.

- Some *behavior* associated with the event, expressed as *conditional equations* that become active or are deactivated at the event. *Instantaneous equations* are a special case of conditional equations that are only active at events.

Discrete-Time and Continuous-Time Variables

The so called *discrete-time* variables in *Modelica* only change value at discrete points in time (i.e., at event instants) and keep their values constant between events. This is in contrast to *continuous-time* variables, which may change value at any time, and usually evolve continuously over time. Figure 8 shows graphs of two variables, one continuous-time and one discrete-time.

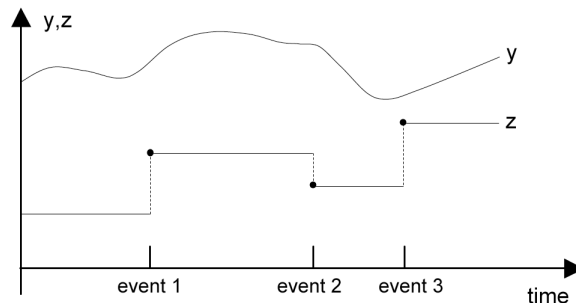


Figure 8. Example graphs of continuous-time and discrete-time variables.

Note that discrete-time variables change their values at an event instant by solving the equations active at the event. The previous value of a variable (i.e., the value before the event) can be obtained via the `Pre` function.

Variables in *Modelica* are discrete-time if they are declared using the `Discrete` prefix (e.g., `Discrete Real y`) or if they are of type `Boolean`, `Integer`, or `String`, or of types constructed from discrete types. A variable being on the left-hand side of an equation in a `When`-equation is also discrete-time. A `Real` variable *not* fulfilling the conditions for discrete-time is continuous-time. It is not possible to have continuous-time `Boolean`, `Integer`, or `String` variables.

■ 3. The MathModelica Integrated Interactive Environment

The *MathModelica* system consists of three major subsystems that are used during different phases of the modeling and simulation process, as depicted in Figure 9.

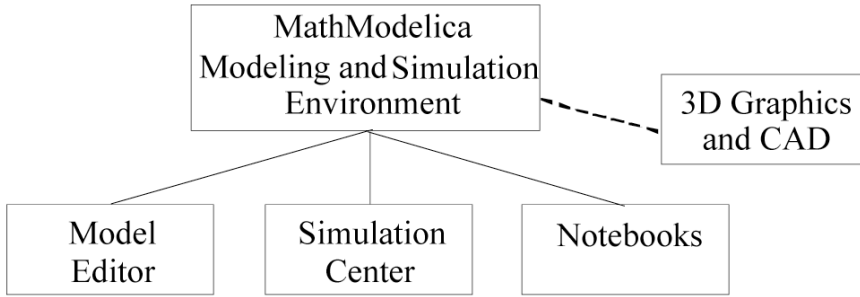


Figure 9. The *MathModelica* system architecture.

These subsystems are the following.

- The graphic *Model Editor* used for designing models from library components.
- The interactive *Notebook* facility used for literate programming, documentation, running simulations, scripting, graphics, and symbolic mathematics with *Mathematica*.
- The *Simulation Center* used for specifying parameters, running simulations, and plotting curves.

A menu palette enables the user to select whether to use the notebook interface for editing and simulations, or the Model Editor combined with the Simulation Center graphical user interface.

Additionally, *MathModelica* is loosely coupled to two optional subsystems for *3D graphics* visualization and automatic translation of *CAD* models to *Modelica*. In order to provide the best possible facilities available on the market for the user, *MathModelica* also integrates and extends several professional software products that are included in the three subsystems. For example, one version of the simulation kernel includes simulation algorithms from Dynasim [23, 32], and the notebook facility includes the technical *Mathematica* computing system [9].

A key aspect of *MathModelica* is that the modeling and simulation is done within an environment that also provides a variety of technical computations. This can be utilized both in a preprocessing stage in the development of models for subsystems as well as for postprocessing of simulation results such as signal processing and further analysis of simulated data.

□ 3.1. Graphic Model Editor

The *MathModelica* Model Editor is a graphical user interface for model diagram construction by “drag-and-drop” of model classes from the Modelica Standard Library or from user-defined component libraries, visually represented as graphic icons in the editor. A screen shot of the Model Editor is shown in

Figure 10. In the left part of the window three library packages have been opened, visually represented as overlapping windows containing graphic icons. The user can drag models from these windows and drop them on the drawing area in the middle of the tool.

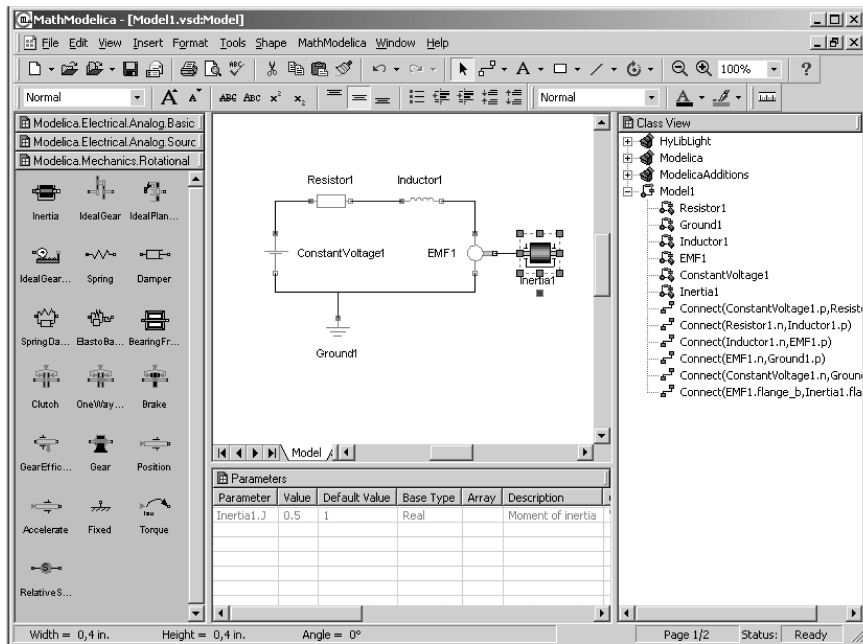


Figure 10. The graphic Model Editor showing an electrical motor with the Inertia parameter J modified.

The Model Editor can be viewed as a user interface for graphical programming in *Modelica*. Its basic functionality consists of selecting components from libraries, connecting components in model diagrams, and entering parameter values for different components.

For large and complex models it is important to be able to intuitively navigate quickly through component hierarchies. The Model Editor supports such navigation in several ways. A model diagram can be browsed and zoomed.

The Model Editor is well integrated with notebooks. A model diagram stored in a notebook is a tree-structured graphical representation of the *Modelica* code of the model, which can be converted into textual form by a command.

□ 3.2. Simulation Center

The simulation center is a subsystem with a graphical user interface for running simulations, setting initial values and model parameters, plot results, etc. These facilities are accessible via a graphic user interface accessible through the simula-

tion window (see Figure 11). However, remember that it is also possible to run simulations from the textual user interface available in the notebooks. The simulation window currently (*MathModelica* in 2005) consists of five areas or subwindows with different functionality:

- The uppermost part of the simulation window is a control panel for starting and running simulations. It contains two fields for setting start and stop times for simulation, followed by Build, Run Simulation, Plot, and Stop buttons.
- The left subwindow in the middle section shows a tree-structure view of the model selected and compiled for simulation, including all its submodels and variables. Here variables can be selected for plotting.
- The center subwindow is used for diagrams of plotted variables.
- The right subwindow in the middle section contains the legend for the plotted diagram (i.e., the names of the plotted variables).
- The subwindow at the bottom is divided into three sections: Parameters, Variables, and Messages, of which only one at a time is visible. The Parameters section, shown in Figure 11, allows changing parameter values, whereas the Variables section allows modifying initial (start) values, and the Message section allows viewing possible messages from the simulation process.

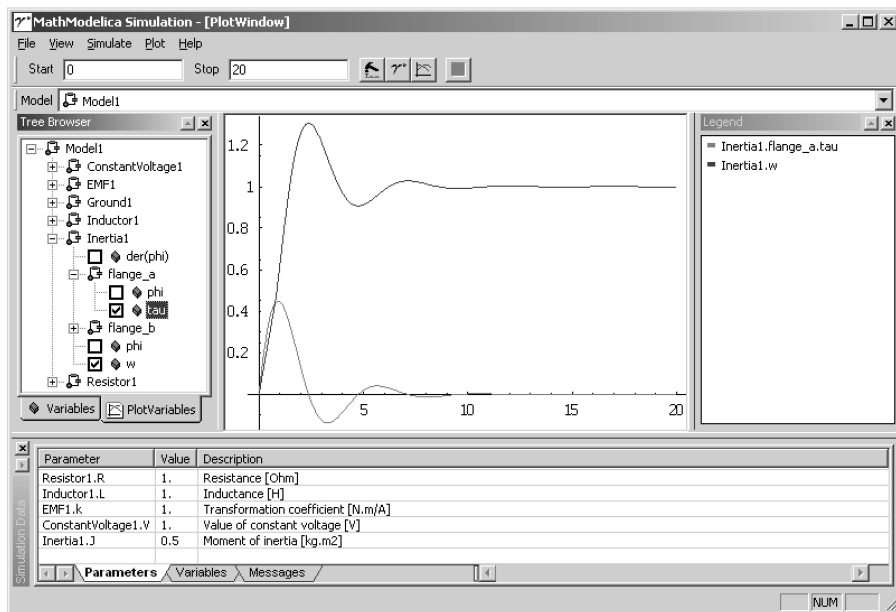


Figure 11. The simulation window with plots of the signals `Inertia1.flange_a.tau` and `Inertia1.w` selected in the menus.

If a model parameter or initial value has been changed, it is possible to rerun the simulation without rebuilding the executable code if the changed parameter does not influence the equation structure. Structure changing parameters are sometimes called *structural parameters*.

□ 3.3. Interactive Notebooks with Literate Programming

In addition to purely graphical programming of models using the Model Editor, *MathModelica* also provides a text-based programming environment for building textual models using *Modelica*. This is done using *Mathematica* notebooks, which are documents that may contain technical computations and text as well as graphics. Hence, these documents are suitable to be used for simulation scripting, model documentation and storage, model analysis and control system design, etc. In fact, this article is written as such a notebook and in the live version the examples can be run interactively. A number of sample notebooks are shown in Figure 12.

The *Mathematica* notebook facility is actually an interactive What-You-See-Is-What-You-Get (WYSIWYG) realization of *Literate Programming*, a form of programming where programs are integrated with documentation in the same document, originally proposed in [33]. A noninteractive prototype implementation of Literate Programming in combination with the document processing system L^AT_EX has been realized [34]. However, *MathModelica* is one of very few interactive WYSIWYG systems so far realized for Literate Programming and to our knowledge the only one yet for Literate Programming in modeling, which also might be called *Literate Modeling*.

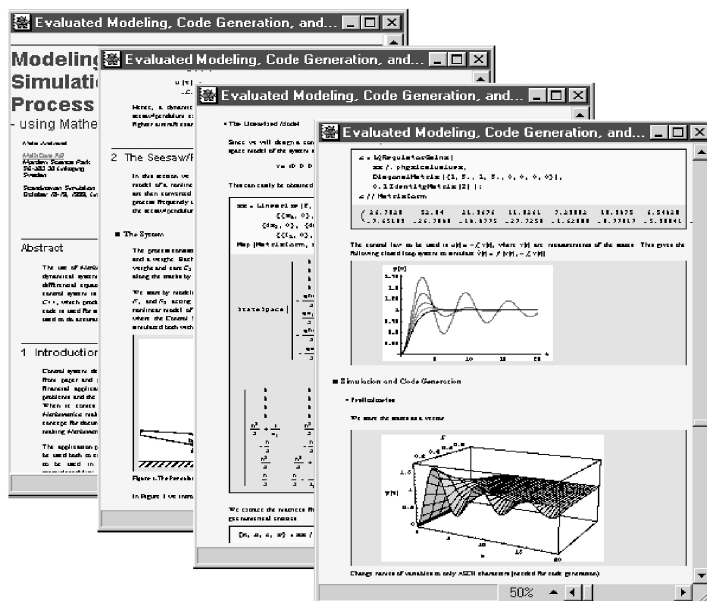


Figure 12. Examples of *MathModelica* notebooks.

Integrating *Mathematica* with *MathModelica* does not only give access to the notebook interface but also to thousands of available functions and many application packages, as well as the ability of communicating with other programs and import and export data in different formats. These capabilities make *MathModelica* more of a complete workbench for the innovative engineer than just a modeling and simulation tool. Once a model has been developed, there is often a need for further analysis such as linearization, sensitivity analysis, transfer function computations, control system design, parametric studies, Monte Carlo simulations, etc.

In fact, the combination of the ability of making user defined libraries of reusable components in *Modelica* and the notebook concept of living technical documents provides an integrated approach to model and documentation management for the evolution of models of large systems.

Tree-Structured Hierarchical Document Representation

Traditional documents (e.g., books and reports) essentially always have a hierarchical structure. They are divided into sections, subsections, paragraphs, etc. Both the document itself and its sections usually have headings as labels for easier navigation. This kind of structure is also reflected in *Mathematica* notebooks used in the *MathModelica* system.

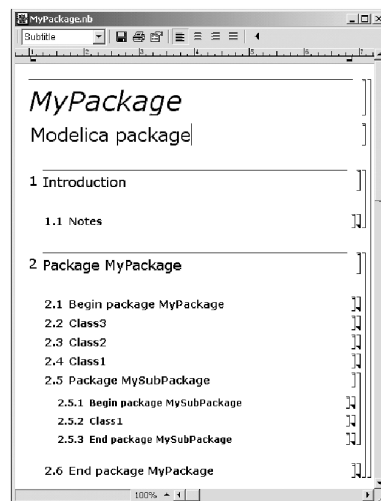


Figure 13. The MyPackage package in a notebook.

In the *MathModelica* system, *Modelica* packages including documentation and test cases are primarily stored as notebooks as in Figure 13. Those cells that contain *Modelica* model classes intended to be used from other models, e.g. library components or certain application models, should be marked as export cells. This means that when the notebook is saved, such cells are automatically exported into a *Modelica* package file in the standard *Modelica* textual representation (.mo file) that can be processed by any *Modelica* compiler and imported into

other models. For example, when saving the notebook MyPackage.nb of Figure 13, a file MyPackage.mo would be created with the following textual contents in *Modelica* syntax.

```

package MyPackage
  model Class3
  ...
end Class3;
  model Class2
  ...
end Class2
  model Class1
  ...
end Class1
package MySubPackage
  model Class1
  ...
  end Class1;
end MySubPackage;
end MyPackage;

```

This can be expressed in the *Mathematica* style of *Modelica* syntax as:

```

Package[MyPackage,
  Model[Class3, ...];
  Model[Class2, ...];
  Model[Class1, ...];
  Package[MySubPackage,
    Model[Class1, ...];
  ]
]

```

Program Cells, Documentation Cells, and Graphic Cells

A notebook cell can include other cells and/or arbitrary text or graphics. In particular, a cell can include a code fragment or a graph with computational results.

The contents of cells can, for example, be one of the following forms.

