Contents

Lecture 1: Introduction and Course Description

Sept. 5 2007

These notes and all course materials will be available at http://pruffle.mit.edu/3.016-2007. Students should bookmark this site and use it to download lecture notes, homework, and reading assignments for laboratories and lectures. The site will develop throughout the semester.

The web materials for 3.016 are revised each year. Previous years' notes, homeworks, and laboratories are available at `http://pruffle.mit.edu/3.016-2006` and may be useful to you; however, previous courses used MATHEMATICA® 5 and this year's course uses MATHEMATICA® 6.

Preface

Materials Science and Engineering is a discipline that combines knowledge of chemistry, mechanics, and physics and then applies them to the study of materials and their properties. It is a challenging and diverse enterprise—obtaining expertise in a large set of diverse subjects—but is one that will be very rewarding and fulfilling.

Mathematics is the *language* that binds together disparate topics in physics, engineering, and chemistry. Thus, this subject is for undergraduate materials scientists and engineers who wish to learn about the mathematics that is essential to their chosen field.

While it is possible to become an excellent materials scientist and engineer without some working knowledge of a large subset of mathematical topics, it is much easier to master this discipline with mathematics to guide you. Through mathematics, you will discover that some topics have similarities that are not obvious and, in fact, are not taught to you as *being* similar. Such similarities and analogies will make learning much, much easier—and I believe much more enjoyable.

MIT's Department of Materials Science and Engineering (MS&E) subjects have been designed with the philosophy that students will benefit more from a solid backgroun in, and a working knowledge of, the wide range math problem solving techniques that pertain specifically to MS&E, rather than a limited subset of topics (albeit at more depth and rigor) as

taught in a one-semester subject in a Mathematics department. It is reasonable to ask, "Is this subject a substitute for a a 'Linear Algebra' or 'Partial Differential Equations' from the Math department?" The answer is, "no, not entirely" this is not a replacement for a single subject that provides the rigorous foundations of Mathematics, and I encourage you to take such math classes in the future if you have time. This subject is designed to be very broad in scope and therefore its depth in any one topic is limited. However, you will learn to use *mathematics as a tool* to solve engineering problems in this course. And, you will learn math by applying it to familiar problems. I believe it will be easier and more interesting to take a Math department course *after* you finish 3.016.

I do believe very strongly that you will enjoy studying math more after taking this introduction and that the mathematical background you will receive this semester will make your Materials Science education richer and more rewarding.

I have designed this subject to help you learn as much essential math as possible in a short time. To this end, this subject has several unusual aspects that you will need to know.

3.016 Mathematical Software

Symbolic mathematical computer software is a tool used by almost every applied scientist. Such software helps produce results quickly, visualizes and documents the results, and minimizes the silly errors that creep into complicated mathematical manipulations. Although there are many other good choices, I have decided to use MATHEMATICA® as a vehicle for learning and doing mathematics. It has a fairly steep learning curve. Once learned however, it rewards the time investment with a powerful tool that, though not a replacement for mathematical understanding, will serve as an aid to help you think about, visualize, compute, and solve mathematics problems faster and more accurately. and packages.

MATHEMATICA® is available for all MIT students, both on Athena (free) and via licenses for personal laptop and desktop machines (There is a modest fee for student licenses, but it is a useful investment for other subjects). The process to access MATHEMATICA® on Athena and the steps to download a license will be explained to you; the pertinent website is http://web.mit.edu/is/products/vsls/. You will need MATHEMATICA® for your first homework set and laboratory, and you should try to get it loaded and working very soon. If you have a laptop, I suggest that you install MATHEMATICA® on it as soon as possible. If you don't have a laptop, I can write you a letter so that you can get one through MIT's laptop-loaner program: http://web.mit.edu/lcp/.

3.016 Home

Full Screen

Close

Quit

Note, the purpose of this course is not to teach MATHEMATICA® . I will teach you how to use it as a tool to learn and solve problems. Thus, you will have a fairly good working knowledge of the software and will have the elements—if you invest the time and work—to become a MATHEMATICA® power-user in the future. In this course, you will learn enough basics so that you be able to solve engineering problems faster, more accurately, and more beautifully, than your MATHEMATICA® -naive cohorts.

As of June 2007, there was a new release of MATHEMATICA® . Version 6.0 differs considerably from its predecessors, and its graphics and graphical interactivity have been greatly enhanced. At the time of this writing, MATHEMATICA® –6.0 is much bigger (you'll need more memory on your computer to run applications effectively), and limited memory can make it appear to be slower in some cases. The documentation is now more difficult to use for beginners, and I find that it can be difficult to find the useful tutorials and guides.

To help you find the Mathematica tutorials, you should download the file http://pruffle.mit.edu/3.016/Help-Palette-Builder.nb once you have MATHEMATICA® running, and open this as a MATHEMATICA® notebook (there are instructions to evaluate a cell and install the palette that gets built).

Laboratories in this course will be performed using MATHEMATICA® . Laboratory assignments must be completed during the laboratory period and an electronic copy of a MATHEMATICA® notebook for each laboratory must be emailed to the instructor and the TA.

3.016 Examination Philosophy

Tests and exams are powerful motivators to get students to take a subject seriously, but I believe that working through homework problems better promotes learning, particularly for self-motivated students.

Therefore, there will be no exams, tests, or quizzes in 3.016. Your grade will be based on your homeworks and laboratories. These will graded carefully (described below). A homework calendar is given below.

3.016 Homework

The purpose of the homework is to help you solidify your understanding of *mathematics applied to engineering and science problems* by working through examples. Some examples will be exercises in mathematics; others will be exercises in application of mathematics for solving engineering and science problems. I encourage you to use MATHEMATICA® to solve your homework problems, and you may turn in solutions as printed MATHEMATICA® notebooks. Nevertheless to appreciate what symbolic mathematics programs can do for you, there will be some exercises that I will ask you to do with pencil and paper. Nevertheless, there is no harm in checking your "by-hand" results with MATHEMATICA® .

When you do homework, you are not under the potentially menacing eyes of an exam proctor. This means that you can receive help in the form through various resources, by asking experts, or interacting with fellow students.

Resources Go to the library, or refer other media to find solutions to problems. It is good practice and you will learn quite a bit by doing so. I recommend that you first attempt to find a solution before going to the library—not only will it help you appreciate the solution, it will also make your search a bit easier! *However, if your solutions derive from any source that you have found, then you must cite that source.* **Plagarists will receive NO CREDIT for their work.**

If any two or more handed-in homework-assignments appear to be duplicated without proper citation, then ALL of the homework-assignments will receive NO CREDIT. This applies to the "original document" as well.

Experts By all means, consult with experts on your homework. It is a good idea to do this as long as you understand what *you* turn in.

Classmates This is the best choice of all. I think it is both inevitable and beneficial to give and receive help. Cooperating on homework will help you learn to communicate your ideas and begin to appreciate the difficulties and rewards of teamwork.

As explained below, the homework assignments in 3.016 will be, in part, cooperatively oriented.

You will find that you are more busy some weeks than others and relying on a classmate during a busy week can be a life-saver. However, if you start slacking off and don't hold up your end of the bargain when you are able, you will engender resentment,

as well as endanger professional and friendly relationships. I leave it to your own conscience to play fairly and contribute when you can and, while understanding that everyone experiences different kinds of pressures, to be forthright and honest with others who do not contribute consistently.

However, you must contribute something to every problem on every group assignment. If you don't contribute, you must state it. If I find out later that you didn't contribute to a problem, but took credit for it, then you will receive NO CREDIT for the entire assignment.

It is fairly easy for the instructor to ascertain who is slacking and who is not. I can't say that my good opinion has any particular value; but keep in mind, that slackers may have hard time regaining my good opinion.