- *Model* classes and parts of models (i.e., formal descriptions) that can be used for verification, compilation, and execution of simulation models.
- *Mathematical formulas* in the traditional mathematical two-dimensional syntax.
- *Text/documentation* used as comments to executable formal model specifications.
- *Dialogue forms* for specification and modification of input data.
- *Result tables* whose results can be automatically represented in (live) tables, which can even be automatically updated after recomputation.

- *Graphical result representation* with 2D vector and raster graphics as well as 3D vector and surface graphics.
- *2D structure graphs* that, for example, are used for various model structure visualizations such as connection diagrams and data structure diagrams.

A number of examples of these different forms of cells are available throughout this article.

Mathematics with 2D-Syntax, Greek Letters, and Equations

MathModelica uses the syntactic facilities of *Mathematica* to allow writing formulas in the standard well-known mathematical notation from textbooks in mathematics and physics. Certain parts of the *Mathematica* language syntax are however a bit unusual compared to many common programming languages. The reason for this design choice is to make it possible to use traditional mathematical syntax. The following three syntactic features are unusual:

- Implied multiplication is allowed (i.e., a space between two expressions, such as x and $f(x)$, means multiplication just as in traditional mathematics notation). A multiplication operator $*$ can be used if desired, but is optional.
- Square brackets are used around the arguments at function calls. Round parentheses are only used for grouping of expressions. The exception is `TraditionalForm`, which is discussed later in this section.
- Support for two-dimensional mathematical syntactic notation such as integrals, division bars, square roots, matrices, etc.

The reason for the unusual choice of square brackets around function arguments is that the implied multiplication makes the interpretation of round parenthesis ambiguous. For example, $f(x+1)$ can be interpreted either as a function call to f with the argument $x+1$ or f multiplied by $(x+1)$. The integral in the following cell contains examples of both implied multiplication and two-dimensional integral syntax. The cell style is called *MathModelica* input form (`StandardForm` in *Mathematica*) and is used for mathematics and *Modelica* code in *Mathematica* syntax.

$$\int \frac{x f[x]}{1 + x^2 + x^3} dx$$

There is also a purely textual input form using a linear sequence of characters. For example, this is used for entering *Modelica* models in the standard *Modelica* syntax and is currently the only cell format in *MathModelica* that can interpret standard *Modelica* syntax. However, all mathematics can also be represented in this syntax. The previous example in this textual format appears as follows.

```
Integrate[(x*f[x]) / (1 + x^2 + x^3), x]
```

Finally, there is also a cell format called `TraditionalForm`, which is very close to traditional mathematical syntax, avoiding the square brackets. The aforemen-

tioned syntactic ambiguities can be avoided if the formula is first entered using one of the earlier input forms and then converted to TraditionalForm.

$$\int \frac{x f(x)}{x^3 + x^2 + 1} dx$$

The *MathModelica* environment allows easy conversion between these forms using keyboard characters or menu items. In the following, we show a small example of a *Modelica* model class called SimpleDAE represented in the *Mathematica*-style syntax of *Modelica* that allows Greek characters and two-dimensional syntax. The apostrophe (') is used for the derivatives just as in traditional mathematics and in a *MathModelica* derivative with respect to time, corresponding to the *Modelica* der() operator. The *initial conditions* of the variables, if not explicitly specified in the declarations, are the default start-attribute values of the variables, which is zero for Real variables. Initial conditions for variables and derivatives can also be given in a special InitialEquation section.

```
Model[SimpleDAE,
  Real  $\beta_1$ ;
  Real  $x_2$ ;
  Equation[
    
$$\frac{\beta_1'}{1 + (\beta_1')^2} + \frac{\sin[x_2']}{1 + (\beta_1')^2} + \beta_1 x_2 + \beta_1 == 1;$$

    
$$\sin[\beta_1'] - \frac{x_2'}{1 + (\beta_1')^2} - 2 \beta_1 x_2 + \beta_1 == 0;$$

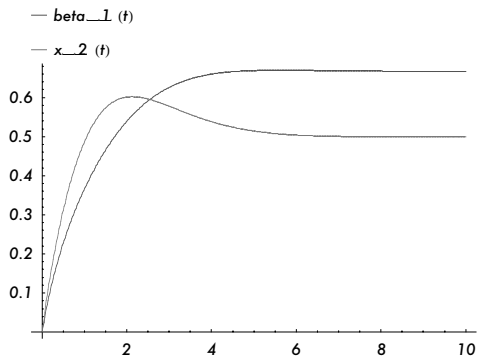
  ]]
```

We simulate the model for ten seconds by giving a Simulate command.

```
Simulate[SimpleDAE, {t, 0, 10}];
```

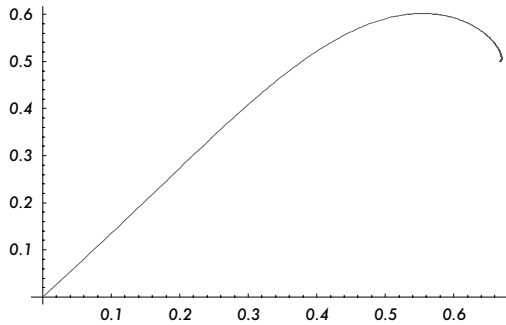
We use the command PlotSimulation for plotting the solutions for the two state variables, which of course are both functions of time, here denoted by t in *Mathematica* syntax.

```
PlotSimulation[{ $\beta_1[t]$ ,  $x_2[t]$ }, {t, 0, 10}]
```



A phase plane plot appears as follows.

```
ParametricPlotSimulation[{ $\beta_1[t]$ ,  $x_2[t]$ }, {t, 0, 10}]
```



□ 3.4. Environment and Language Extensibility

Programming environments need to be flexible to adapt to changing user needs. Without flexibility, a programming tool becomes too hard to use for practical needs and therefore obsolete. Adaptability and flexibility are especially important for integrated environments, since they need to interact with a number of external tools and data formats, contain many different functions, and usually need to add new ones.

There are two major ways to extend a programming environment:

- Extension of functionality (e.g., through user-defined commands, user-extensible menus, and a scripting language for programmability).
- Extension of language and notation (e.g., by facilities to add new syntactic constructs and new notation or extending the meaning of existing ones).

Mathematica has been designed from the start to be an inherently extensible environment, which is what is used in *MathModelica*. Almost anything can be redefined, extended, or added.

Scripting for Extension of Functionality

An interactive scripting language is a common way of providing extensibility of flexibility in functionality. The *MathModelica* environment primarily uses the *Mathematica* language and its interpreter as a scripting language, as can be seen from a number of examples in this article. Another possibility is to use the *Modelica* language itself as a scripting language (e.g., by providing an interpreter for the algorithmic and expression parts of the language). This can easily be realized in *MathModelica* since the intermediate form has been designed to be compatible with *Mathematica*, and we already have *Modelica* input cells—just use *Modelica* input cells for commands, which are sent to the *Mathematica* interpreter instead of the simulator.

Extensible Syntax and Semantics

As was already apparent in the section on mathematical syntax, *MathModelica* provides a *Mathematica*-like input syntax for *Modelica* in addition to the usual *Modelica* syntax. One reason is to give support for mathematical notation, as explained previously. Another reason is to provide user-extensible syntax.

This is easy since syntactic constructs in *Mathematica*, apart from the operators, use a simple prefix syntax: a keyword followed by square brackets surrounding the contents of the construct (i.e., the same syntax as for function calls). If there is a need to add a new construct, no changes are needed in the parser, and no reserved words need to be added. Just define a *Mathematica* function to do the desired symbolic or numeric processing.

The other major class of syntactic constructs are operators. There are special facilities in *Mathematica* to add new operators by defining their priority, operator syntax, and internal representation. It is also possible to extend the meaning of existing operators like $+$, $*$, $-$, etc. However, it is not possible to just use any *Mathematica* function or operator without a *Modelica* definition within a *Modelica* class. For this to work, a *MathModelica/Modelica* definition of the function or operator must be provided.

Mathematica versus Modelica Syntax

In order to show the difference between the standard *Modelica* textual syntax and the extensible *Mathematica*-like syntax, we first show a simple *Modelica* model in a *Modelica*-style input cell.

```
model SecondOrderSystem
  Real x(start=0);
  Real xdot(start=0);
  parameter Real a=1;
equation
  xdot=der(x);
  der(xdot)+a*der(x)+x=1;
end SecondOrderSystem;
```

The same model in the *Mathematica*-like *Modelica* syntax appears later. Note the use of the simple prefix syntax: a keyword followed by square brackets surrounding the contents of the construct. All reserved words, and by convention also predefined functions, and types in *MathModelica* start with an upper-case letter just as in *Mathematica*. Equation equality is represented by the `==` operator since `=` is the assignment operator in *Mathematica*. The derivative operator is the mathematical apostrophe (`'`) notation rather than `der()`. The semicolon (`;`) is a sequencing operator to combine more than one declaration, statement, or expression.

Note that the `Start` attribute values, e.g. for `x` and `xdot`, are defined using declaration modifier equations `Start==0`. These `Start` attributes are used as hints for the initial conditions when the simulation starts. The simulator is free to deviate somewhat from these hints if needed to obtain a consistent set of initial values by solving an equation system for the initial values. However, if the

attribute `Fixed` is `True` (default `False`), then the initial variable value is required to be the `Start` attribute value.

```

Model[SecondOrderSystem,
  Real x[{Start == 0}];
  Real xdot[{Start == 0}];
  Parameter Real a == 1;
Equation[
  xdot == x';
  xdot' + a*x' + x == 1
]
]

```

Choice of Mathematica Representation of Modelica Syntax

In order to represent *Modelica* in a *Mathematica*-compatible way and thereby potentially extend the *Mathematica* language, certain choices were made during the design and implementation of the *MathModelica* environment. Some of these design tradeoffs were not easy, and a number of variants have been implemented and tried out. Here we give a short motivation for our design in a few tricky cases.

- *Type prefixes.* In declarations type prefixes are separated from the declared entity using one or more space characters. This happens to be standard notation in common programming languages such as Java, C++, or C. This choice is unambiguous also in *Mathematica* since the usual *Mathematica* interpretation of space as a multiplication operator is not valid in a type declaration and is instead interpreted as a type prefix operation. Such type prefixes are represented using the `Prefixed[]` node that is always unparsed into space. Space is still parsed as implied multiplication in the usual expression context. Our implementation prevents the usual evaluation of declarations, which must be prohibited anyway regardless of the syntax, and converts multiplications in the type declaration prefix parts to `Prefixed[]` nodes.

Some people have suggested using the *Mathematica* `@` operator to represent such prefixes. This is however ambiguous since `@` is equivalent to function application. At unparsing it would be hard to know whether `Real@x` should be unparsed as `Real@x` or `Real[x]`. Moreover, the ambiguity is really serious and unsolvable in some cases such as array declarations, for example, the array declarations inside `WaveEquation`: `Sample` and `initialPressure` in Section 4.4.

- *Dot notation.* Accessing the member of an object using dot notation (`.`), for example, `person.length` is an almost universally accepted notation in object-oriented programming languages. However, this clashes with the (rather rare) use of `.` in *Mathematica* for inner product as the `Dot[]` operator. We experimented with several different operators, including the use of backtick (`'`). However, all these attempts eventually failed because of inappropriate priority and precedence of other operators. For example, assuming `persons` is an array of person records, `persons[[2]].length` accesses the `length` field of the second record.

However, when using backtick, `persons[[2]]'length` is incorrectly parsed into implied multiplication of `persons[[2]]` and `'length` due to the high priority of backtick. Therefore, *MathModelica* redefines `.` to be parsed into `Member[[]]` nodes, that is, `a.b` means `Member[a,b]`. To perform dot product between `a` and `b`, use the `FullForm Dot[a,b]`.

- *Underscore*. Most programming languages, including *Modelica*, allow underscore in identifiers. This is not possible in *Mathematica* since underscore is mapped to the `Blank[[]]` pattern-matching operator. Our solution is to use the *Mathematica* `\[UnderBracket]` character (`⏟`), which is very similar to underscore in appearance. For example, see its use in the design optimization model in Section 4.2 with the `idealGear`: `R2T1.flange⏟b` variable.

□ 3.5. Simulation, Translation, and Graphic Animation of CAD Models

The Model Editor provides an easy-to-use high-level user interface that works quite well for most application areas. However, for certain application areas, such as the design of 3D mechanical parts and assemblies of such parts, the two-dimensional user interface of the Model Editor is not very intuitive and sometimes hard to use. On the other hand, tools with 3D-interactive user interfaces for design of mechanical systems already exist. These are known as CAD systems for mechanical applications.

For these reasons we have developed an integration mechanism or a translator between existing CAD systems and the *MathModelica* environment. A CAD system is used as the interactive user interface to design the geometry, constraints, and connection structure of the mechanical application. This design is then automatically translated into a mechanical *Modelica* model for dynamic simulation. The generated *Modelica* model consists of connected instances of classes from the *Modelica* MultiBody System (MBS) library [35, 36]. Such translators integrated with the simulation environment have so far been developed for the two CAD systems SolidWorks [37] and AutoDesk's Mechanical Desktop [38].

We have also developed an OpenGL-based 3D visualizer and animation system called MVIS (*Modelica* VISualizer) [39] that provides an online dynamic display of the mechanical assembly during simulation, or an offline display based on saved state information for each time step.

Both the MVIS visualizer and the CAD translators are separate subsystems that communicate with the rest of the *MathModelica* environment using files and other means. They are not yet official parts of the *MathModelica* product release and are therefore indicated by a dotted line in the previously presented *MathModelica* structure diagram. The interplay between the simulation environment and the CAD environment is shown in Figure 14.

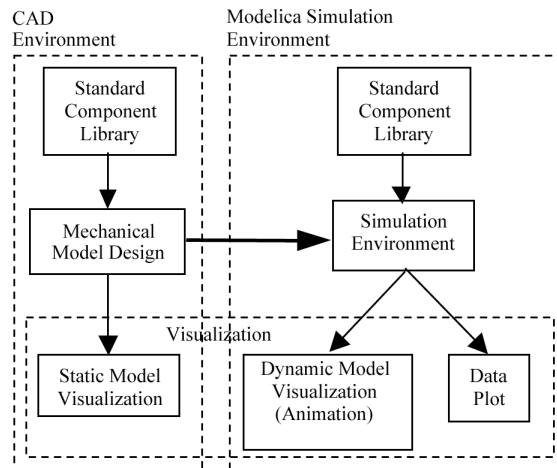


Figure 14. Functional structure of the information flow between the *MathModelica* simulation environment, the MIVS 3D visualizer, and the CAD environment.

Both translators are implemented as CAD system plug-ins that extract geometry, mass, inertia, and constraints information, and translate this information to *Modelica* source code. This code is combined with other code fragments (e.g., control system models) and simulated. The output can subsequently be visualized as a data plot of the system variables and/or as a 3D or 2D dynamic model animation. The 3D visualizations are scenes that display the geometry of the parts in motions prescribed by the simulation results. The graphical user interface of the CAD model and the output visualization capabilities of the simulation environment make it easy to describe and modify model geometry as well as examine analysis results at the same time. A more detailed picture of the translation and visualization mechanisms including associated data flows is shown in Figure 15.