Homework cooperation has a potential downside because you all receive individual grades. We will attempt to mitigate this downside by dividing the homework into two parts:

Group For each homework set, a few problems will be designated as *Group Exercises*. For these problems, the entire group will turn in *one homework*. Every member of the group *who puts their name on the turned-in assignment* will receive exactly the same credit for the homework grade (unless it has been determined later that the student did not contribute).

Homework groups will be *assigned* with each homework set. The groups will change from week to week and the members will be assigned randomly. Each group will be assigned a homework leader who will be responsible for arranging meetings and turning in the homework.

For each group problem, I suggest that two students work together to develop the MATHEMATICA® code. A third student should comment the code, and another student should be responsible for writing descriptions and interpretations of the results. Yet another student should be responsible for improving the graphics. These responsibilities will should be rotated for each problem in the homework set.

By putting each individual's name on a homework assignment, the group verifies that each indicated person has contributed to the assignment. By putting your own name on the group's turned-in assignment, indicates that you have reviewed *all* of the assignment; if questioned, each person should be able to describe how each problem was done. MIT's policy on academic integrity is also the policy for 3.016.

Individual Each problem set will contain a few problems for each student to complete individually. These problems will come out of the textbook and tend to be a bit easier than the group exercises. They are designed to maintain a sufficient

amount of currency and emphasize that reading the textbook is an essential part of this course. The problems will demonstrate the benefits of having symbolic mathematics software.

If you turn in work that you did not do, and do not attribute the solution to its rightful author(s), then you are plagiarizing. As a first assignment in this course, every one of you should read MIT's policy on academic integrity (html) or (pdf) immediately. There is also an on-line learning module http://web.mit.edu/uaap/learning/modules/acadintegrity/ to help you understand the philosophy of integrity and your responsibility as an MIT student.

With your first homework, print out the first page of this handbook and sign it—by doing so you imply that you have read it. *Students who do not sign this document will receive an incomplete for the course.*

3.016 Laboratory

There will be a 3.016 laboratory each friday that 3.016 meets (see calendar below). The labs will be practical and will focus on using MATHEMATICA® effectively.

There will be assigned readings from the MATHEMATICA® help browser that comes with the software. *You should always do this reading before the laboratory, or you may not be able to* FINISH YOUR ASSIGNMENT AND TURN IT IN BEFORE THE END OF THE LABORATORY IN ORDER TO GET CREDIT.

If you stay current in the course material and keep up with the homework assigments, you should have no difficulty doing the laboratory assignments *if you do the pre-assigned reading.*

You should have your own laptop running MATHEMATICA® and bring it to the lab with you. Laptop loaners from MIT computing services are available.

Grades

As stated above, all of the final grade will depend on the homework and the laboratory assignments. There is no fixed average grade for this course; the average will depend on the entire class performance. However, if your homework grades and your laboratory reports are consistently within the top quartile, then it is extremely likely that you will receive an **A**. Your turned-in homework-assignments will be ranked from *Best* to *Least Best Homework*. A decision will be made regarding how many points (out of a possible 100) the *Least Best Homework* deserves, and the homework scores will then be interpolated between a score of 99 for the *Best Homework* and that of the *Least Best*.

Homeworks will be evaluated on the basis of:

Accuracy The solution must be a reasonable and correct answer to the homework question.

Exposition The solution must clearly show the reasoning that was utilized to find it and the method of solution should be clearly apparent. Exegetic solutions will be ranked higher.

Beauty Good solutions will often require graphics that, with care, can beautifully explain the solution. The layout of the page, the quality of the supporting prose, the clarity of the graphics, and all that "je ne sais quoi" is fairly subjective but very important. The grader will include a judgment of your craft and art.

Observation Supplemental observations provide aids in understanding and demonstrate mastery of a topic. An example of a supplemental observation might be something like, "Note that in the limit of long times, that the total concentration goes to zero. This is sensible because the boundary condition on mass flux is directed outward everywhere on the finite domain."

Laboratories will be graded on their completeness, demonstrated mastery of MATHEMATICA® for that assignment, and exposition.

Note that there will be times when you have two homework sets pending—this is done so that you can arrange your time conveniently.

3.016 Home

Full Screen

Close

Quit

Homework Calendar and Weighting

Homework Schedule			
Homework Assignment	**Available After**	**Available By**	**Due Date**
Set 1	Lect. 1	5 Sept.	14 Sept.
Set 2	Lect. 4	12 Sept.	28 Sept.
Set 3	Lect. 9	3 Oct.	19 Oct.
Set 4	Lect. 14	17 Oct.	9 Nov.
Set 5	Lect. 17	31 Oct.	21 Nov.
Set 6	Lect. 22	14 Nov.	7 Dec.

Late Policy

Students will be allowed to turn in one homework up to 3 days late, for the individual portion only. No second late homework will be allowed without formal documentation about an unforeseeable emergency. No late group homework portions will be accepted—no exceptions.

Laboratory assignments must be turned in during the laboratory period. You must show documentation of unforeseeable emergencies that prevent you from attending a laboratory period. Any missed laboratories must be made up by special arrangement. If for some reason, you cannot complete a laboratory during the laboratory period, you should send a paragraph explaining why you could not finish.

It is your responsibility to do the assigned reading before the laboratory.

Textbook

We will use a fairly general textbook on applied mathematics (E. Kreyszig, *Advanced Engineering Mathematics*, ninth ed., J.W. Wiley, ≈ 1200 pages). You'll notice that reading assignments do not follow the table of contents—while I like the book, there are pedagogical reasons for studying mathematics in the sequence I chose to follow in this subject. Extra material pertaining to materials science specifically will be created and placed on the web.

I have identified 66 sections of the book (330 pages in total) for required reading. The readings for each lecture will appear in the Lecture Notes and posted on the course web-site at: http://pruffle.mit.edu/3.016-2007. I hope you will keep up with the reading—I think it would be wise to give the material a cursory reading prior to the lecture and then read it more carefully before starting the homework.

This course is designated as a 12 (*3-1-8*) unit subject[1] Time spent awake during lectures and recitations is less than half of your job—reading and doing homework is the greater part.

Lecture Notes

Lecture notes (like these) will be available for you to print out for each lecture. The lecture notes will be available at: http://pruffle.mit.edu/3.016-2007. These will supplement (not replace) the textbook. The lecture notes also serve as a guide to help the student understand what parts of the text are considered more relevant or important.

The specific purpose of the notes is to provide neatly typeset equations and graphics that will be used in the lecture along with a few observations. This will eliminate the time required to write and draw, perhaps a bit sloppily, for you in your notes and for me on the blackboard.

The lecture notes will have reading assignments printed at the beginning of each lecture; they will look like this:
Kreyszig **6.1, 6.2, 6.3, 6.4** (pages: 304–309, 312–318, 321–323, 331–336). Part of the units for this course involve reading.

[1] Units at MIT are assigned under the following schema: *lec-lab-out* where *lec* is the number of lecture/recitation hours, *lab* is the number of laboratory hours, and *out* is the number of outside (reading, preparation, homework) hours per week. One MIT unit represents about 14 hours of semester work on the average.

3.016 Home
Full Screen
Close
Quit

You are receiving an expensive education—you should strive to make your education valuable by doing all the required reading. Your intellect will profit even more by doing outside reading.

Those concepts that are fundamental to this course will be presented in lectures by the lecturer (or in the form of welcome questions and points of clarification by the students) and some explanatory notes will be written upon the blackboard.

The notes will have places for you to fill in auxiliary discussion and explanation. Those places will look like this: Lines appear on formats for printed notes only You can use these notes in several ways. You could print them out before lecture, and write your own lecture notes directly on them during the lecture. You could take lecture notes on your own paper and then neatly copy them onto a printout later. You could print them before lecture and write on them rapidly and then copy—neatly and thoughtfully—your notes onto a freshly printed set. I recommend the latter for effective learning and the creation of a set of notes that might provide future reference material—but do whatever works for you.