Figure 15 shows a mechanical model designed in the AutoDesk's Mechanical Desktop environment serving as the starting point of the specification of the virtual prototype with dynamic properties. The model is first saved in the DWG format, which contains all the information, including connections and mates constraints, related to the geometrical properties of the parts and the whole mechanical assembly.

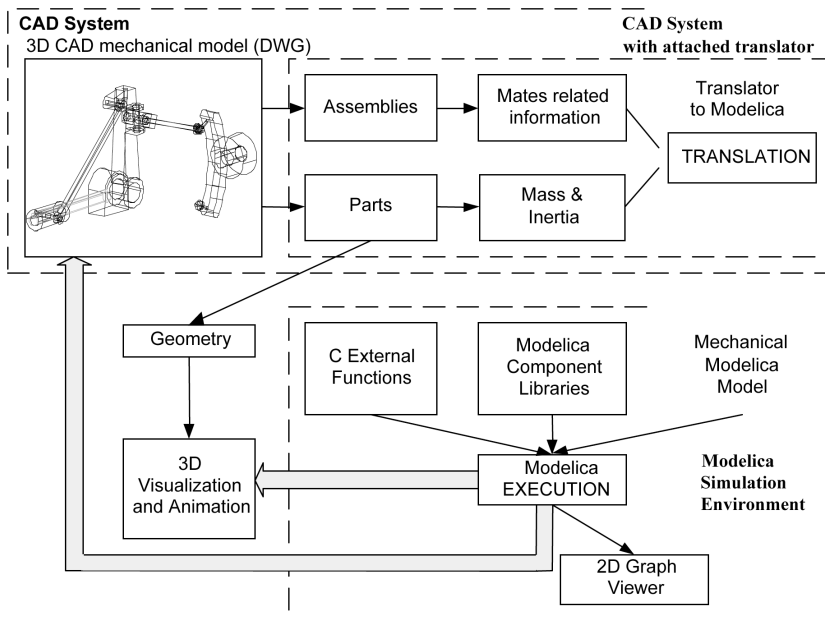


Figure 15. The path from a static CAD model to a dynamic system simulation and visualization.

The geometry of each part is exported to the .stl file format [40] for use by the MVIS visualizer. At the same time, the mass and inertia of the parts are extracted with mates information from the mechanical assembly. The translator uses this information to generate a corresponding set of *Modelica* class instances coupled by connections. This automatically generated *Modelica* file is processed by the *MathModelica* simulation environment. The simulation code can be enhanced by adding components from other *Modelica* libraries or by adding externally defined C code. In this phase electrical, control, or hydraulics components can be added to the generated mechanical model, providing multidomain simulation. The translated CAD model contains a set of dynamic equations of motion. The solution of these equations during simulation computes the dynamic response to a given set of initial conditions including force and/or torque loads, which might even be functions of time.

We might ask why develop yet another tool for multibody simulation of mechanical systems? There are already commercially available MBS simulation packages like ADAMS or WorkingModel3D. Many CAD systems are integrated with some multibody simulation tool. However, the primary limitation of these environments is the difficulty of integrating multidomain simulation within the same environment. Usually an interface to common simulation tools, like MATLAB and Simulink, is provided, but such solutions are not very flexible and do not give good performance because of the loose integration. By contrast, the *MathModelica* environment provides solutions to both of the following two major requirements:

- The need to integrate multidomain simulation in the same environment.
- The generation of quality documentation coupled to the design and code.

The main advantage of the *MathModelica* solution is that multidomain modeling and simulation is available in an integrated way in the same environment. This is provided in a way that is both very flexible and gives very efficient simulations, which, for example, is needed for tightly interacting system components like controllers embedded in mechanical systems. The control algorithms can, for example, be tested in parallel with the design of the mechanical parts of the system.

■ 4. Application Examples

This section gives a number of application examples of the use of the *MathModelica* environment. The intent is to demonstrate the power of integration and interactivity—the interplay between the object-oriented modeling and simulation capabilities of *Modelica* integrated with the powerful scripting facilities of *Mathematica* within *MathModelica*. This includes the representation of simulation results as 1D and 2D interpolating functions of time being combined with arithmetic operations and functions in expressions, advanced plotting facilities, and computational capabilities, such as design optimization, Fourier analysis, and solution of time-dependent PDEs.

□ 4.1. Advanced Plotting and Interpolating Functions

This section illustrates the flexible usage of simulation results (represented as interpolating functions) for further computations that may include simulation results in expressions, and for both simple and advanced plotting. The following simple bouncing ball model [12] is used in the simulation and plotting examples.

```
Model[BouncingBall, "Simple model of a bouncing ball",
  Constant Real g == 9.81 "Gravity constant";
  Parameter Real c == 0.9 "Coefficient of restitution";
  Parameter Real radius == 0.1 "Radius of the ball";
  Real height[{Start == 1}] "height of the ball center";
  Real velocity[{Start == 0}] "Velocity of the ball";
  Equation[
    height' == velocity;
    velocity' == -g;
    When[height ≤ radius,
      Reinit[velocity, -c Pre[velocity]]
    ]
  ]
]
```

Interpolating Function Representation of Simulation Results

The following simulation of the previous BouncingBall model is done for a short time period using very few points.

```
res1 = Simulate[BouncingBall, {t, 0, 0.5}, NumberOfIntervals → 10]
<SimulationData: BouncingBall : 2005-10-21 16:32:
  17.5845168 : {0., 0.5} : 16 data points : 2 events : 7 variables>
{c, Derivative[1], Derivative[1], g, height, radius, velocity}
```

The results returned by `Simulate` are represented by an access descriptor or handle. Note that the output also mentions the parameters `c` and `g` in the “variables” list even though their values are constant and not generated by the simulation. Some of the contents of such a descriptor are shown as the result of the previous call to `Simulate`. At this stage the simulation data is stored on disk and referenced by `res1`, which acts as a handle to the simulation data. When one of the variables from the last simulation is referenced, e.g. `height`, `radius`, etc., the data for that variable are loaded into the system in a load-by-need manner and represented as an ordinary *Mathematica* `InterpolatingFunction`.

Working with simulation result datasets in *Mathematica* is done in a very convenient way using the *Mathematica* `InterpolatingFunction` mechanism that encapsulates the data into a function object and provides interpolation so that the data acts as a regular function. The mechanism of loading simulation data into the system and representing it as a function object is performed by the function `VariableTrajectory`, which can be called explicitly as follows, but is called automatically on any variable from the last simulation when referenced.

```
h = VariableTrajectory[height]
Function[{t$120667}, Which[t$120667 < 0.428353,
  InterpolatingFunction[{{0., 0.428353}}, <>][t$120667],
  t$120667 ≥ 0.428353,
  InterpolatingFunction[{{0.428353, 0.5}}, <>][t$120667]]]
```

The expression returned from the `VariableTrajectory` is an anonymous function object having one formal parameter (for the time t) and consists of a body containing an expression that computes a value of the variable for the given time. In this case the body consists of a *Mathematica* `Which` expression that switches between two (or more) `InterpolatingFunction` objects, dependent on whether the time is less than the event time point at 0.428 or not. The `InterpolatingFunction` objects use interpolation order 3 as the default but can be altered by using options. Pure discrete data (i.e., data changing only at event points) is encapsulated by one `InterpolatingFunction` object with zero interpolation order to get a piecewise constant behavior in the interpolation. This is more efficient than using `Which` statements. The system will automatically choose the most efficient representation of these two alternatives.

The interpolation function can now be used in any computation in *Mathematica*. In this case we just evaluate the derivative at the time 0.2.

```
h'[0.2]
-1.962
```

In the previous case *Mathematica* made the differentiation of the `InterpolatingFunction` object `h`. Normally the derivatives of the simulation variables are also available in the simulation data.

As previously mentioned, to help the *MathModelica* user, variables of the most recent simulation are always accessible directly. In this case the function `VariableTrajectory` is automatically applied. Therefore, instead of assigning the variable `h` as done earlier, we can write the following and get the same result.

```
height' [0.2]
```

```
-1.962
```

Note, to keep variable values from previous simulations accessible, we should use `VariableTrajectory` on the appropriate variable, specifying the desired descriptor (e.g., `res1`).

```
h = VariableTrajectory[height, SimulationResult → res1];
```

Now we perform a new simulation, with the result denoted by `res2`.

```
res2 = Simulate[BouncingBall, {t, 0, 0.5},  
  NumberOfIntervals → 10, ParameterValues → c == 0.95];
```

Having the previous height curve represented as the function object `h`, we can easily compute the difference of the curves between the simulations (e.g., using a plot expression `height[t]-h[t]`).

PlotSimulation

First, we simulate the bouncing ball for eight seconds and store the results in the variable `res1` for subsequent use in the plotting examples.

```
res1 = Simulate[BouncingBall, {t, 0, 8}];
```

The command `PlotSimulation` is used for simple standard plots. If nothing else is specified (i.e., by the optional `SimulationResult` parameter), the command refers to the results from the most recent simulation. In the following diagram the height above ground of the ball from the bouncing ball model simulation is plotted for the first eight seconds of simulation. The optional parameter `PlotJoined` has been set to `False` to create a dotted plot.

```
PlotSimulation[height[t], {t, 0, 8}, PlotJoined → False]
```

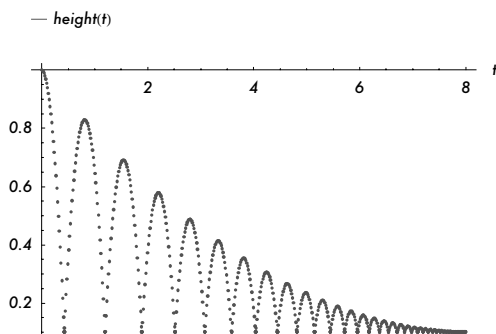


Figure 16. Dotted plot of bouncing ball example model.

PlotSimulation can also handle expressions containing simulated results. When this is done, a warning is returned to emphasize that interpolation is performed, which could result in a slightly less accurate plot.

```
PlotSimulation[e-Cos[height[t]], {t, 0, 8}]
```

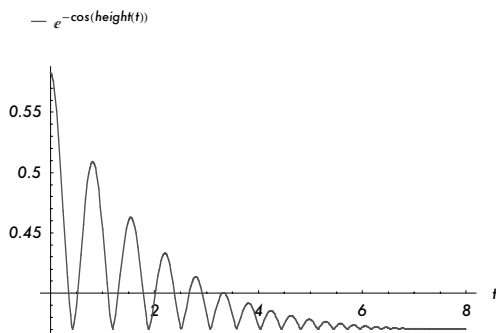


Figure 17. Plot of expression involving interpolated function of simulation result.

Plotting several arbitrary functions can be done using a list of function expressions instead of a single expression.

```
PlotSimulation[{height[t] +  $\sqrt{3}$ , Abs[velocity[t]]}, {t, 0, 8}]
```

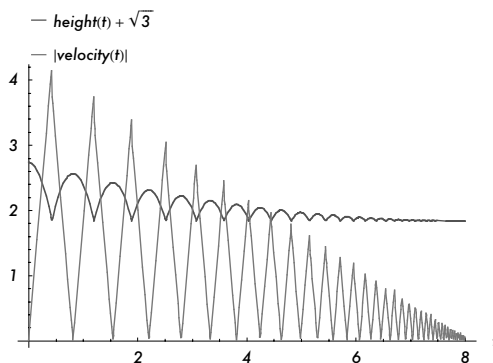


Figure 18. Plot of arbitrary functions in the same diagram.

Now we simulate the bouncing ball again but with a different value of the coefficient of restitution, c , which is changed to 0.95. The result is stored in `res2`.

```
res2 = Simulate[BouncingBall, {t, 0, 8}, ParameterValues → c == 0.95];
```

The optional argument `SimulationResult` specifies which simulation data to use. In this case we will use `res1` and `res2`. The two plots are stored as two graphics objects `gr1` and `gr2`, which are displayed together using the *Mathematica* command `Show`.

```
gr1 = PlotSimulation[height[t], {t, 0, 8},  
  SimulationResult → res1, DisplayFunction → Identity];  
gr2 = PlotSimulation[height[t], {t, 0, 8},  
  SimulationResult → res2, DisplayFunction → Identity];  
Show[GraphicsArray[{gr1, gr2}], DisplayFunction → $DisplayFunction]
```

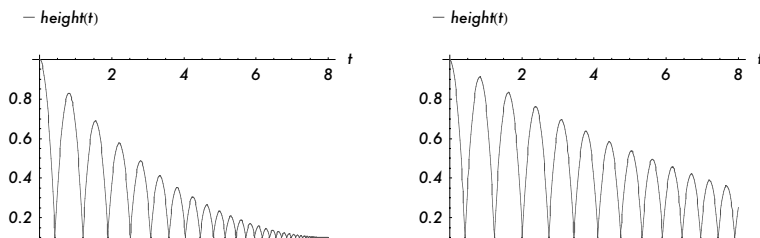


Figure 19. Parallel display of two diagrams.

It is possible to plot variables with the same name from several different simulations together. This is specified by an array value for the optional argument `SimulationResult`.

```
PlotSimulation[height[t], {t, 0, 8}, SimulationResult → {res1, res2}]
```

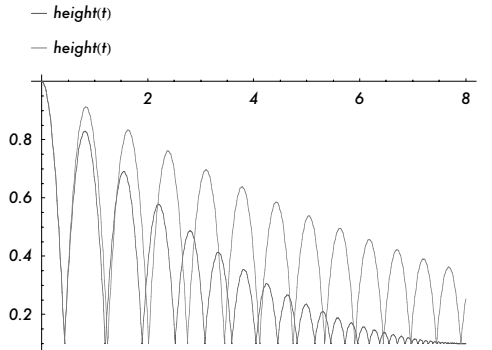


Figure 20. Plot of variables from several simulations in the same diagram.

The plot colors are specified by the `$PlotSimulationColors` predefined variable.

```
$PlotSimulationColors
```

```
{RGBColor[0, 0, 1], RGBColor[0, 0.5, 0], RGBColor[1, 0, 0],  
 RGBColor[0, 0.75, 0.75], RGBColor[0.75, 0, 0.75],  
 RGBColor[0.75, 0.75, 0], RGBColor[0.25, 0.25, 0.25]}
```

Here we check the color scheme of `$PlotSimulationColors`.

```
PlotSimulation[{height[t], height[t]^2, height[t]^3,  
 height[t]^4, height[t]^5, height[t]^6, height[t]^7}, {t, 0, 2}]
```

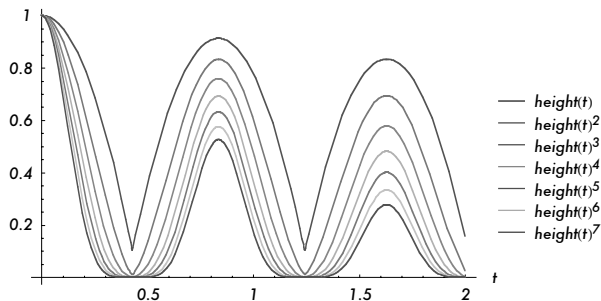


Figure 21. Plot of multiple curves with different colors.