The lecture notes will also refer to MATHEMATICA® notebooks available on the 3.016 website for downloading. These notebooks will be used as MATHEMATICA® sessions during the lectures to illustrate specific points and provide examples for you to help solve homework problems.

References to MATHEMATICA® notebooks look like the ones given at the end of this lecture's notes in section 1.

These examples will serve as place-holders in the lecture note when we switch from chalkboard and/or projected display of the notes to a live MATHEMATICA® session.

Lecture and Laboratory Calendar

This calendar will be updated throughout the semester. Students should consult this calendar weekly to obtain the required reading assignments for the laboratory.

Week of 3–7 September

Lectures		
	Topics	**Reading**
M 09/03	Labor Day, No Lectures	
W 09/05 Lect. 1	Course organization and introduction to Mathematica, Common Errors for Beginners	Course Notes and Mathematica Notebook *I*
F 09/07 Lect. 2	Introduction to Mathematica, assignment and evaluation, rules and replacement, basic calculus and plotting, lists and matrices, getting help	Course Notes and Mathematica Notebook *II*
Laboratory		
F 09/07 Lab 1 (Not Graded)	"First Five Minutes with Mathematica"	*Mathematica Help Browser* **Documentation Center**

Homeworks		
Homework Set	Available	Due Date
1	**Wednesday 5 Sept.**	Friday 14 Sept.

3.016 Home

Full Screen

Close

Quit

Week of 10-14 September

Lectures		
	Topics	**Reading**
M 09/10 Lect. 3	Mathematica programming: functions and patterns, localized variables, logical switches, recursion; Graphics: plotting lists of data, examples	Course Notes and Mathematica Notebook *III*
W 09/12 Lect. 4	Mathematica: symbolic and numerical operations, operations on expressions, solving equations, numerical solutions, file input and output, using packages	Course Notes and Mathematica Notebook *IV*
F 09/14 Lect. 5	Mathematica: overview of graphics, animation, interaction, graphics primitives, complete worked examples Course Notes and Mathematica Notebook *V*	

Laboratory		
F 09/15 Lab 2	Symbolic calculations and plotting	*Mathematica Help Browser* **Mathematica Tutorial Overviews: "Input and Output in Notebooks," "Building Up Calculations," "Algebraic Calculations," "Calculus," ; Functions: Integrate, Simplify, NIntegrate, Plot, Plot3D, ContourPlot**

Homeworks		
Homework Set	Available	Due Date
1	Wednesday 5 Sept.	**Friday 14 Sept.**
2	**Wednesday 13 Sept.**	Friday 28 Sept.

3.016 Home

Full Screen

Close

Quit

Week of 17-21 September

Lectures		
	Topics	**Reading**
M 09/17 Lect. 6	Linear algebra: matrix operations, interpretations of matrix operations, multiplication, transposes, index notation	*Kreyszig* **7.1, 7.2, 7.3, 7.4** (pages: 272–276, 278–286, 287–294, 296–301)
W 09/19 Lect. 7	Linear algebra: solutions to linear systems of equations, determinants, matrix inverses, linear transformations and vector spaces	*Kreyszig* **7.5, 7.6, 7.7, 7.8, 7.9** (pages: 302–305, 306–307, 308–314, 315–323, 323–329)
F 09/21 Lect. 8	Complex numbers: complex plane, addition and multiplication, complex conjugates, polar form of complex numbers, powers and roots, exponentiation, hyperbolic and trigonometric forms	*Kreyszig* **13.1, 13.2, 13.5, 13.6** (pages: 602–606, 607–611, 623–626, 626–629)
Laboratory		
F 09/21 Lab 3	Solving linear systems of equations	*Mathematica Help Browser* **Mathematica Tutorial Overview "Linear Algebra (Introduction, Matrix and Tensor Operations, Matrix Multiplication, Solving Linear Systems)" , Functions: Inverse, Transpose, Eigensystem**

Week of 24-28 September

3.014 Laboratory Week: 3.016 does not meet.

Homeworks		
Homework Set	Available	Due Date
2	Wednesday 12 Sept.	**Friday 28 Sept.**

Week of 1–5 October

Lectures		
	Topics	**Reading**
M 10/01	No Lecture	
W 10/03 Lect. 9	Matrix eigenvalues: eigenvalue/eigenvector definitions, invariants, principal directions and values, symmetric, skew-symmetric, and orthogonal systems, orthogonal transformations	*Kreyszig* **8.1, 8.2, 8.3** (pages: 334–338, 340–343, 345–348)
F 10/05 Lect. 10	Hermitian forms, similar matrices, eigenvalue basis, diagonal forms	*Kreyszig* **8.4, 8.5** (pages: 349–354, 356–361)
Laboratory		
F 10/05 Lab 4	File input/output, plotting data	*Mathematica Help Browser* **Mathematica Tutorial Overview "Files and External Operations (Secs 1-3, 6)"; Functions: Dimensions, Append, AppendTo, Do, Mean, StandardDeviation, ListPlot, Table, MultipleListPlot, Fit**

Homeworks		
Homework Set	Available	Due Date
3	**Wednesday 3 Oct.**	Friday 19 Oct.

Week of 8–12 October

Lectures		
	Topics	**Reading**
M 10/08	Holiday, No Lectures	
W 10/10 Lect. 11	Vector calculus: vector algebra, inner products, cross products, determinants as triple products, derivatives of vectors	*Kreyszig* **9.1, 9.2, 9.3, 9.4** (pages: 364–369, 371–374, 377–383, 384–388)
F 10/12 Lect. 12	Multi-variable calculus: curves and arc length, differentials of scalar functions of vector arguments, chain rules for several variables, change of variable and thermodynamic notation, gradients and directional derivatives	*Kreyszig* **9.5, 9.6, 9.7** (pages: 389–398, 400–403, 403–409)
Laboratory		
F 10/12 Lab 5	Statistics, fitting data, error analysis	*Mathematica Help Browser* **Mathematica Documentation: "guide/CurveFittingAndApproximateFunctions"; Functions: Fit, FindFit**

3.016

Week of 15–19 October

Lectures		
	Topics	**Reading**
M 10/15 Lect. 13	Vector differential operations: divergence and its interpretation, curl and its interpretation	*Kreyszig* **9.8, 9.9** (pages: 410–413, 414–416)
W 10/17 Lect. 14	Path integration: integral over a curve, change of variables, multidimensional integrals	*Kreyszig* **10.1, 10.2, 10.3** (pages: 420–425, 426–432, 433–439)
F 10/19 Lect. 15	Multidimensional forms of the Fundamental theorem of calculus: Green's theorem in the plane, surface representations and integrals	*Kreyszig* **10.4, 10.5, 10.6, 10.7** (pages: 439–444, 445–448, 449–458, 459–462)
Laboratory		
F 10/19 Lab 6	Graphical representations in three and higher dimensions	*Mathematica Help Browser* **Mathematica Tutorial Overview: "Graphics and Sound (secs 1–7)"**

Homeworks		
Homework Set	Available	Due Date
3	Wednesday 3 Oct.	**Friday 19 Oct.**
4	**Wednesday 17 Oct.**	Friday 9 Nov.