ParametricPlotSimulation

Parametric plots can be done using `ParametricPlotSimulation`.

```
ParametricPlotSimulation[{height[t], velocity[t]}, {t, 0, 8}]
```

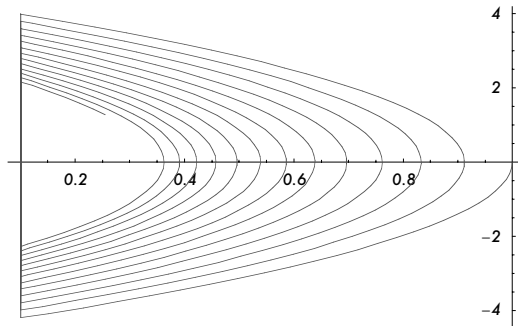


Figure 22. A parametric plot.

In the same way as `PlotSimulation`, the `ParametricPlotSimulation` function can handle several plots.

```
ParametricPlotSimulation[
  {{height[t], velocity[t]}, {velocity[t], height[t]}}, {t, 0, 8}]
```

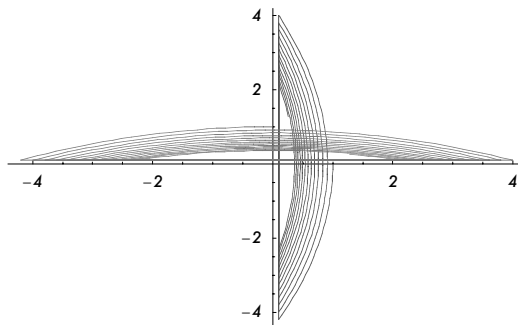


Figure 23. Multiple parametric plots in the same diagram.

`ParametricPlotSimulation` can also handle results from different simulations and plot only the actual data points.

```
ParametricPlotSimulation[{height[t], velocity[t]},
  {t, 0, 3}, PlotJoined → False, SimulationResult → {res1, res2}]
```

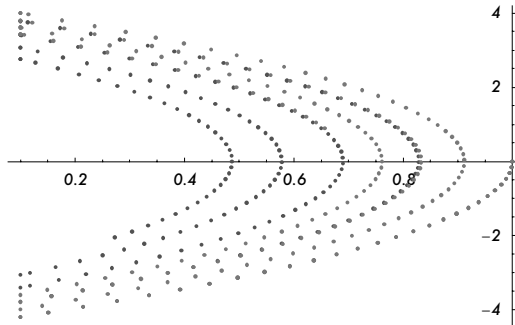


Figure 24. Parametric plots of data points from different simulations in the same diagram.

ParametricPlotSimulation3D

In this example we are going to use the Rossler attractor to show the `ParametricPlotSimulation3D` command. The Rossler attractor is named after Otto Rossler's work in chemical kinetics. The system is described by three coupled nonlinear differential equations.

$$\begin{aligned}\frac{dx}{dt} &= -y - x \\ \frac{dy}{dt} &= x + \alpha y \\ \frac{dz}{dt} &= \beta + (x - \gamma)z.\end{aligned}$$

Here α , β , and γ are constants. The attractor never forms limit circles nor does it ever reach a steady state. The model is shown in *Mathematica* syntax, enabling the use of Greek characters.

```
Model[Rossler, "Rossler attractor",
  Parameter Real  $\alpha$  == 0.2;
  Parameter Real  $\beta$  == 0.2;
  Parameter Real  $\gamma$  == 8;
  Real x[{Start == 1}];
  Real y[{Start == 3}];
  Real z[{Start == 0}];
  Equation[
    x' == -y - z;
    y' == x +  $\alpha$  y;
    z' ==  $\beta$  + x z -  $\gamma$  z
  ]
]
```

The model is simulated using different initial values. Changing these can considerably influence the appearance of the attractor.

```
Simulate[Rossler, {t, 0, 40}, InitialValues → {x == 2, y == 2.5, z == 0}];
```

The Rossler attractor is easy to plot using `ParametricPlotSimulation3D`:

```
ParametricPlotSimulation3D[  
  {x[t], y[t], z[t]}, {t, 0, 40}, AxesLabel → {X, Y, Z}
```

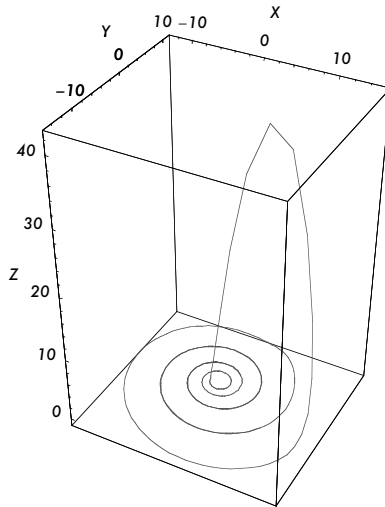


Figure 25. 3D parametric plot of interpolated curve from the Rossler attractor simulation.

The plot does not look smooth at some areas, especially for high values of Z . Let us take a look at the actual data points of the simulation.

```
ParametricPlotSimulation3D[{x[t], y[t], z[t]},  
  {t, 0, 40}, PlotJoined → False, AxesLabel → {X, Y, Z}]
```

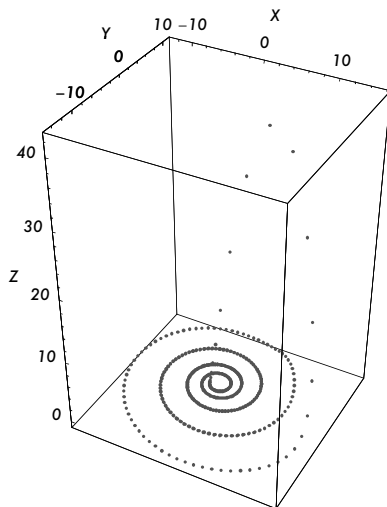


Figure 26. 3D parametric plot of actual data points from the Rossler attractor simulation.

There seem to be few data points outside the “rings.” This can be fixed by adding data points during simulation. The default value is 500. Let us try 1000 data points.

```
Simulate[Rosler, {t, 0, 40},  
  InitialValues → {x == 2, y == 2.5, z == 0}, NumberOfIntervals → 1000];
```

Now the plot looks smoother.

```
ParametricPlotSimulation3D[
  {x[t], y[t], z[t]}, {t, 0, 40}, AxesLabel -> {X, Y, Z}]
```

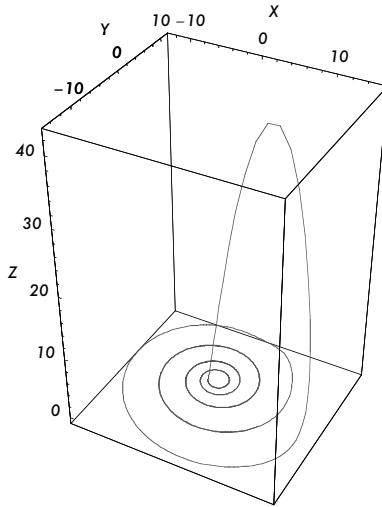


Figure 27. 3D parametric plot of a curve with many data points from the Rossler attractor simulation.

□ 4.2. Design Optimization

Let us describe an example of how the powerful *Mathematica* scripting language available within *MathModelica* can be utilized to solve nontrivial optimization problems that contain dynamic simulations. First, we will define a *Modelica* model of a linear actuator with spring damped stopping and then a first-order system. Using *MathModelica* scripting, we will then find a damping for the translational spring-damper such that the step response is as “close” as possible to the step response from a first-order system.

Consider the following model of a linear actuator with a spring damped connection to an anchoring point.

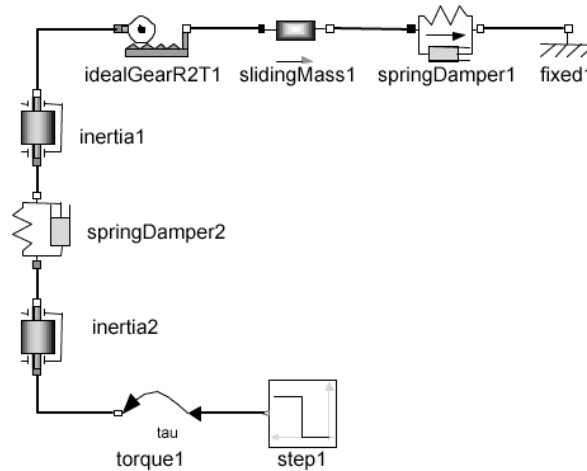


Figure 28. A LinearActuator model containing a spring damped connection to an anchoring point.

Here is the corresponding model code in *Mathematica*-style syntax (the squares contain graphical annotations for icons, lines, and so on).

```
Model[LinearActuator,
  ■;
  ■;
  Modelica.Mechanics.Translational.SlidingMass
    slidingMass1[{m == 0.5}] ■;
  Modelica.Mechanics.Translational.SpringDamper
    springDamper1[{d == 3, c == 20}] ■;
  Modelica.Mechanics.Translational.Fixed fixed1 ■;
  Modelica.Mechanics.Rotational.IdealGearR2T idealGearR2T1 ■;
  Modelica.Mechanics.Rotational.Inertia inertia1[{J == 0.1}] ■;
  Modelica.Mechanics.Rotational.SpringDamper
    springDamper2[{c == 15, d == 2}] ■;
  Modelica.Mechanics.Rotational.Inertia inertia2[{J == 0.1}] ■;
  Modelica.Mechanics.Rotational.Torque torque1 ■;
  Modelica.Blocks.Sources.Step step1 ■;
  Equation[
    Connect[inertia1.flange_b, idealGearR2T1.flange_a] ■;
    Connect[idealGearR2T1.flange_b, slidingMass1.flange_a] ■;
    Connect[slidingMass1.flange_b, springDamper1.flange_a] ■;
    Connect[springDamper1.flange_b, fixed1.flange_b] ■;
    Connect[inertia1.flange_a, springDamper2.flange_b] ■;
    Connect[springDamper2.flange_a, inertia2.flange_b] ■;
    Connect[inertia2.flange_a, torque1.flange_b] ■;
    Connect[torque1.tau, step1.y] ■
  ]
]
```

As a comparison, we also show the model code in *Modelica* textual syntax:

```
model LinearActuator
  Modelica.Mechanics.Translational.SlidingMass
  slidingMass1(m=0.5);
  Modelica.Mechanics.Translational.SpringDamper
  springDamper1(d=3,c=20);
  Modelica.Mechanics.Translational.Fixed fixed1;
  Modelica.Mechanics.Rotational.IdealGearR2T idealGearR2T1;
  Modelica.Mechanics.Rotational.Inertia inertia1(J=0.1);
  Modelica.Mechanics.Rotational.SpringDamper
  springDamper2(c=15,d=2);
  Modelica.Mechanics.Rotational.Inertia inertia2(J=0.1);
  Modelica.Mechanics.Rotational.Torque torque1;
  Modelica.Blocks.Sources.Step step1;
equation
  connect(inertia1.flange_b, idealGearR2T1.flange_a);
  connect(idealGearR2T1.flange_b, slidingMass1.flange_a);
  connect(slidingMass1.flange_b, springDamper1.flange_a);
  connect(springDamper1.flange_b, fixed1.flange_b);
  connect(inertia1.flange_a, springDamper2.flange_b);
  connect(springDamper2.flange_a, inertia2.flange_b);
  connect(inertia2.flange_a, torque1.flange_b);
  connect(torque1.tau, step1.y);
end LinearActuator;
```

The small square boxes in the code represent graphical annotations about the visual appearance of the components of the model, as seen in Figure 28. These annotations are hidden behind boxes; otherwise, the code becomes harder to read.

Here we simulate a step response and store the result in `res0`.

```
res0 = Simulate[LinearActuator, {t, 0, 5}];
PlotSimulation[slidingMass1.s[t], {t, 0, 5}]
```

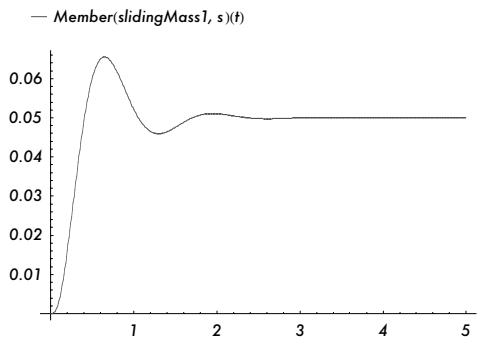


Figure 29. Plot of the step response from the linear actuator.

Assume that we have some freedom in choosing the damping in the translational spring-damper. A number of simulation runs shows what kind of behavior we have for different values of the damping parameter d . The *Mathematica* `Table` function is used with `Simulate` to collect the results into an array `res`. This array then contains the results from simulations of `LinearActuator` with a damping of 2 to 14 with a step size of 2 (i.e., seven simulations are performed).

```
res = Table[Simulate[LinearActuator, {t, 0, 4},
  ParameterValues → {springDamper1.d == s}], {s, 2, 15, 2}];
PlotSimulation[slidingMass1.s[t],
  {t, 0, 4}, SimulationResult → res, Legend → False]
```

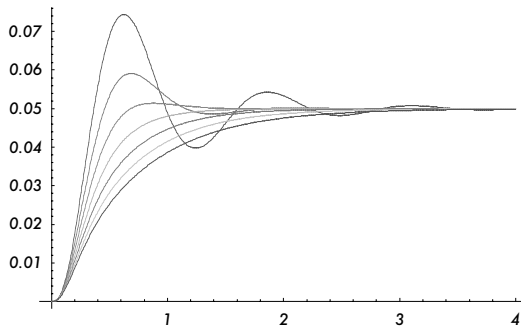


Figure 30. Plots of step responses from seven simulations of the linear actuator with different damping coefficients.

Now assume that we would like to choose the damping d so that the resulting system behaves as closely as possible to the following first-order system response, obtained by solving a first-order ODE using `NDSolve`:

```
Clear[y]
res1 = NDSolve[{0.2 y'[t] + y[t] == 0.05, y[0] == 0}, {y}, {t, 0, 4}];
```

We make a comparison with the step response we simulated first ($d=2$) and the first-order system.

```
PlotSimulation[{slidingMass1.s[t], y[t] /. res1},
  {t, 0, 4}, SimulationResult → res0]
```

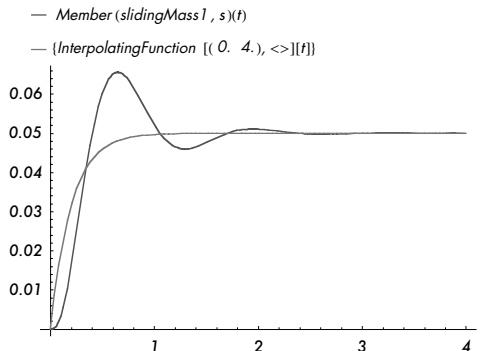


Figure 31. Comparison plot between the step response of the linear actuator and a first-order system.