Week of 22–26 October

3.014 Laboratory Week: 3.016 does not meet.

Week of 29 September–2 November

Lectures		
	Topics	**Reading**
M 10/29 Lect 16	Multi-variable calculus: triple integrals and divergence theorem, applications and interpretation of the divergence theorem, Stokes' theorem.	*Kreyszig* **10.8, 10.9** (pages: 463–467, 468–473)
W 10/31 Lect. 17	Periodic functions: Fourier series, Interpretation of Fourier coefficients, convergence, odd and even expansions	*Kreyszig* **11.1, 11.2, 11.3** (pages: 478–485, 487–489, 490–495)
F 11/02 Lect. 18	Fourier theory: complex form of Fourier series, Fourier integrals, Fourier cosine and sine transforms, the Fourier transforms	*Kreyszig* **11.4, 11.7, 11.8, 11.9** (pages: 496–498, 506–512 513–517, 518–523)
Laboratory		
F 11/02 Lab 7	Review of Mathematica functions, programs, and graphics	*Mathematica Help Browser* **Mathematica Tutorial Overview: "Functions and Programs"**

Homeworks		
Homework Set	Available	Due Date
5	**Thursday 31 Oct.**	Wednesday 21 Nov.

Week of 5–9 November

Lectures		
	Topics	**Reading**
M 11/05 Lect 19	Ordinary differential equations: physical interpretations, geometrical interpretations, separable equations	*Kreyszig* **1.1, 1.2, 1.3** (pages: 2–8, 9–11, 12–17)
W 11/07 Lect. 20	ODEs: derivations for simple models, exact equations and integrating factors, the Bernoulli equation	*Kreyszig* **1.4, 1.5** (pages: 19–25, 26–32)
F 11/09 Lect. 21	Higher order differential equations: homogeneous second order, initial value problems, second order with constant coefficients, solution behavior	*Kreyszig* **2.1, 2.2** (pages: 45–52, 53–58)
Laboratory		
F 11/09 Makeup Lab or Extra Credit	Possibility to make up missed labs and/or review lab exercise for extra credit	

Homeworks		
Homework Set	Available	Due Date
4	Wednesday 17 October	**Friday 9 Nov.**

3.016 Home

Full Screen

Close

Quit

Week of 12–16 November

Lectures		
	Topics	**Reading**
M 11/12	Holiday, No Lectures	
W 11/14 Lect. 22	Differential operators, damped and forced harmonic oscillators, non-homogeneous equations	*Kreyszig* **2.3,2.4, 2.7** (pages: 59–60, 61–69, 78–83)
F 11/16 Lect. 23	Resonance phenomena, higher order equations, beam theory	*Kreyszig* **2.8, 2.9, 3.1, 3.2, 3.3** (pages: 84–90, 91–96, 105–111, 111–115, 116–121)
Laboratory		
F 11/16 Lab 8	Solutions to ordinary differential equations	*Mathematica Help Browser* **Mathematica Tutorial Overview "Calculus (sec: Differential Equations)". "DSolve"; Functions: DSolve, NDSolve, NIntegrate**

Homeworks		
Homework Set	Available	Due Date
6	**Wednesday 14 Nov.**	Friday 7 Dec.

3.016 Home

Full Screen

Close

Quit

Week of 19–23 November

Lectures		
	Topics	**Reading**
M 11/19 Lect. 24	Systems of differential equations, linearization, stable points, classification of stable points	*Kreyszig* **4.1, 4.2** (pages: 131–135, 136–139)
W 12/21 Lect. 25	Linear differential equations: phase plane analysis and visualization	*Kreyszig* **4.3, 4.4** (pages: 139–146, 147–150)
F 11/23	Holiday, no 3.016 lecture	

Homeworks		
Homework Set	Available	Due Date
5	Wednesday 31 Oct.	**Wednesday 21 Nov.**

3.016

Week of 26–30 November

3.014 Laboratory Week: 3.016 does not meet.

Week of 3–7 December

Lectures		
	Topics	**Reading**
M 12/03 Lect. 26	Solutions to differential equations: Legendre's equation, orthogonality of Legendre polynomials, Bessel's equation and Bessel functions	*Kreyszig* **5.3, 5.5, 5.6** (pages: 177–180, 189–197, 198–202)
W 12/05 Lect. 27	Sturm-Louiville problems: eigenfunction, orthogonal functional series, eigenfunction expansions	*Kreyszig* **5.7, 5.8** (pages: 203–208, 210–216)
F 12/07	3.014 Laboratory continues, No more Maths lectures	

Homeworks		
Homework Set	Available	Due Date
6	Wed. 14 Nov.	**Friday 7 Dec.**

3.016 Home

Full Screen

Close

Quit

Week of 10–14 December

3.014 Laboratory Week: 3.016 does not meet.

Beginners to MATHEMATICA

Beginners to MATHEMATICA® tend to make the same kinds of mistakes. I've been collecting a list of such mistakes and present them to you as a reference tool.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Common Mathematica Mistakes

A list of common beginner MATHEMATICA mistakes. The entries here are typical **mistakes**. I welcome input from others to might add to this list

1-7 are examples of confusing usages of parentheses (—), curlies {—}, and square brackets [—]. Generally, parentheses (—) are for logical grouping of subexpressions (i.e., `(a+b)/(a-b)`); curlies {—} are for forming lists or iteration-structures, single square brackets [—] contain the argument of a function (i.e., `Sin[x]`), double square brackets [[—]] pick out parts of an expression or list.

Examples of Common Mistakes!

```
Cos (k x)
```
1

```
Plot[Sin[x], (x, 0, π)]
```
2

```
Sort[(x, y, z)]
```
3

$\left\{\frac{\sqrt{2}}{2}\right\}$ {a, b, c}

4

```
SomeList = {a, b, c, d};

SomeList[1]
```
5

$\left[\left(z^2 + y^2\right) c + b\, y^3\right] a$

6

```
Exp[[1]]
```
7

```
arccos[1]
```
8

```
Arccos[1]
```
9

```
MyFunction[x, y, z] := Sin[x] Sin[y] Sin[z]
```
10

```
MyFunction[π, π / 2, 0]
```
11

```
x = π / 2;
AbsSin[x_] = Abs[Sin[Abs[x]]]
```
12

```
Plot[AbsSin[z],
 {z, -2 π, 2 π}, PlotStyle -> Thick]
```
13

1: *Probable error:* The parenthesis do not call a function, but would imply multiplication instead.

2: *Error:* The plot's range should be in curlies {—}.

3: *Error:* `Sort` should be called on a list, which must be formed with curlies—not parenthesis.

4: *Probable error:* If the intention was to multiply the list by a constant, then the first set of curlies turned the constant into a list, not a constant.

5: *Probable error:* If the intention was to extract the first element in the list, then double square brackets are needed (i.e., [[—]]).

6: *Error*: brackets cannot be used for grouping, use parentheses instead.

7: *Probable error:* The double brackets do not make a function call.

8–9: *Probable error:* MATHEMATICA® is case sensitive and functions are usually made by concatenating words with their first letters capitalized (e.g., `ArcCos`).

10: Functions are *usually* created designed with patterns (i.e., x_, y_) for variables. This is an error if `x` is a defined variable. This line is correct in using the appropriate *delayed assignment* :=.

12: *Probable error:* Here a function is defined with a direct assignemt (=) and **not** *delayed assignment* :=. Because `x` was defined previously, the function will not use the current value of `x` in future calls, but the old one.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Common Mathematica Mistakes

(continued) list of common beginner MATHEMATICA® mistakes. The entries here are typical **mistakes** involving: the difference between assignment (=) and logical equality (==); forgetting commas; and, inadvertantly reusing a defined variable.

Common Mistakes!

The difference between assignment = and equality testing ==

```
Solve[{h = 3 p + 4 q, k = 5 p - q}, {p, q}]
```
1

```
δ == 24
```
2

Not using enough commas

```
Plot[Sin[x + Exp[-x]] {x, 0, Pi}]
```
3

Forgetting that a variable has been defined

```
A = e^(-1.2/(k 373))
```
4

Practical Advice 1:
Clear Variables

```
Clear[k];
A = e^(-1.2/(k 373))
```
5

Practical Advice 2:
Second to last resort, clear everything

```
Clear["Global`*"];
```
6

Practical Advice 3:
Last resort, kill the kernel and restart it Use menu: Evaluation

1: *Probable error:* In the first line, assignments (=) are used instead of the double equals (==) which is a *logical equality*.