Now, let us make things a little more automatic. Simulate and compute the integral of the square error from $t = 0$ to $t = 4$.

```
res = Simulate[LinearActuator,
  {t, 0, 4}, ParameterValues → {springDamper1.d == 3}];
NIntegrate[First[(y[t] /. res1) - slidingMass1.s[t]]^2, {t, 0, 4}]
0.000162508
```

Here we define a function, $f[a_]$, doing the same thing as previously, but for different spring-damper parameter values $d = a$

```
f[a_] := Module[{res, t}, res = Simulate[LinearActuator,
  {t, 0, 4}, ParameterValues → {springDamper1.d == a}];
NIntegrate[First[(y[t] /. res1) - slidingMass1.s[t]]^2, {t, 0, 4}]]
```

and tabulate some results for $2 \leq d = a \leq 10$.

```
res2 = Table[{a, f[a]}, {a, 2, 10, .5}]
{{2, 0.000317676}, {2.5, 0.000221485},
 {3, 0.000162508}, {3.5, 0.000125518}, {4, 0.000102754},
 {4.5, 0.000089821}, {5, 0.0000840739}, {5.5, 0.0000836789},
 {6, 0.0000874671}, {6.5, 0.0000945833}, {7, 0.000104409},
 {7.5, 0.000116474}, {8, 0.000130405}, {8.5, 0.000145933},
 {9, 0.000162832}, {9.5, 0.000180907}, {10, 0.000200013}}
```

The tabulated values are interpolated using an interpolating function object. The default interpolation order is 3.

```
f_pre = Interpolation[res2];
```

```
Plot[f_pre[a], {a, 2, 10}]
```

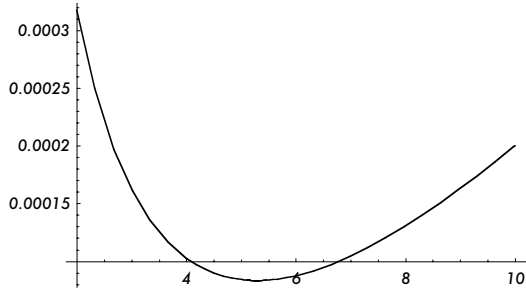


Figure 32. Plot of the error function for finding a minimum deviation from the desired step response.

The minimizing value of a can be computed using `FindMinimum`.

```
FindMinimum[f_pre[s], {s, 4}]
{0.0000832678, {s → 5.28665}}
```

Here is a simulation with the optimal parameter value.

```
Simulate[LinearActuator, {t, 0, 4},
  ParameterValues → {springDamper1.d == 5.28655}];
```

Here is a plot comparing the first- and second-order system response with a plot of the squared error amplified by a factor of 100.

```
PlotSimulation[{slidingMass1.s[t], y[t] /. res1,
  100 (slidingMass1.s[t] - (y[t] /. res1))^2}, {t, 0, 4}, Legend → False]
```

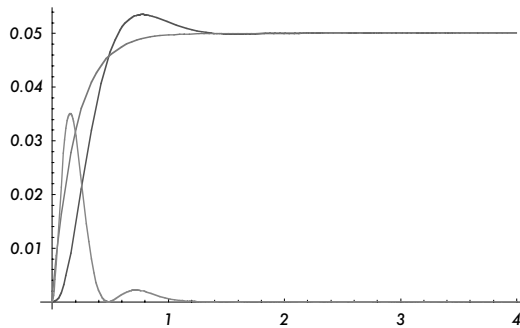


Figure 33. Comparison plot of the first- and second-order system step responses with the squared error.

□ 4.3. Fourier Analysis of Simulation Data

Consider a weak axis excited by a torque pulse train. The axis is modeled by three segments joined by two torsion springs. The following diagram is imported from the *MathModelica* Model Editor where the model was defined.

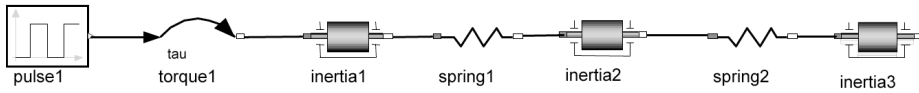


Figure 34. A WeakAxis model excited by a torque pulse train.

Here is the corresponding *Modelica* code in *Mathematica* syntax.

```
Model[WeakAxis,
  Modelica.Mechanics.Rotational.Torque torque1;
  Modelica.Mechanics.Rotational.Inertia inertia1;
  Modelica.Mechanics.Rotational.Spring spring1[{c == 0.7}];
  Modelica.Mechanics.Rotational.Inertia inertia2;
  Modelica.Mechanics.Rotational.Spring spring2[{c == 1}];
  Modelica.Mechanics.Rotational.Inertia inertia3;
  Modelica.Blocks.Sources.Pulse pulse1[{width == 1, period == 200}];
  Equation[
    Connect[pulse1.y, torque1.tau];
    Connect[torque1.flange_b, inertia1.flange_a];
    Connect[inertia1.flange_b, spring1.flange_a];
    Connect[spring1.flange_b, inertia2.flange_a];
    Connect[inertia2.flange_b, spring2.flange_a];
    Connect[spring2.flange_b, inertia3.flange_a]
  ]
]
```

The model code in *Modelica* syntax.

```
model WeakAxis
  Modelica.Mechanics.Rotational.Torque torque1;
  Modelica.Mechanics.Rotational.Inertia inertia1;
  Modelica.Mechanics.Rotational.Spring spring1(c=0.7);
  Modelica.Mechanics.Rotational.Inertia inertia2;
  Modelica.Mechanics.Rotational.Spring spring2(c=1);
  Modelica.Mechanics.Rotational.Inertia inertia3;
  Modelica.Blocks.Sources.Pulse pulse1(width=1,period=200);
equation
  connect(pulse1.y, torque1.tau);
  connect(torque1.flange_b, inertia1.flange_a);
  connect(inertia1.flange_b,spring1.flange_a);
  connect(spring1.flange_b, inertia2.flange_a);
  connect(inertia2.flange_b,spring2.flange_a);
  connect(spring2.flange_b, inertia3.flange_a);
end WeakAxis;
```

Here we simulate the model during 200 seconds.

```
Simulate[WeakAxis, {t, 0, 200}];
```

The plot of the angular velocity of the rightmost axis segment appears as follows.

```
PlotSimulation[{inertia3.w[t], torque1.τ[t]}, {t, 0, 200}]
```

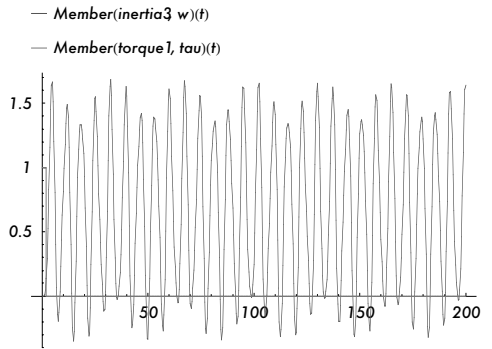


Figure 35. Plot of the angular velocity of the rightmost axis segment of the *WeakAxis* model.

Now, let us sample the interpolated function *Inertia3.w* using a sample frequency of 4Hz, and put the result into an array using the *Mathematica* Table array constructor.

```
data1 = Table[inertia3.w[t], {t, 0, 200, .25}];
```

Here we compute the absolute values of the discrete Fourier transform of *data1* with the mean value removed.

```
fdata1 = Abs[Fourier[data1 - Mean[data1]]];
```

Here we plot the first 80 points of the data.

```
ListPlot[fdata1[[Range[80]]],  
PlotStyle → {Hue[1.0], PointSize[0.015]}]
```

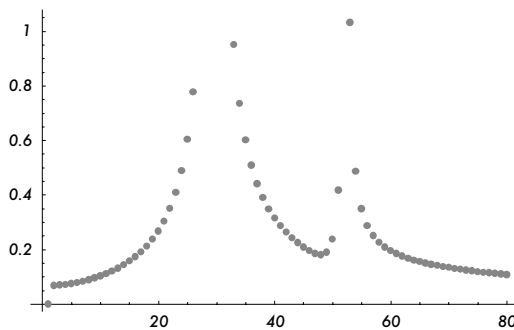


Figure 36. Plot of the data points of the Fourier transformed angular velocity.

It is easy to write a function `FourierPlot` that repeats the previous operations. `FourierPlot` also scales the axes such that amplitudes of trigonometric components are plotted against frequency (Hz).

```
FourierPlot[signal_, {t_, tmin_, tmax_}, T_, options___] :=
Module[{data1, n, data2, fdata1, fdata2, f},
  data1 = Table[signal, {t, tmin, tmax, T}];
  n = Length[data1]; data2 = data1 - Mean[data1];
  fdata1 = 2/Sqrt[n] Abs[Fourier[data2]];
  fdata2 = Drop[fdata1, -Round[n/2.]];
  f = Range[0, 1/(2*T), (1/(2*T) - 0)/Round[n/2.]];
  ListPlot[Transpose[{f, fdata2}], options] ]

FourierPlot[inertia3.w[t], {t, 0, 200}, 0.5,
  PlotRange -> All, PlotJoined -> True, PlotStyle -> Hue[1.0]]
```

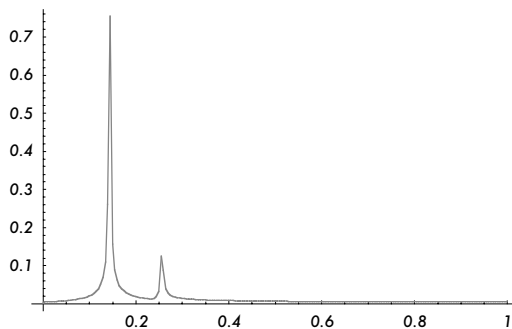


Figure 37. Plot of the curve of the Fourier transformed angular velocity.

It can be shown that the frequencies of the eigenmodes of the system are given by the imaginary parts of the eigenvalues of the following matrix (c_1 and c_2 are the spring constants).

$$\frac{1}{2\pi} \text{Eigenvalues} \left[\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ -c_1 & 0 & -c_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ -c_1 & 0 & -c_1 - c_2 & 0 & -c_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & -c_2 & 0 & -c_2 & 0 \end{pmatrix} \right] /. \{c_1 \rightarrow 0.7, c_2 \rightarrow 1\} //$$

Chop

```
{0.256077 i, -0.256077 i, 0.143344 i, -0.143344 i, 0, 0}
```

These values, 0.256077, 0.143344, fit very well with the peaks in the previous diagram.

□ 4.4. Solution and 2D-Interpolation of Discretized PDEs

Currently *Modelica* cannot handle partial differential equations directly since there is only the notion of differentiation with respect to time built into the language. However, in many cases derivatives with respect to other variables such as, for example, spatial dimensions can be handled by simple discretizations schemes easily implemented using the array capabilities in *Modelica*. Here we will give an example of how the one-dimensional wave equation can be represented in *Modelica* and how *MathModelica* can be used for simulation and display of the results, as well as representing the result as a two-dimensional interpolating function.

The one-dimensional wave equation is given by a partial differential equation of the following form

$$\frac{\partial^2 p}{\partial t^2} = c^2 \frac{\partial^2 p}{\partial x^2},$$

where $p = p[x, t]$ is a function of both space and time. As a physical example, let us consider a duct of length 10 where we let $-5 \leq x \leq 5$ describe its spatial dimension. Since *Modelica* can only handle time as the independent variable, we need to discretize the problem in the spatial dimension and approximate the spatial derivatives using difference approximations using the approximation

$$\frac{\partial^2 p}{\partial x^2} \approx \frac{p_{i-1} + p_{i+1} - 2 p_i}{\Delta x^2}$$

Utilizing this approach a *MathModelica* model of a duct whose pressure dynamics is given by the wave equation can be written as follows.

```
Model[WaveEquationSample,
  ModelicaImport[Modelica.SIunits];
  Parameter SIunits.Length L == 10 "Length of duct";
  Parameter Integer n == 30 "Number of sections";
  Parameter SIunits.Length dL ==  $\frac{L}{n}$  "Section length";
  Parameter SIunits.Velocity c == 1;
  SIunits.Pressure p[n, {start == initialPressure[n]}];
  Real dp[n, {start == Fill[0, n]}];
  Equation[
    p[[1]] == exp[-( $\frac{L}{2}$ )2];
    p[[n]] == exp[-( $\frac{L}{2}$ )2];
    dp == p';
    Do[
      dp[[i]]' ==  $\frac{c^2 ((p[[i+1]] - 2 p[[i]]) + p[[i-1]])}{dL^2}$ , {i, Range[2, n-1]}
    ]
  ]
]
```

Here we are using a *Modelica* function `initialPressure` (defined as follows) to specify the initial value of the pressure along the duct. Assume that we would like an initial pressure profile in the duct of the form e^{-x^2} (Figure 38).

```
Plot[E-x2, {x, -5, 5}, PlotRange → All]
```

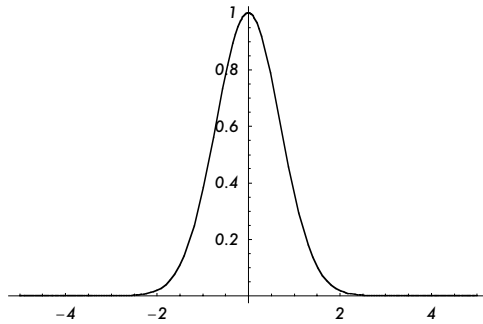


Figure 38. The initial pressure profile of the duct.

To provide an initial value profile (the `start` value of the vector variable `p`) for the pressure, we need to define a function that returns a sampled version of this profile. In principle this function could have been expressed in ordinary *Mathematica* function syntax with type extensions as in the *MathCode* system [41], but here we are using a more *Modelica*-like *Mathematica* syntax that is close to the syntax used for declaration of classes.

```
ModelicaFunction[initialPressure,
  Input Integer n;
  Output Real p[n];
  Protected[
    Parameter Modelica.SIunits.Length L == 10
  ];
  Algorithm[
    Do[p[[i]] = exp[-(L/2 + ((i-1)L)/(n-1))^2], {i, Range[1, n]}]]
]
```

Here we simulate the `WaveEquationSample` model.

```
Simulate[WaveEquationSample, {t, 0, 10}];
```

The result is packed into a 2D interpolation function.

```
intp = Interpolation[Flatten[Table[
  {-5. + (k-1)/29*10, t, p[[k]][t]}, {k, 1, 30}, {t, 0, 10, .2}], 1]];
```

A 3D plot of the pressure distribution in the duct can be obtained as follows.

```
Plot3D[intp[x, t], {x, -5, 5}, {t, 0, 10}, PlotPoints → 30,
  PlotRange → {-1, 1}, AxesLabel → {"x", "t", "p[x, t]"}]
```

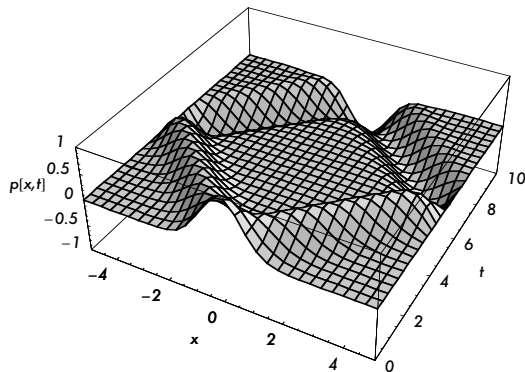


Figure 39. A 3D plot of the pressure distribution in the duct.

Let us also plot the wave as a function of position for a fixed point in time.

```
Plot[intp[x, 1.2], {x, -5, 5}]
```

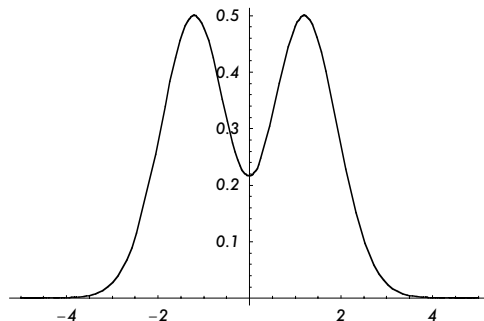


Figure 40. The wave as a function of position at time=1.2.