2: *Probable error* Assignment of the symbol δ was probably intended by here a logical equality is queried (i.e., Is δ the same as 24?) and no value is assigned to δ.

3: *Error* Commas separate arguments in functions like `Plot` that require at least two arguments.

4: *Probable error:* The symbol `k` was inadvertently assigned in the first line, and its value is still known to MATHEMATICA® .

5: Practical advice is to clear the variable definitions with `Clear`.

6: More powerful practical advice, but slight overkill, is to clear all user-defined variables. As a last resort when everything seems awry, kill the kernel with the menu and restart it. This starts up a new MATHEMATICA® session, but does not destroy the text in the `Notebook`.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Common Mathematica Mistakes

(continued) list of common beginner MATHEMATICA® mistakes. The entries here are typical **mistakes** associated with using functions that are defined in Packages; assigning a variable to a formatted expression; and, using `Evaluate` withing `Plot`.

Common Mistakes!

```
Histogram3D[
 Table[{RandomReal[], RandomReal[]}, {5}]]
```
1

```
<< Histogram`
```
2

```
SetDirectory[$InstallationDirectory]
FileNames[]
```
3

```
SetDirectory["AddOns"]
FileNames[]
```
4

```
SetDirectory["Packages"]
FileNames[]
```
5

```
<< Histograms`
```
6

```
Histogram3D[
 Table[{RandomReal[], RandomReal[]}, {20}]]
```
7

Mistake: Using formatting commands in assignments

```
mymat = {{1, 3, 7},
         {3, 2, 4},
         {7, 4 , 3}} // MatrixForm
```
8

```
mymat = { {1, 3, 7},
          {3, 2, 4},
          {7, 4 , 3}};
mymat // MatrixForm
```
9

```
Eigenvalues[mymat] // N
```
10

Not using Evaluate : slow and monochrome.

```
Plot[Table[LegendreP[i, z], {i, 1, 11, 2}],
 {z, -1, 1}, PlotStyle → Thickness[0.01]]
```
11

Using Evaluate : Fast and multicolored.

```
Plot[Evaluate[
  Table[LegendreP[i, z], {i, 1, 11, 2}]],
 {z, -1, 1}, PlotStyle → Thickness[0.01]]
```
12

1: Some of the less-used MATHEMATICA® functions are defined in *Mathematica Packages* and are not loaded automatically when MATHEMATICA® is started.

2: *Probable error:* `Histogram3D` is defined in `Histograms`

3–6 Demonstrate a method to find the names of the installed MATHEMATICA® packages. The current version of MATHEMATICA® 's help-browser (6.0) doesn't provide a way to find them. *It is probably a good idea to* `Clear` *the definition of a function like* `Histogram3D` *if you use it before loading its package.* `Clear` *before reading in the package.*

8: *Probable error:* here the formatting becomes part of the variable assignment. In this case, a `MatrixForm` of a matrix *is not* a matrix and so matrix operations are not defined (i.e., `EigenValues` would not produce the expected result).

9: Practical advice is to separate the definition from the display of the assigned variable. Here a matrix is defined; its `MatrixForm` is display, and `Eigenvalues` of the matrix can be calculated.

10: Some functions, such as `Plot`, evaluate their arguments in a round-about way. This produced an error in pre-6.0 versions of MATHEMATICA® . However, in 6.0, not using `Evaluate` makes the time of the compuation long and will not produce a nice multi-colored set of curves.

11: If a computationally intensive function is not doing what you expect, then try to wrap an expression in an `Evaluate` function—in this case it will tell `Plot` that it is operating on a list of particular size and produce a different color for each curve.

Lecture 2: Introduction to Mathematica

Expressions and Evaluation

There are very many ways to learn how to use MATHEMATICA® . Nearly all of the best ways involve performing examples from the very beginning. That is how we are going to start—with examples. Using MATHEMATICA® 's *FrontEnd* you may execute a command by pressing Shift-Enter; simply pressing Enter tells MATHEMATICA® 's that you merely wish to have a "carriage return" on the screen.

Mathematica's syntax will feel fairly natural after a while. Use the following notebook to get started. Execute a few commands until you get a sense for what output MATHEMATICA® will produce; try editing the commands; try to make MATHEMATICA® do something strange—just try playing with it and you will soon get the hang of what is going on.

One way to use MATHEMATICA® is simply as a calculator that allows symbols to get carried along. MATHEMATICA® will usually try to resolve every symbol and return precise information about it. If something is undefined to MATHEMATICA® , it simply returns it as a symbolic expression.

A number is not returned until all of the symbols in an expression are defined as numbers. MATHEMATICA® will try to be exact—it does not calculate $\frac{1}{3}+\frac{1}{2}$ by adding $0.33333\cdots+0.5=0.83333\ldots$, it has an algorithm for adding rational numbers and gives $\frac{5}{6}$.

Getting Started

There are a variety of ways to get MATHEMATICA® started and these are specific to the operating system your computer uses. A license must be purchased to run MATHEMATICA® code, but free MATHEMATICA® -display tools can be obtained from Wolfram.

The *FrontEnd* is the graphical interface between the user and MATHEMATICA® —you arrange your MATHEMATICA® input, sometimes with text-like comments, in the FrontEnd. The user must request the FrontEnd to pass something to MATHEMATICA® 's *kernel*, by pressing Shift-Enter. The kernel is the resident symbolic algebra software engine behind MATHEMATICA® .

The appearance of the FrontEnd depends on either provided or user-designed *StyleSheets*. The StyleSheet for this course can be downloaded from the course website. The course style is particulary ugly—it is hoped that this will provide an incentive for students to create their own style.

There is also a useful notebook to help you build a *Palette* to find documentation, tutorials, overviews, and information on MATHEMATICA® packages. When you have MATHEMATICA® running, you should download this Help-Palette-Builder.nb, open it up as as a MATHEMATICA® notebook and follow the instructions to install the Palette that gets built.

3.016 Home

Full Screen

Close

Quit

Lecture 02 MATHEMATICA® Example 1

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Basic Input and Assignment

The methods of assigning symbols (`SomeVariable`) to expressions via `SomeVariable = expr`. The `expr` can contain other symbolic variables, functions, programs, graphics, and many other things. There are important differences between exact (symbolic) objects and numerical objects. Logical equalities (`==`) are not assignments, but are Boolean operations.

Assigning values to symbols

```
a = 4 π / 3                                   1
UnitSphereVolume = a                          2
2 a                                           3
ANewVariable = (2 a + b)^2                    4
ANewVariable^2                                5
b = 4 (3.14159265358979) / 3                  6
UnitSphereNumericalVolume = b                 7
ANewVariable                                  8
```

Differences between exact expressions and numerical expressions

```
UnitSphereVolume - UnitSphereNumericalVolume  9
a - 4 ArcCos[-1] / 3                          10
a - 4 ArcCos[-1.0] / 3                        11
2 Pi - 2 (3.141519)                           12
N[5 / 6]                                      13
```

Distinction between Equality (= =) and Assignment (=)

```
a == 4 ArcCos[-1] / 3                         14
a == 4 (3.14159) / 3                          15
```

1: A symbol is assigned to an expression with an equals sign `=`. Some symbols, such as π, are already defined—in MATHEMATICA® it is *exactly* the ratio of a circle's circumference to its diameter. Here, `a` is a symbol that could represent, for example, the volume of a sphere with radius 1—*and not an approximation depending on how many digits are used to numerically represent* π.

2: In my opinion, the variable `a` is not a very good name. We might forget what it represents, or try to use it again in a different context. I think it is much better to use descriptive names, such as `UnitSphereVolume`. Here, because there is an assignment in `UnitSphereVolume = a`, MATHEMATICA® tries to see if there are any other assignments associated with the right-hand-side, and if there are it uses them until all possible assignments have been made.

3: Because no assignment was made to `a` just above, its value is not changed.

4: The RHS in an assignment (here to `ANewVariable`) can contain unassigned symbols.

6–7: Here, the symbols `b` and `UnitSphereNumericalVolume` are assigned to an approximation to the unit sphere volume.

8: Note that, because `ANewVariable` contains `b`, the assignment of `b` above is reflected in the current value of `ANewVariable`: MATHEMATICA® will check to see if any symbol being output has been assigned.

9: To show the difference between the numerical approximation of π and the symbol π, subtraction shows that the difference is a very very small number.

10: Some functions can behave as *exact* if their values can be expressed exactly: here `ArcCos`[-1] is *exactly pi*.

11: Notice that the output here is different, showing that `ArcCos`[-1.0] has been replaced with a numerical representation because the function was executed on a numerical object.

14: The operator `==` tests to see if the LHS (left-hand side) and the RHS (right-hand side) can be determined to be equal, in which case it returns true.

15: If `==` can do so, it will return false if the two sides are not equal; otherwise if it can't say whether true or false, it will just return the statement itself.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Building Expressions and Functions and Operations on Expressions

Sometimes it is easier to build up complicated expressions by entering shorter subexpressions beforehand. There are usually many ways to do the same thing in MATHEMATICA® , and this is demonstrated for functions. As you begin, pick the most simple method that works. Someday later you can pick up the alternative methods—they can be useful in advanced usage.

Mathematica Functions

```
a = 1 / Exp[x]                                          1
b = Cos[x]                                              2
c = (a + b) ^ 2                                         3
```

Alternative Syntax for Functions (There are many ways to do the same thing)

```
AnotherVersionofb = x // Cos                            4
YetAnotherVersionofb = Cos@x                            5
YetEvenAnotherVersionofb =
 Function[z, Cos[z]][x]                                 6
YetStillAnotherVersionofb =
 Function[Cos[#]][x]                                    7
FinallyAnotherVersionofb = (Cos[#] &)[x]                8
ANewVariable[x]                                         9
```

Mathematica Operations on expressions

```
c
AnotherVersionofC = Expand[c]                           10
c
Simplify[AnotherVersionofC]                             11
```

Calculus

```
IntegralofC = Integrate[c, x]                           12
Integrate[c / x, x]                                     13
```

Getting information (part 1)

```
? ExpIntegralEi                                         14
```

1–3: This is a simple example of building up an expression piece-by-piece. For very complicated expressions, this is much easier and less prone to typing errors.

4–8: One of the difficulties of learning MATHEMATICA® is that the syntax can appear to be very complicated and hard to remember. As you begin, just use functions in the form of `Cos`[x]. Here, just as a heads-up, other ways to do the samething are presented. We will use **4** sometimes in this course, because it is convenient. The most useful form is probably **8**, this invokes the concept of a *pure function*.

10–11: One of the powerful aspects of a symbolic algebra program is the manipulation of expressions. It's fast and it doesn't make mistakes as one might using pencil and paper. Here are examples of `Expand` (which expands all products) and `Simplify` (which uses an algorithm to choose among various forms).

12–13: Another powerful aspect is the ability to perform more advanced mathematics. The integral in **12** is one that *perhaps* you might have been able to do after one semester of calculus; **13** is one you would have to manipulate and look up in tables—the answer demonstrates that MATHEMATICA® knows about many many different functions.

14: If you see a symbol that you don't recognize you can either use the help-browser or ask the front-end directly. MATHEMATICA® has a fairly consistent function naming strategy The first letter of a word is always capitalized; compound words are concatenated together while maintaining the first letter capitalization; thus `InverseBetaRegularized`. A function is just another symbol—if a symbol is followed by square brackets `[]` the stuff inside the brackets become the argument(s) for the function.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Lecture 02 MATHEMATICA® Example 3

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Calculus and Plotting

The derivative and integration methods are introduced. Simple plotting methods are demonstrated with an example of annotating a plot.

```
D[ANewVariable[x], x]                                                    1
Integrate[ANewVariable[x], x]                                            2
D[ANewVariable[x], z]                                                    3
tempvar =
 Integrate[ANewVariable[x], {x, 0, y}]                                   4
D[tempvar, x]                                                            5
D[tempvar, y]                                                            6
Factor[IntegralofC]                                                      7
IntegralofC
AnotherVersionofIntegralofC =
 Integrate[AnotherVersionofC, x]                                         8
c
D[IntegralofC, x]                                                        9
Factor[c]
Simplify[D[IntegralofC, x]]                                              10
Plot[IntegralofC, {x, 0, 10}]                                            11
Plot[{IntegralofC, c}, {x, 0, 10}]                                       12
Plot[c, {x, 0, 10}, PlotRange → {0, 0.0001}]                             13
Options[Plot]                                                            14
Plot[{IntegralofC, c}, {x, 0, 10},
 PlotStyle → {{Red, Thickness[0.005]},
   {RGBColor[0.2, 0.56, 1],
    Thickness[0.0075]}}, BaseStyle →
  {FontFamily → "Helvetica", FontSize → 24},
 PlotLabel → " A Function (Pretty Sky
    Blue)\nand Its Integral (Red)\n",
 AxesLabel → {"Value", "Argument"},
 ImageSize → 800]                                                        15
```

1: If MATHEMATICA® can't differentiate or integrate a function, it will be left in a symbolic form. `D` is MATHEMATICA® 's function to take derivatives.

2: If `Integrate` can't integrate a function, it will return the result in symbolic form.

3: MATHEMATICA® is rigorous about applying the rules of calculus. . .

4: Let `tempvar` be the result of integrating some function of `x` from `0` up to some arbitrary value `y`—the result should be a function of `y`.

5–6: MATHEMATICA® knows about the fundamental theorems of calculus. . .

7: The calculus operations will often create long and complicated expressions. That two expressions are equivalent can *sometimes* be shown with built-in functions such as `Simplify`, `FullSimplify`, `Factor`, `Expand`, `Collect`, etc., but sometimes it is an art to turn an expression into an aesthetic form.

8–10: Reassign `IntegralofC` to the *indefinite integral of* c—note, the constant of integration is set to zero. The integration-result is not necessarily left in the most simple form.

11: This is the simplest form of `Plot`. The second argument is a list giving the variable and its bounds. The first argument should have a numerical value at most of the points within the variable's bounds.

12: These two expressions ought to be the same; however, the output-result doesn't make this obvious.

13: Here, operations on the above expressions do show that they are same.

14: This is the simplest version of `Plot`—all it needs is the expression to plot and the range over which to plot a variable—in this case `x` from 0 to 10.

15: If we form a list of two expressions with {–}, then we get a curve for each expression.

16: MATHEMATICA® 's `Plot` has an algorithm to set the values of y-axis if it is not specified. To specify, one sends `Plot` and option in the form of a rule—here the rule is specified for `PlotRange`.

17: To find all the possible options for a function with their default values, the `Options` function provides a way to decipher what aspects of a plot can be changed easily.