■ 5. Using the Symbolic Internal Representation

In order to satisfy the requirement of a well-integrated environment and language, the new *MathModelica* internal representation was designed with a *Mathematica*-compatible version of the syntax. Note that the *Mathematica* version of the syntax has the same internal abstract syntax tree representation and the same semantics as *Modelica*, but different concrete syntax. Which syntax to use—the standard *Modelica* textual syntax or the *Mathematica*-style syntax for *Modelica*—is however largely a matter of taste.

The fact that the *Modelica* abstract syntax tree representation is compatible with the *Mathematica* standard representation means that a number of symbolic operations, such as simplifying model equations, performing Laplace transformations, and performing queries on code as well as automatically constructing new code, is available to the user. The capability of automatically generating new code is especially useful in the area of model diagnosis, where there is often a need for generating a number of erroneous models for diagnosis based on corresponding fault scenarios.

□ 5.1. *Mathematica*-Compatible Internal Form

An inherent property of *Mathematica* is that code or models are normally not written as free formatted text. Instead, *Mathematica* expressions (also called terms) are used, internally represented as abstract syntax trees. These can be conveniently written in a tree-like prefix form or entered using standard mathematical notation. Every term is a number, an identifier, or a form such as:

```
head[term1, ..., termn]
```

For example, an expression such as $a+b$ is represented as `Plus[a,b]` in prefix form, also called `FullForm` syntax. A while loop is represented as the term `While[test,body]`.

In order to satisfy the requirement of a well-integrated environment, we designed the new *MathModelica* internal representation with a *Mathematica*-compatible version of the concrete syntax, called `MathModelicaForm`. Note that `MathModelicaForm` has the same abstract syntax trees and the same semantics as `ModelicaForm` representing standard *Modelica*, but different concrete syntax. This means that essentially the same language constructs are written differently, as illustrated in the following. Since the same internal representation is used, a cell expressed in `ModelicaForm` can easily be converted to `MathModelicaForm` or vice versa by just calling `GetDefinition` with a different value of the `Format` parameter.

The *Mathematica* language syntax uses some special operators and arbitrary arithmetic expressions composed from terms.

```
term1 ; ... ; termn,      // Sequencing operator ;
{term1, ..., termn},      // Array/list constructor {}
term1 term2,              // Implied multiplication by space instead of *
term1 = term2             // Equation equality
```

Internally the *MathModelica* system uses the `MathModelicaFullForm` format. This format is the abstract syntax of the *MathModelica* language where all the elements of the language have been defined to be easy to extract and compare for the functions operating on the *MathModelica* language representation, as well as achieving a high degree of compatibility with both *Modelica* and *Mathematica*.

The following is a simple constant declaration inside a model `Arr`.

```

Model[Arr,
  Constant Real unitarr[2, 2] == {{1, 0}, {0, 1}} "2D Identity"
]

```

This definition is stored internally in the `MathModelicaFullForm` format, which can be retrieved by calling the function `GetDefinition` that returns the internal abstract syntax tree representation of the model:

```
ff2 = GetDefinition[Arr, Format → MathModelicaFullForm];
```

The tree is wrapped into the node `Hold[]` to prevent symbolic evaluation of the model representation while we are manipulating it. All nodes are shown in prefix form except the array/list nodes shown as `{...}` instead of the prefix form `List[...]` for arrays.

```

Hold[SetType[Arr,
  TYPE[Model[
    Declaration[TYPE[Real, {2, 2}, {Constant}, {}],
    VariableComponent[unitarr,
      ValueBinding[{{1, 0}, {0, 1}}, {}, {}, Null]],
    "2D Identity"], {}, {}, {}, {}, Null, Null]]
]

```

A declaration of a variable such as `unitarr` is represented by the `Declaration` node in the abstract syntax. This node has two arguments: the type and the variable instance. The type is represented by the `TYPE` node which stores the name, array dimension, type attributes (`Constant`), and type modifications, which is empty in this case. The instance argument contains a `VariableComponent` including the name of the variable, the initialization (`ValueBinding`), at the end of the comment string that is associated with the variable.

There are several goals behind the design of the `MathModelicaFullForm` format, which are fulfilled in the current system:

- *Abstract syntax.* The format systematically sorts out the different constructs in the language making the navigation of types and code easier.
- *Preservation of the syntactic structure of both Modelica and Mathematica code.* This means that the mapping from *Modelica* to `MathModelicaFullForm` format should be injective (e.g., the source code can be recreated from the intermediate form), and that transformations from *Modelica* via `MathModelicaFullForm` into *Mathematica*-style *Modelica* form should be reversible.
- *Explicit semantic structure.* The format has reserved fixed attribute positions for certain kinds of semantic information to simplify semantic analysis and queries. There is also a *canonical subset* of the format that is even simpler for semantic analysis, but does not always recreate exactly the same source code since the same declaration often can be stated in several ways.
- *Symbol table and type representation format.* The `MathModelicaFullForm` format should be possible to use in the symbol table, e.g., to represent types. Types are represented by anonymous type expressions such as the

TYPE node in the previous example. Anonymous means that the type representation is separate from the entity having the type.

- *Internal standard.* The `MathModelicaFullForm` format should be used by all the parts of the *MathModelica* system.

Here we show a small model `SecondOrderSystem` in the different representations. First, the *Modelica* model is in the *Mathematica*-style *Modelica* syntax.

```
Model[SecondOrderSystem,
  Real x[{Start == 0}];
  Real xdot[{Start == 0}];
  Parameter Real a == 1;
  Equation[
    xdot == x';
    xdot' + a x' + x == 1
  ]
]
```

This model is parsed and stored internally as *Modelica* in the abstract syntax `MathModelicaFullForm` representation. This can be retrieved and unparsed into standard textual *Modelica* as follows.

```
GetDefinition[SecondOrderSystem, Format -> ModelicaForm]

model SecondOrderSystem
  Real x(start=0);
  Real xdot(start=0);
  parameter Real a=1;

equation
  xdot = der(x);
  der(xdot)+a*der(x)+x = 1;
end SecondOrderSystem;
```

We retrieve the model once more, but specify unparsing into the *Mathematica*-style *Modelica* syntax by setting the format argument to `MathModelicaForm`. The model is returned in a `Hold` node to prevent symbolic evaluation.

```
GetDefinition[SecondOrderSystem, Format -> MathModelicaForm]

Hold[Model[SecondOrderSystem,
  Real x[{Start == 0}];
  Real xdot[{Start == 0}];
  ParameterReal a == 1;
  Equation[
    xdot == x';
    xdot' + a x' + x == 1
  ]
]]
```

Finally we obtain the `MathModelicaFullForm` of the model. Certain operators like `{}`, `==`, `+`, etc. are still unparsed as infix notation. To get the prefix notation

List[], Equal[], Plus[], etc., for these operators, use the FullForm command on the result (not done here).

```
GetDefinition[SecondOrderSystem, Format → MathModelicaFullForm]

Hold[SetType[SecondOrderSystem,
  TYPE[Model[Declaration[TYPE[Real, {}, {}, {}],
    VariableComponent[x, Null, {}, {ElementModification[
      start, EqualBinding[0], {}, {}, Null]], Null]];
  Declaration[TYPE[Real, {}, {}, {}], VariableComponent[xdot, Null,
    {}, {ElementModification[start, EqualBinding[0], {}, {}, Null]],
    Null]];
  Declaration[TYPE[Real, {}, {Parameter}, {}],
    VariableComponent[a, EqualBinding[1], {}, {}, Null]];
  Equation[
    xdot == x';
    xdot' + a x' + x == 1
  ], {}, {}, {}, {}, {}, Null, Null]]]
```

□ 5.2. Extracting and Simplifying Model Equations

This section will illustrate a few user-accessible symbolic operations on equations, such as obtaining the system of equations and the set of variables from a *Modelica* model, and symbolically simplifying this system of equations with the intention of performing symbolic Laplace transformation.

Definition and Simulation of Model1

The example class Model1 has been drawn in the graphic Model Editor and imported into the following notebook.

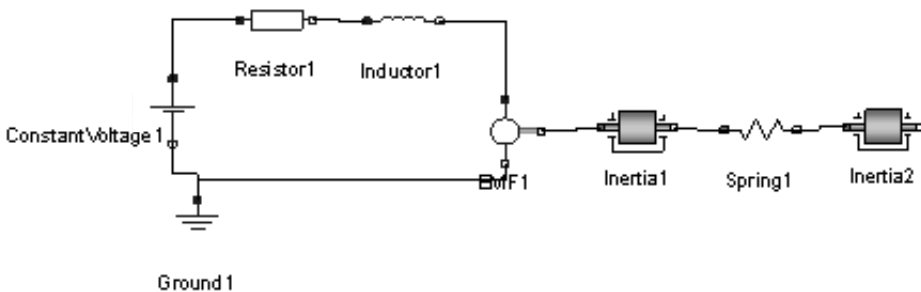


Figure 41. Connection diagram of Model1.

Here is the model code.

```

Model[Model1,
  ■;
  ■;
  Modelica.Electrical.Analog.Basic.Resistor Resistor1 ■;
  Modelica.Electrical.Analog.Basic.Inductor Inductor1 ■;
  Modelica.Electrical.Analog.Basic.Ground Ground1 ■;
  Modelica.Electrical.Analog.Basic.EMF EMF1 ■;
  Modelica.Electrical.Analog.Sources.ConstantVoltage
    ConstantVoltage1 ■;
  Modelica.Mechanics.Rotational.Spring Spring1[{c == 0.5'}] ■;
  Modelica.Mechanics.Rotational.Inertia Inertia1 ■;
  Modelica.Mechanics.Rotational.Inertia Inertia2 ■;
  Equation[
    Connect[ConstantVoltage1.p, Resistor1.p];
    Connect[Resistor1.n, Inductor1.p];
    Connect[Inductor1.n, EMF1.p];
    Connect[EMF1.n, Ground1.p];
    Connect[ConstantVoltage1.n, Ground1.p];
    Connect[EMF1.flange_b, Inertia1.flange_a];
    Connect[Inertia1.flange_b, Spring1.flange_a];
    Connect[Spring1.flange_b, Inertia2.flange_a]
  ]
]

```

We simulate the model, smooth the result, and make two plots, where the first is a plot of the product of the voltage and current over Resistor1.

```

res0 =
  Simulate[Model1, {t, 0, 25}, ParameterValues -> {Resistor1.R == 0.9}];
  PlotSimulation[{(Resistor1.v)[t] * (Resistor1.i)[t]}, {t, 0, 10}]

```

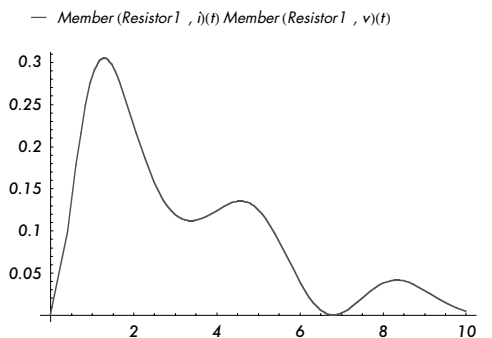


Figure 42. Plot of the current-voltage product over Resistor1 in Model1.

The second plot is parametric, where we plot the Resistor1 current against its derivative.

```
ParametricPlotSimulation[{(Resistor1.i)[t], (Resistor1.i)'[t]},
  {t, 0, 25}, SimulationResult → res0]
```

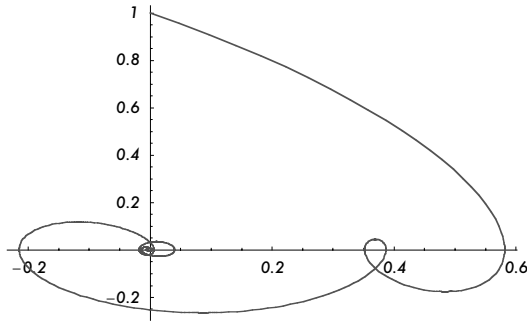


Figure 43. Parametric plots of the Resistor1 current against its derivative.

Some Symbolic Computations

Now, flatten Model1 and extract the model equations and the model variables as lists, and compute the lengths of these lists.

```
eqn = GetFlatEquations[Model1];
Length[eqn]
48
Length[GetFlatVariables[Model1]]
49
```

There is one equation less than the number of variables. Therefore, add an equation for zero torque on the right flange to the equation system.

```
eqn = Append[eqn, Inertia2.flange__b.tau == 0];
```

We would like to simplify the equations by eliminating the connector variables before further symbolic processing. First, we obtain the connector variables from the flattened model.

```
connvars = GetFlatConnectionVariables[Model1]
{Resistor1.p.v, Resistor1.p.i, Resistor1.n.v,
 Resistor1.n.i, Inductor1.p.v, Inductor1.p.i, Inductor1.n.v,
 Inductor1.n.i, Ground1.p.v, Ground1.p.i, EMF1.p.v, EMF1.p.i,
 EMF1.n.v, EMF1.n.i, EMF1.flange__b.phi, EMF1.flange__b.tau,
 ConstantVoltage1.p.v, ConstantVoltage1.p.i, ConstantVoltage1.n.v,
 ConstantVoltage1.n.i, Spring1.flange__a.phi, Spring1.flange__a.tau,
 Spring1.flange__b.phi, Spring1.flange__b.tau,
 Inertia1.flange__a.phi, Inertia1.flange__a.tau,
 Inertia1.flange__b.phi, Inertia1.flange__b.tau,
 Inertia2.flange__a.phi, Inertia2.flange__a.tau}
```

Here we use the `Eliminate` function for symbolic elimination of some variables from the system of equations.

```
eqn2 = Eliminate[eqn, connvars]

der[Inertia1.phi] == Inertia1.w && der[Inertia1.w] == Inertia1.a &&
der[Inertia2.phi] == Inertia2.w && der[Inertia2.w] == Inertia2.a &&
ConstantVoltage1.i == -(Resistor1.i) &&
ConstantVoltage1.v == (EMF1.k) (EMF1.w) +
  der[Inductor1.i] (Inductor1.L) + (Resistor1.i) (Resistor1.R) &&
ConstantVoltage1.V == (EMF1.k) (EMF1.w) +
  der[Inductor1.i] (Inductor1.L) + (Resistor1.i) (Resistor1.R) &&
EMF1.i == Resistor1.i && EMF1.v == (EMF1.k) (EMF1.w) &&
Inductor1.i == Resistor1.i &&
Inductor1.v == der[Inductor1.i] (Inductor1.L) &&
(Inertia1.a) (Inertia1.J) == (EMF1.k) (Resistor1.i) + Spring1.tau &&
Inertia1.phi == Inertia2.phi - Spring1.phi_rel &&
(Inertia2.a) (Inertia2.J) == -(Spring1.tau) &&
Resistor1.v == (Resistor1.i) (Resistor1.R) &&
(Spring1.c) (Spring1.phi_rel0) ==
  (Spring1.c) (Spring1.phi_rel) - Spring1.tau &&
Inertia2.flange_b.phi == Inertia2.phi &&
Inertia2.flange_b.tau == 0 &&
der(-1) [EMF1.w] == Inertia2.phi - Spring1.phi_rel
```