18: Here is an example with a plot title, axes labels, different colors and thickness for the curves.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Lists, Lists of Lists, and Operations on Lists

Lists are useful ways to keep related information together, and MATHEMATICA® uses them extensively. Lists could be created in MATHEMATICA® by using the `List` function, but they are usually entered in with curly-brackets {} and each *element* of the list is separated by commas. List elements can be about anything, even lists themselves.

```
AList = {a, b, 2, 7, 9, 1.3, π/2, 0}                                    1
Length[AList]                                                           2
Cos[AList]                                                              3
AList
AList[[2]]                                                              4
AList[[{3, 6}]]                                                         5
AList[[-2]]                                                             6
Sort[AList]                                                             7
Select[AList, NumberQ]                                                  8
Reverse[Sort[Select[AList, NumberQ]]]                                   9
Select[AList, EvenQ]                                                   10
Select[AList, PrimeQ]                                                  11
Perms =
 Permutations[Select[AList, ExactNumberQ]]                             12
Dimensions[Perms]                                                      13
Transpose[Perms] // MatrixForm                                         14
TranPerms = Transpose[Perms];                                          15
TranPerms[[3]]                                                         16
TranPerms[[1, 4]]                                                      17
IntList =
 Table[i, {i, 1, Length[TranPerms[[1]]]}]                              18
TranPerms // MatrixForm
TranPerms[[All, Select[IntList, OddQ]]] //                             19
 MatrixForm
```

1–2: Here is a simple assignment of a list to a variable and the operation of `Length` on the list.

3: Some functions, such as `Cos` here, are *threadable functions*; when called on a list-argument, they will produce a list of that function applied to each list element.

4: A list's parts (its elements) can be picked out in a variety of ways. The `Part` function has a shorthand double-bracket form.

5: If the argument inside the double bracket is a list of integers, the elements corresponding to those integers are returned.

6: Negative integers pick elements from the *end* of the list.

7: `Sort` returns a sorted list; it will also take a second argument to specify alternative sorting rules.

8–11: There are plenty of functions designed to operate on lists; here `Select` returns those elements for which the second argument (`NumberQ`, in **8**) evaluates to True. Functions can be applied sequentially, as in **9**, and the inner-most function is applied first.

12–13: A list's elements can be lists themselves. For example, a matrix is represented by a list of a list. In **12**, `Permutations` creates a list whose elements are all the permutations (which are list themselves) of the list on which `Permutations` was executed. And there even are higher-dimensional structures such as tensors. `Dimensions` is a useful way of learning about such structures.

14: Here, the *post-fix operator* for a function (`MatrixForm`) is used to change the way a matrix is displayed. *Note, the result is* ***not*** *a matrix, but a* `DisplayForm` *of a matrix.*

15: `Transpose` returns the result of taking a matrix and turning the rows into columns and vice-versa.

18: `Table` is a common way to produce a list; here, a list of integers as long as the first row of `TransPerms` is produced.

19: This is a fairly advanced example of extracting the odd-numbered columns of a matrix. The list `IntList` is simply the integers for each column; its odd-numbered members are selected and become the second (i.e., column) argument of the `Part` selection. The first argument is `All`, so the entire row is captured for each selected column.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Rules (→) and Replacement (/.); Getting Help

A rule `leftvar` → `rightvar` is *similar* to assignment in that it associates a new symbol (`leftvar`) with something else, but the value is not assigned—it does not effect future values of the left-hand-side symbol. Rules are often used in conjunction with replacements and to set options in functions. Many of MATHEMATICA® functions, (e.g., `Solve`) return rules as a result.

Rules → and Replacement /.

```
ARule = a → π/3                                   1
a                                                 2
AList
AList /. ARule                                    3
SomeRules = {ARule, b -> π/12}                    4
AList /. SomeRules                                5
a = SomeOtherSymbol;                              6
AList                                             7
StrangeRule = {Rational[x_, y_] :> y / x}         8
(AList /. SomeRules) /. StrangeRule               9
```

Getting Help: *Several methods of getting help are available.*

1. Typing `?ExpIntegralEi` returned information about the symbol `ExpIntegralEi`. Typing `??FunctionName` gives even more information—try `??Plot`. Wildcards can also by used as demonstrated below. You can click on the resulting grid-list to pull up documentation.

```
? *Exp*                                           10
```

Each of the above is linked to Mathematica's Help Browser.

2. Typing Options[Plot] returned a list of options that can be adjusted by the user until the result (in this case the appearance) of the plot is satisfactory.

Mathematica's Help Browser is a very useful tool and will probably become a primary resource for students. It contains good tutorials and demonstrations that can be copied and pasted. It has very good and concise descriptions of mathematics; in fact, exploring the Help Browser is a good way to explore mathematics as well as Mathematica. For instance , the discussion of "Combinatorial Functions" by typing "Combinatorial" at the help browser---you will get a list of results that points to tutorials and overvies.

1: The *rule* `a` → $\pi/3$ is *assigned* to the symbol `ARule`. The rule can be read as, "let `a` become $\pi/3$".

2: Note, the rule does *not* make an assignment to the symbol `a`.

3: A rule can be applied with the function `Replace`, but the syntax (.) is typically used instead; one can read `expression/.rule` as "what would `expression` become, if `rule` was applied to it."

4–5: Rules can be collected into lists, and then applied sequentially to an expression.

6: Assignment of `a` will change the form of `ARule`, because if MATHEMATICA® is asked for a symbol it will make any assignments that have been called—in this case, `ARule` will automatically become `SomeOtherSymbol`→ $\pi/3$.

7: Likewise, `AList` will change because it contained the symbol `a`.

8: This is a somewhat advanced example using patterns and delayed rules, which will be explained later, but the point is this: Rules are necessary for manipulations in MATHEMATICA® , but can be used to generate "mistakes." Think of Rule and Replace acting on an expression as "What would the expression be if a certain rule were applied to it?" If the rule is wrong, the resulting expression will be as well.

9: As an exercise, see if you can figure out why this list turned out like this.

10: Besides the help browser, there are ways to get help directly from the FrontEnd. Here, a list of hyperlinks to documentation for functions containing the string "`Exp`" is obtained.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Getting Help on Mathematica

With the implementation of MATHEMATICA® 6.0, the help-browser changed considerably. Many consider it an improvement, but I am not yet convinced—perhaps I am becoming inflexible and resistant to change.

In the old days, one would memorize large portions of the MATHEMATICA® book—which has grown continuously heavier since its first publication in the early 1990's—and rely on the useful "?" and "??" operators. The use of "?" with the wildcard "" enabled a beginning user to track down almost any MATHEMATICA® function. The Options function is also a very efficient way to discover alternate ways of getting results.

Wolfram decided to stop updating the book as of 6.0, and it is no longer being published. When MATHEMATICA® first came out (I was already familiar with symbolic algebra packages like Macsyma, which predated MATHEMATICA®), I learned it by reading the entire book in one sitting and then I could quickly find and reread parts as I needed it. I found this very effective, and I am curious about what is the most effective way to learn MATHEMATICA® today. I'd be happy to hear suggestions.

In any case, I encourage you to idly explore the MATHEMATICA® Help-Browser. You will not only learn about MATHEMATICA® , but also about mathematics.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

Lecture 3: Introduction to Mathematica II

Sept. 10 2007

3.016

Functions and Rules

Besides MATHEMATICA® 's large set of built-in mathematical and graphics functions, the most powerful aspects of MATHEMATICA® are its ability to recognize and replace patterns and to build functions based on patterns. Learning to *program* in MATHEMATICA® is very useful and to learn to program, the basic programmatic elements must be acquired.

The following are common to almost any programming language:

Variable Storage A mechanism to define variables, and subsequently read and write them from memory.

Loops Program structures that iterate. A well-formulated loop will always be guaranteed to exit[2].

Variable Scope When a variable is defined, what other parts of the program (or other programs) will be able to read its value or change it? The scope of a variable is, roughly speaking, the extent to which it is available.

Switches These are commands with outcomes that depend on a quality of variable, but it is unknown, when the program is written, what the variable's value will be. Common names are If, Which, Switch, IfThenElse and so on.

Functions Reusable sets of commands that are stored away for future use.

All of the above are, of course, available in MATHEMATICA® .