□ 5.3. Symbolic Laplace Transformation and Root Locus Computation

We would now like to perform a Laplace transformation of the symbolic equation system obtained in the previous section. This can be done by the application of two transformation rules: $\text{der}^{(-1)}[a_] \rightarrow \frac{a}{s}$, $\text{der}[b_] \rightarrow s \cdot b$. Note that $\text{der}^{(-1)}$ is the inverse of taking a derivative (i.e., an integration operation). Note also that the second rule contains an implied multiplication.

```

eq3 = eqn2 /. {der(-1)[a_] →  $\frac{a}{s}$ , der[b_] → s b}

s (Inertia1.phi) == Inertia1.w && s (Inertia1.w) == Inertia1.a &&
s (Inertia2.phi) == Inertia2.w && s (Inertia2.w) == Inertia2.a &&
ConstantVoltage1.i == -(Resistor1.i) &&
ConstantVoltage1.v == (EMF1.k) (EMF1.w) +
  s (Inductor1.i) (Inductor1.L) + (Resistor1.i) (Resistor1.R) &&
ConstantVoltage1.V == (EMF1.k) (EMF1.w) +
  s (Inductor1.i) (Inductor1.L) + (Resistor1.i) (Resistor1.R) &&
EMF1.i == Resistor1.i && EMF1.v == (EMF1.k) (EMF1.w) &&
Inductor1.i == Resistor1.i &&
Inductor1.v == s (Inductor1.i) (Inductor1.L) &&
(Inertia1.a) (Inertia1.J) == (EMF1.k) (Resistor1.i) + Spring1.tau &&
Inertia1.phi == Inertia2.phi - Spring1.phi_rel &&
(Inertia2.a) (Inertia2.J) == -(Spring1.tau) &&
Resistor1.v == (Resistor1.i) (Resistor1.R) &&
(Spring1.c) (Spring1.phi_rel0) ==
  (Spring1.c) (Spring1.phi_rel) - Spring1.tau &&
Inertia2.flange_b.phi == Inertia2.phi &&
Inertia2.flange_b.tau == 0 &&
 $\frac{EMF1.w}{s}$  == Inertia2.phi - Spring1.phi_rel

```

Here we introduce short names for the model parameter to obtain a more concise symbolic notation.

```

shortnames =
  {Resistor1.R → R, Inductor1.L → L, EMF1.k → k, Inertia1.J → J1,
   Spring1.c → c1, Spring1.phi_rel0 → 0, Inertia2.J → J2};

```

Then we derive the relation between Inertia2.w and the input voltage.

```

eq4 =
  Eliminate[eq3, Complement[GetFlatNonConnectionVariables[Model1],
    {Inertia2.w}]] /. shortnames

(k c1 (ConstantVoltage1.V) ==
  k2 c1 (Inertia2.w) + R s c1 J1 (Inertia2.w) + L s2 c1 J1 (Inertia2.w) +
  k2 s2 J2 (Inertia2.w) + R s c1 J2 (Inertia2.w) + L s2 c1 J2 (Inertia2.w) +
  R s3 J1 J2 (Inertia2.w) + L s4 J1 J2 (Inertia2.w)) && s ≠ 0

```

The transfer function H is obtained by symbolically solving for Inertia2.w in the equation system eq4 and using the obtained solution on a form Inertia2.w → expr to eliminate Inertia2.w, thus obtaining H .

```

H[s_] = First[ $\frac{\text{Inertia2.w}}{\text{ConstantVoltage1.V}}$  /. Solve[eq4, Inertia2.w]]

(k c1) / (k2 c1 + R s c1 J1 + L s2 c1 J1 +
  k2 s2 J2 + R s c1 J2 + L s2 c1 J2 + R s3 J1 J2 + L s4 J1 J2)

```

A Root Locus Computation

Here a list of model parameter values is defined for subsequent use.

```
parametervalues = {R → 1, L → 1, c1 → 1, J1 → 1, J2 → 1, k → 1};
```

Here we compute the poles of the transfer function to obtain the root locus.

```
poles = CharacteristicRoots[H[s], s]

{Root[k2 c1 + R c1 J1 #1 + R c1 J2 #1 + L c1 J1 #12 +
  k2 J2 #12 + L c1 J2 #12 + R J1 J2 #13 + L J1 J2 #14 &, 1],
 Root[k2 c1 + R c1 J1 #1 + R c1 J2 #1 + L c1 J1 #12 + k2 J2 #12 +
  L c1 J2 #12 + R J1 J2 #13 + L J1 J2 #14 &, 2],
 Root[k2 c1 + R c1 J1 #1 + R c1 J2 #1 + L c1 J1 #12 + k2 J2 #12 +
  L c1 J2 #12 + R J1 J2 #13 + L J1 J2 #14 &, 3],
 Root[k2 c1 + R c1 J1 #1 + R c1 J2 #1 + L c1 J1 #12 + k2 J2 #12 +
  L c1 J2 #12 + R J1 J2 #13 + L J1 J2 #14 &, 4]}

N[poles /. parametervalues]

{-0.395123 - 0.506844 i, -0.395123 + 0.506844 i,
 -0.104877 - 1.55249 i, -0.104877 + 1.55249 i}
```

Here is a root locus plot that substitutes values for the model parameters.

```
ParametricPlotComplexPlane[CharacteristicRoots[H[s], s] /.
 {R → 1, L → 1, c1 → 0.7, J1 → 1, J2 → JJ, k → 1}, {JJ, 0.1, 2}]
```

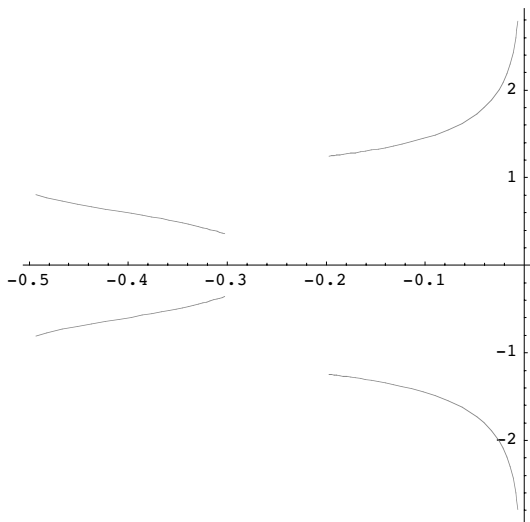


Figure 44. Root locus plot over the complex plane.

□ 5.4. Queries and Automatic Generation of Models

The following example of advanced scripting shows how the easily accessible internal representation in the form of abstract syntax trees can be used for automatic generation of models. The `CircuitTemplateFn` is a function returning a symbolic representation of a model. This function has two formal pattern parameters where the second one specifies an internal structure. The first parame-

ter is `name_`, which matches symbolic names. The underscore in `name_` is not part of the parameter identifier itself, it is just a short form of the syntax `name:_`, which means that `name` will match any item.

The second pattern parameter is the list `{type1_,type2_,type3_}`, internally containing the three pattern parameters `type1_,type2_,type3_`. This second parameter will therefore only match lists of length 3, thereby binding the pattern variables `type1`, `type2`, and `type3` to the three type names presumably occurring in the list at pattern matching. For example, matching `{type1_,type2_,type3_}` against the list `{Capacitor,Conductor,Resistor}` will bind the variable `type1` to `Capacitor`, `type2` to `Conductor`, and `type3` to `Resistor`.

```
CircuitTemplateFn[name_, {type1_, type2_, type3_}] := (
  Model[name,
    type1 a;
    type2 b;
    type3 c;
    Modelica.Electrical.Analog.Basic.Ground g;
    Equation[
      Connect[g.p, a.p];
      Connect[a.n, b.p];
      Connect[b.p, c.p];
      Connect[b.n, g.p];
      Connect[c.n, g.p]
    ]
  )
```

The aim of this exercise is to automatically generate models based on this template for all combinations of the types that extend the type `OnePort` in the library package `Modelica.Electrical.Analog.Basic`.

First, we need to extract all the types that extend the type `OnePort` in the library package `Modelica.Electrical.Analog.Basic`. This is done by performing a query operation on the internal form using the `Select` function that has two arguments: the list to be searched, and a predicate function returning `True` or `False`. Only the elements for which the predicate is `True` are returned. In this case the query is performed on the list of model names in the package `Modelica.Electrical.Analog.Basic`. This list is returned by the function `ListModelNames`.

First, we call `GetDefinition` to load the `Modelica.Electrical.Analog.Basic` package into the internal symbol table.

```
GetDefinition[Modelica.Electrical.Analog.Basic];
```

Then we perform the actual query.

```

types = Select[
  ListModelNames[Modelica.Electrical.Analog.Basic],
  Function[modelName,
    Not[FreeQ[GetDefinition[modelName, Format -> MathModelicaFullForm],
      HoldPattern[Extends[TYPE[Modelica.Electrical.
        Analog.Interfaces.OnePort, {}, {}, {}]]]]]]]]
{Modelica.Electrical.Analog.Basic.Inductor,
 Modelica.Electrical.Analog.Basic.Capacitor,
 Modelica.Electrical.Analog.Basic.Conductor,
 Modelica.Electrical.Analog.Basic.Resistor}

```

All 64 three-type combinations (e.g., {Inductor,Inductor,Inductor}, {Inductor,Inductor,Capacitor}, etc.), whose prefixes are not shown for brevity, of the four types {Inductor,Capacitor,Conductor,Resistor} are computed by taking a generalized outer product of the three types lists, which is flattened.

```

typecombinations = Flatten[Outer[List, types, types, types], 2];
Length[typecombinations]
64

```

We generate a list of 64 synthetic model names by concatenating the string "foo" with numbers, using the *Mathematica* string concatenation operation "<>".

```

names = Table[ToExpression["foo" <> ToString[i]], {i, 64}]
{foo1, foo2, foo3, foo4, foo5, foo6, foo7, foo8, foo9, foo10,
 foo11, foo12, foo13, foo14, foo15, foo16, foo17, foo18, foo19,
 foo20, foo21, foo22, foo23, foo24, foo25, foo26, foo27, foo28,
 foo29, foo30, foo31, foo32, foo33, foo34, foo35, foo36, foo37,
 foo38, foo39, foo40, foo41, foo42, foo43, foo44, foo45, foo46,
 foo47, foo48, foo49, foo50, foo51, foo52, foo53, foo54, foo55,
 foo56, foo57, foo58, foo59, foo60, foo61, foo62, foo63, foo64}

```

Here all 64 test models are created by the call to `MapThread`, which applies `CircuitTemplateFn` to each combination.

```

MapThread[CircuitTemplateFn, {names, typecombinations}];

```

Here we retrieve the definition of one of the automatically generated models, `foo53`, and unparse it from its internal representation to the *Modelica* textual form.

```

GetDefinition[foo53, Format -> MathModelicaForm]

```

```

Model[foo53,
  Modelica.Electrical.Analog.Basic.Resistor a;
  Modelica.Electrical.Analog.Basic.Capacitor b;
  Modelica.Electrical.Analog.Basic.Inductor c;
  Modelica.Electrical.Analog.Basic.Ground g;
  Equation[
    Connect[g.p,a.p];
    Connect[a.n,b.p];
    Connect[b.p,c.p];
    Connect[b.n,g.p];
    Connect[c.n,g.p]
  ]
]

```

Now we create a `Total` model within which all 64 generated models will be instantiated. First, create the an empty model.

```

Model[Total,
];

```

Then we use the `Within` statement to move the current scope inside the model and make a declaration (i.e., instantiation of the first test model).

```

Within[Total]

Declare[foo1 m1]

```

Since we are free to use *Mathematica* scripting, we should use a loop for the 63 remaining declarations.

```

Do[
  With[{type = names[[i]],
        instanceName = ToExpression["m" <> ToString[i]]},
    Declare[type instanceName]
  ],
  {i, 2, Length[names]}
];

```

Finally, we move the scope back to the global scope.

```

EndWithin[]

```

Here we retrieve the generated model `Total`, where we have abbreviated the output to save some space.

```

GetDefinition[Total, Format -> MathModelicaForm]

```

```

Model[Total,
  foo64 m64;
  foo63 m63;
  foo62 m62;
  foo61 m61;
  foo60 m60;
  ...
  foo3 m3;
  foo2 m2;
  foo1 m1
]

```

Finally, we simulate the `Total` model to verify that the test models are semantically correct.

```
Simulate[Total, {t, 0, 1}];
```

□ 5.5. Language Extension Example for PDEs

As previously stated, the uniform prefix syntax makes it easy to experiment with language extensions since both the syntax and the internal representation are obtained automatically. The following example is from an experiment in extending *Modelica* with partial differential equations [42]. Here we have added a new restricted class called `Domain`, the prefix `Space`, and a new kind of equation section called `Boundary` containing equations that specify boundary conditions.

```

Class[TestModel,
  Parameter Real xc == 0;
  Parameter Real yc == 0;
  Parameter Real r = 1;
  Parameter Real delta = 0;
  Domain Circle dom[{xc == xc, yc == yc, r == r}];
  Space Real u;
  Boundary[
    u[{dom, 0}] == finit[dom.x, dom.y];
    ∂t u[{dom, 0}] == 0;
    u[{dom.lefthalf, time}] == delta;
    ∂x u[{dom.lefthalf, time}] == 0;
    ∂y u[{dom.lefthalf, time}] == 0;
    u[{dom.righthalf, time}] == delta;
    ∂x u[{dom.righthalf, time}] == 0;
    ∂y u[{dom.righthalf, time}] == 0;
  ];
  Equation[
    ∂{t,2} u[{dom, time}] == ∂{x,2} u[{dom, time}] + ∂{y,2} u[{dom, time}];
  ];
];

```

Here is a plotted result of a solution at a specific time instant from running the prototype simulator on this model.

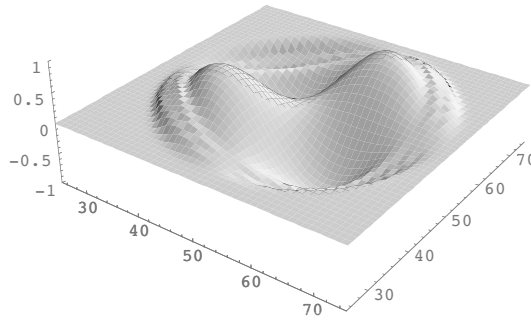


Figure 45. Plot of PDEs solution at a specific time instant.

■ 6. Conclusion

This article has presented a number of important issues concerning integrated interactive programming environments, especially with respect to the *MathModelica* environment for object-oriented modeling and simulation. We have especially emphasized environment properties such as integration and extensibility.

One of the current strong trends in software systems is the gradual unification of documents and software. Everything will eventually be integrated into a uniform, perhaps XML-based, representation. The integration of documents, model code, graphics, etc. in the *MathModelica* environment based on *Mathematica* capabilities is one strong example of this trend.

Another important aspect is extensibility. Experience has shown that tools with built-in extensibility mechanisms can cope with unforeseen user needs to a great extent, and therefore often have a substantially longer effective usage lifetime.

The *MathModelica* system integrated with *Mathematica* is currently one of the best existing examples of advanced integrated extensible environments. However, as most systems, it is not perfect. There are still a number of possible future improvements in the system including enhanced programmability and extensibility.

■ Acknowledgments

I would like to thank the entire *MathModelica* team from MathCore AB and the Dymola team at Dynasim, as well as my colleagues at PELAB—the Programming Environment Laboratory, without whom this work would not have been possible. I would specially also like to thank Peter Bunus for inspiration and help in preparing this article.

I would like to acknowledge the following members of the *MathModelica* team for contributions to the design and implementation of the *MathModelica* system, as well as providing several examples in this article: Peter Aronsson, Jan Brugård, Peter Bunus, Jonas Elbornsson, Vadim Engelsson, Daniel Hedberg, Andreas Idebrant, Lucian Popescu, and Yelena Turetskaya. I also thank Mikael Adlers, Henrik Johansson, Andreas Karström, and Pontus Lidman, for contributions to an early version of *MathModelica*, and Levon Saldamli for the PDE example in Section 5.5. Special thanks to Mats Jirstrand and Johan Gunnarsson who made substantial contributions to the early version of *MathModelica*. Moreover, Jirstrand wrote most of sections 4.2, 4.3, and 5.3, and Gunnarsson most of Section 5.4 in this article.

The author would also like to thank the other members of the Modelica Association for inspiring discussions and contributions to the *Modelica* language design. The most recent version at the time of this writing, *Modelica* 2.2, was released February 2, 2005. The Modelica Association was formed in Linköping Sweden, on February 5, 2000 and is responsible for the design of the *Modelica* language (see www.modelica.org). Special thanks to Hilding Elmqvist, who was the first chairman of the Modelica Association, and to Martin Otter, second chairman of the Modelica Association.

Contributors to the Modelica Language, Version 2.2

Bernhard Bachmann, University of Applied Sciences, Bielefeld, Germany

John Batteh, Ford Motor Company, Dearborn, MI, U.S.A.

Dag Brück, Dynasim, Lund, Sweden

Francesco Casella, Politecnico di Milano, Milano, Italy

Christoph Clauß, Fraunhofer Institute for Integrated Circuits,
Dresden, Germany

Jonas Eborn, Modelon AB, Lund, Sweden

Hilding Elmqvist, Dynasim, Lund, Sweden

Rüdiger Franke, ABB Corporate Research, Ladenburg, Germany

Peter Fritzzon, Linköping University, Sweden

Anton Haumer, Technical Consulting & Electrical Engineering,
St. Andrae-Woerden, Austria

Christian Kral, arsenal research, Vienna, Austria

Sven Erik Mattsson, Dynasim, Lund, Sweden

Chuck Newman, Ford Motor Company, Dearborn, MI, U.S.A.

Hans Olsson, Dynasim, Lund, Sweden

Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany

Markus Plainer, arsenal research, Vienna, Austria

Adrian Pop, Linköping University, Sweden

Katrin Prölß, Technical University Hamburg-Harburg, Germany

André Schneider, Fraunhofer Institute for Integrated Circuits,
Dresden, Germany

Christian Schweiger, German Aerospace Center, Oberpfaffenhofen, Germany

Michael Tiller, Ford Motor Company, Dearborn, MI, U.S.A.

Hubertus Tummescheit, Modelon AB, Lund, Sweden

Hans-Jürg Wiesmann, ABB Switzerland Ltd., Corporate Research,
Baden, Switzerland

Contributors to the Modelica Standard Library, Version 2.0

Peter Beater, University of Paderborn, Germany

Christoph Clauß, Fraunhofer Institute for Integrated Circuits,
Dresden, Germany

Martin Otter, German Aerospace Center, Oberpfaffenhofen, Germany

André Schneider, Fraunhofer Institute for Integrated Circuits,
Dresden, Germany

Funding Organizations

Several funding organizations have over the years contributed to research on object-oriented modeling language design and simulation tool technology at PELAB—the Programming Environment Laboratory, Department of Computer and Information Science, Linköping University, preceding the development of the *MathModelica* system. These include NUTEK—The Swedish Board for Technical Development, for example, within the Modelica Tools project, TFR—the previous Swedish Council for Technical Research, SSF—the Swedish Strategic Research Foundation under the ECSEL project, WITAS—the Wallenberg Laboratory for Information Technology and Autonomous Systems at Linköping University, and EU under the projects GIPE-II, PREPARE, and Europort. We also acknowledge support from SKF AB.

■ References

- [1] P. Fritzson and D. Fritzson, “The Need for High-Level Programming Support in Scientific Computing Applied to Mechanical Analysis,” *Computers and Structures*, **45**(2), 1992 pp. 387–395.
- [2] E. Sandewall, “Programming in an Interactive Environment: The LISP Experience,” *ACM Computing Surveys (CSUR)*, **10**(1), 1978 pp. 35–71.
- [3] P. Fritzson, “Symbolic Debugging through Incremental Compilation in an Integrated Environment,” *The Journal of Systems and Software*, **3**(4), 1983 pp. 285–294.
- [4] P. Fritzson, L. Viklund, J. Herber, and D. Fritzson, “Industrial Application of Object-Oriented Mathematical Modeling and Computer Algebra in Mechanical Analysis,” in *Proceedings of the Seventh International Conference on Technology of Object-Oriented Languages and Systems (TOOLS EUROPE’92)*, Dortmund, Germany (G. Heeg, B. Magnusson, and B. Meyer, eds.), Hertfordshire, UK: Prentice Hall International (UK) Ltd., 1992 pp. 167–181.
- [5] P. Fritzson, L. Viklund, D. Fritzson, and J. Herber, “High-Level Mathematical Modeling and Programming in Scientific Computing,” *IEEE Software*, **12**(4), 1995 pp. 77–87.
- [6] W. Teitelman, “Toward a Programming Laboratory,” in *Proceedings of the First International Joint Conference on Artificial Intelligence (IJCAI)*, Washington, DC (D. E. Walker and L. M. Norton, eds.), Los Altos, CA: William Kaufmann Publishers, 1969 pp. 1–8.
- [7] W. Teitelman et al., *InterLISP Reference Manual*, Palo Alto, CA: Xerox Palo Alto Research Center, 1974.
- [8] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Redwood City, CA: Addison-Wesley, 1988.

- [9] S. Wolfram, *The Mathematica Book*, 3rd ed., Champaign, Oxford: Wolfram Media/Cambridge University Press, 1996.
- [10] W. Teitelman, "A Display-Oriented Programmer's Assistant," *Computer*, 1977 pp. 39–50.
- [11] A. Goldberg and D. Robson, *Smalltalk-80: The Language*, Reading, MA: Addison-Wesley, 1989.
- [12] P. Fritzson, *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, New York: Wiley-IEEE Press, 2004.
- [13] M. M. Tiller, *Introduction to Physical Modeling with Modelica*, Norwell, MA: Kluwer Academic Publishers, 2001.
- [14] H. Elmqvist, S.-E. Mattsson, and M. Otter, "Modelica—A Language for Physical System Modeling, Visualization and Interaction," in *Proceedings of the 1999 IEEE Symposium on Computer-Aided Control System Design (CACSD'99)*, Hawaii, 1999 pp. 630–639.
- [15] P. Fritzson and V. Engelson, "Modelica—A Unified Object-Oriented Language for System Modeling and Simulation," in *Proceedings of the Twelfth European Conference on Object-Oriented Programming (ECOOP'98)*, Brussels, Belgium (E. Jul, ed.), Heidelberg, Germany: Springer-Verlag GmbH, 1998 pp. 67–90.
- [16] *Modelica—A Unified Object-Oriented Language for Physical Systems Modeling—Tutorial and Design Rationale*, Version 1.0, Linköping, Sweden: Modelica Association, (Sep 1997) www.modelica.org/documents.
- [17] *Modelica—A Unified Object-Oriented Language for Physical Systems Modeling—Language Specification*, Version 1.0, Linköping, Sweden: Modelica Association, (Sep 1997) www.modelica.org/documents.
- [18] *Modelica—A Unified Object-Oriented Language for Physical Systems Modeling—Tutorial and Design Rationale*, Version 1.4, Linköping, Sweden: Modelica Association, (Mar 2000) www.modelica.org/documents.
- [19] *Modelica—A Unified Object-Oriented Language for Physical Systems Modeling—Language Specification*, Version 2.1, Linköping, Sweden: Modelica Association, (Sep 2004) www.modelica.org/documents.
- [20] P. Fritzson, V. Engelson, and J. Gunnarsson, "An Integrated Modelica Environment for Modeling, Documentation and Simulation," in *Proceedings of the 1998 Summer Computer Simulation Conference (SCSC'98)*, Reno, Nevada, 1998.
- [21] M. Jirstrand, J. Gunnarsson, and P. Fritzson, "MathModelica—A New Modeling and Simulation Environment for Modelica," in *Proceedings of the Third International Mathematica Symposium (IMS'99)*, Linz, Austria, 1999.
- [22] H. Elmqvist, "A Structured Model Language for Large Continuous Systems," Ph.D. thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1978.
- [23] H. Elmqvist, D. Bruck, and M. Otter, *Dymola (Dynamic Modeling Laboratory) User's Manual*, Version 3.0. Ideon Research Park, SE-223 70, Lund, Sweden: Dynasim AB, 1996.
- [24] S.-E. Mattsson, M. Andersson, and K.-J. Åström, "Object-Oriented Modelling and Simulation," *CAD for Control Systems* (D. Linkens, ed.), New York: Marcel Dekker, Inc., 1993 pp. 31–69.
- [25] M. Andersson, "Object-Oriented Modeling and Simulation of Hybrid Systems," Ph.D. thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 1994.

- [26] P. Sahlin, A. Bring, and E. F. Sowell, "The Neutral Model Format for Building Simulation, Version 3.02," Technical Report, Department of Building Sciences, The Royal Institute of Technology, Stockholm, Sweden, June 1996.
- [27] A. P. J. Breunese and J. F. Broenink, "Modeling Mechatronic Systems using the SIDOPS+ Language," *Proceedings of the Third International Conference on Bond Graph Modeling and Simulation (ICBGM'97)*, Phoenix, Arizona, San Diego: SCS Publishing, Simulation Series, **20**(1), 1997 pp. 301–306.
- [28] T. Ernst, S. Jähnichen, and M. Klose, "The Architecture of the Smile/M Simulation Environment," *Proceedings of the Fifteenth IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, **6**, Berlin, Germany, 1997 pp. 653–658.
- [29] P. C. Piela, T. G. Epperly, K. M. Westerberg, and A. W. Westerberg, "ASCEND: An Object-Oriented Computer Environment for Modeling and Analysis: The Modeling Language," *Computers and Chemical Engineering*, **15**(1), 1991 pp. 53–72.
- [30] M. Oh, and C. C. Pantelides, "A Modelling and Simulation Language for Combined Lumped and Distributed Parameter Systems," *Computers and Chemical Engineering*, **20**, 1996 pp. 611–633.
- [31] M. Abadi and L. Cardelli, *A Theory of Objects*, Monographs in Computer Science (D. Gries and F. B. Schneider, eds.), New York: Springer-Verlag, 1996.
- [32] *Dymola (Dynamic Modeling Laboratory) User's Manual, Version 5.0*, Ideon Research Park, SE-223 70, Lund, Sweden: Dynasim AB, 2003.
- [33] D. E. Knuth, "Literate Programming," *The Computer Journal*, **27**(2) May 1984 pp. 97–111.
- [34] D. E. Knuth and S. Levy, *The CWEB System of Structured Documentation, Version 3.0*, Reading, MA: Addison-Wesley, 1994.
- [35] M. Otter, "Objektorientierte Modellierung mechatronischer Systeme am Beispiel geregelter Roboter," Ph.D. diss., Fortschrittberichte VDI, Reihe 20, Nr. 147, 1995.
- [36] M. Otter, H. Elmqvist, and F. E. Cellier, "Modeling of Multibody Systems with the Object-Oriented Modeling Language Dymola," *Nonlinear Dynamics*, **9**, 1996 pp. 91–112.
- [37] V. Engelson, H. Larsson, and P. Fritzson, "A Design, Simulation and Visualization Environment for Object-Oriented Mechanical and Multi-Domain Models in *Modelica*," in *Proceedings of the IEEE International Conference on Information Visualization (InfoVis'99)*, London, 1999 pp. 188–193.
- [38] P. Bonus, V. Engelson, and P. Fritzson, "Mechanical Models Translation and Simulation in *Modelica*," in *Proceedings of Modelica Workshop 2000*, Lund University, Lund, Sweden (Peter Fritzson, ed.), Lund, Sweden: Lund University, 2000, pp. 199–208.
- [39] V. Engelson, "Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing," Ph.D. thesis, Department of Computer and Information Science, Linköping University, Linköping, Sweden. Nr. 627, 2000.
- [40] StereoLithography Interface Specification—The STI Format, Valencia, CA: 3Dsystems Inc., 2000.
- [41] P. Fritzson, *MathCode C++*, Linköping, Sweden: MathCore Engineering AB, 1998 www.mathcore.com,
- [42] L. Saldamli and P. Fritzson, "A *Modelica*-Based Language for Object-Oriented Modeling with Partial Differential Equations," in *Shaping the Future with Simulation, Proceedings of the Fourth International EUROSIM Congress*, Delft, the Netherlands, 2001.

About the Author

Peter Fritzson, Ph.D., is a professor and director of the Programming Environment Laboratory (PELAB) at the Department of Computer and Information Science, Linköping University, Sweden. He currently holds positions as Chairman at MathCore Engineering, Chairman of the Scandinavian Simulation Society, Secretary of EuroSim, and Vice Chairman of the Modelica Association (www.modelica.org), an organization he helped to establish. He was previously affiliated with Sun Microsystems. Professor Fritzson has published twelve books and over a hundred scientific papers.

Peter Fritzson

PELAB—Programming Environment Laboratory

Department of Computer and Information Science

Linköping University, SE-581 83, Linköping, Sweden

petfr@ida.liu.se

www.ida.liu.se/labs/pelab

Modelica book home page with MathModelica examples:

www.mathcore.com/DrModelica

Free OpenModelica implementation:

www.ida.liu.se/labs/pelab/modelica/OpenModelica.html

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 “TheMathematicaJournal.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.

Classifieds

Classified advertising is provided for *Mathematica* users. Rates are available at www.mathematica-journal.com/advertising.

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 offers 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* programs, isolating problems in large *Mathematica* programs, implementing custom features, and writing complete programs based on user specifications.

More info: www.wolfram.com/services/training

Online Calculus Solver—calc101.com supplies automatic step-by-step solutions for derivatives, elementary integrals, matrix algebra, and polynomial operations.

More info: www.calc101.com



Are you up to the *Mathematica*[®] challenge?

Wolfram Research's *Mathematica* technology stands behind many of the most important scientific discoveries and technical innovations of the last two decades. In our continuing commitment to revolutionizing the technology and science of computation, we now have unique new opportunities in our world-class team at both junior and senior levels.

If you're skilled with *Mathematica*, clear-thinking, pragmatic, and excited about transforming the way today's most innovative people work, go to **career.wolfram.com** and get ready for the challenge of a lifetime.

An Equal Opportunity Employer



WOLFRAMRESEARCH

New Tools • New Science • New Technology