The following are common to Symbolic and Pattern languages, like MATHEMATICA® .

[2]Here is a joke: "Did you hear about the computer scientist who got stuck in the shower?" "Her shampoo bottle's directions said, 'wet hair, apply shampoo, rinse, repeat'."

Patterns This is a way of identifying common structures and make them available for subsequent computation.

Recursion This is a method to define function that obtains its value by calling itself. An example is the Fibonacci number $F_n \equiv F_{n-1} + F_{n-2}$ (The value of F is equal to the sum of the two values that preceded it.) F_n cannot be calculated until earlier values have been calculated. So, a function for Fibonacci must call itself recursively. It stops when it reaches the end condition $F_1 = F_2 = 1$.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Procedural Programming

Simple programs can be developed by sequences of variable assignment.

Evaluating a sequence of instrutions (;;;)

```
a = 1;
a = a + a; a = a^a
a = a + a;  a = a^a
```
1

```
Clear[a]
```
2

Loops

```
? Do
```
3

```
a = 1; Do[a = 2 a; a = a^a, {i, 1, 2}]
```
4

```
a
```
5

```
a = 0.1; Do[a = 2 a; a = a^a;
 Print["iteration is ", i, " and a is ", a],
 {i, 1, 4}]
```
6

```
Clear[a]
```
7

```
? For
```
8

```
For[a = 0.1; i = 1,
 i ≤ 4, i++, a = 2 a; a = a^a;
 Print["iteration is ", i, " and a is ", a]]
```
9

```
? While
```
10

```
? Table
```
11

```
Clear[a]
```
12

```
a = 0.25;
Table[{i, a = 2 a; a = a^a}, {i, 1, 4}]
```
13

```
a = 0.75;
Table[{i, a = 2 a; a = a^a}, {i, 1, 4}]
```
14

```
datatable = Table[
  {dx, For[a = dx; i = 1, i ≤ 4, i++, a = 2 a;
    a = a^a]; Log[a]}, {dx, 0.01, 0.5, 0.01}]
```
15

1: Here is a simple program that is just a sequence of statements that reassigns `a` from an initial value (`a=1`). The program does this: take `a` add it to itself and assign the result back to `a`; raise this new `a` to the power `a` and assign back to itself. Repeat. In MATHEMATICA® , a semicolon— ;—just indicates that output should be suppressed. There are five executions—two of them produce output on the screen.

3: However, it would be cumbersome and unaesthetic if we wanted to generalize the last two lines to many executions of the same type. This is where *program loops* come in. `Do` is a simple way to loop over an expression a fixed number of times. This is equivalent to item **1**, but could be easily generalized to more iterations.

4: The `Do` loop does not produce intermediate output, the current value of `a` can be obtained by asking MATHEMATICA® for the current value.

5: Here an equivalent example, but extra `Print` statements are added so that *intermediate output* can be observed.

8: A `For` loop is another loop structure that enforces good programming style: Its arguments provide: an initialization, an exit condition, an iteration operator, and a function statement, and is equivalent to item **6**, but it includes (a different) initial value for `a` in the `For` statement and iterates 4 times instead of 2.

9: The are many types of loop constructs; `While` is yet another.

10: `Table` is a very useful MATHEMATICA® iterating function. While it iterates, it leaves intermediate results in a List structure. Thus, the built-up list can be analyzed later.

13: Except for the *intial iteration value* of `a`, and the number of loops, this is practically equivalent to items 1, 4, 6, and 9. We can think of this as a little program that takes an initial value of `a` and returns a final value as the last member of the resulting list.

14: We could change the initial value and see how the function varies with its initial value.

15: Or, we can generalize to many initial values, by putting a `Table` and a `For` together. The result is a list of lists (each of length 2): The first entry in each list is the initial value (`dx`) and the second entry is the result of the `For`-loop after four iterations for that `dx`. Because the values tend to get very large, we wrap a `Log` (natural log) around the result of the `For`-loop. A special *increment structure* is utilized—it sets initial and final values as well as the increment size.

3.016

3.016 Home

Full Screen

Close

Quit

Lecture 03 MATHEMATICA® Example 2

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Plotting Lists of Data and Examples of Deeper MATHEMATICA® Functionality

This demonstrates how visualizing data can be combined with other functions to perform analysis. Here, we show that the little iterative program produces a minimum and then we analyze the minimum with two different methods.

```
ListPlot[datatable]
```
1

```
Options[ListPlot]
```
2

*Note that the options are written as **Rules**.*

```
ListPlot[datatable, PlotRange → {250, 500},
 PlotStyle → PointSize[0.025]]
```
3

```
? *Minimum*
```
4

```
FindMinimum[For[a = xvalue;
  i = 1, i ≤ 4, i++, a = 2 a; a = a^a];
 Log[a], {xvalue, 0.15, 0.25}]
```
5

*By going into the Help Browser, you can see that the output of FindMinimum is a list, the first element of which is the functions minimum value, and the second is a **Rule** specifing where the minimum occurs.*

*Lets try and do the above the hard way. I will use **Nest** to recursively apply the function 4 times (I am just using a shorthand here, we can ignore the use of **Nest** for this course...). You can see that it works. Don't worry about it, but if you want to know about it, use the Help Browser to get information about **Nest** and **Pure Functions.***

```
Clear[x]
```
6

```
fx = Nest[(2 #)^(2 #) &, x, 4]
```
7

Take it derivative and set equal to zero...

```
dfx = D[fx, x] // Simplify
```
8

*Finding the zero of this will not be easy.... but **FindRoot** claims it can do it...*

```
FindRoot[dfx, {x, .1, .3}]
```
9

1: The data produced from the last example can be plotted. It is apparent that there is a minimum between initial values of 0.1 and 0.3. But, it will be difficult to see unless the visualization of the plot can be controlled.

3: By specifying the `ListPlot`'s option for the range of the y-like variable, the character of the minimum can be visually assessed.

4: It is likely that MATHEMATICA® has functions to find minima; here we look for likely suspects.

5: `FindMinimum` is a fairly sophisticated function to obtain the minimum of an expression in a specified range, even if the function only returns a numerical result. Here `FindMinimum` is used, to find a very high precision approximation to the minimum observed in item 3. The function is our `For`-loop with a variable `xvalue` as the initial value. We ask `FindMinimum` to hunt for the `xvalue` that minimizes the (`Log` of the) `For`-loop.

7: This is a fairly advanced example—beginning students should not worry about understanding it yet. `Nest` is a sophisticated method of repeated applications of a function (i.e., $f(f(f(x)))$ is nesting the function f three times on an argument x). It is equivalent to the previous methods of producing the iterative structure, but now the result is an expression with a variable `x` that plays the role of the initial value. This concept uses *Pure Functions* which are produced by the ampersand &.

8: The minimum of the function can be analyzed the standard way, here by taking derivatives with `D`. It would not be amusing (that is, for most of us) to find this derivative by hand.

9: `FindRoot` is sophisticated numerical method to obtain the zero of an expression in a specified range.

Very complex expressions and concepts can be built-up by loops, but within MATHEMATICA® the complexity can be buried so that only the interesting parts are apparent and shown to the user.

Sometimes, as complicated expressions are being built up, intermediate variables are used. Consider the value of `i` after running the program:
`FindMinimum[For[a = dx; i = 1, i ≤ 4, i++, a = 2a; a = a^a]; Log[a], {dx, 0.15, 0.25}]`; the value of `i` (in this case 5) has no useful meaning anymore. If you had defined a symbol such as `x = 2i` previously, then `x` would now have the value of 10, which is probably not what was intended. It is much safer to localize variables—in other words, to limit the scope of their visibility to only those parts of the program that need the variable and this is demonstrated in the next example. Sometimes this is called a "Context" for the variable in a programming language; MATHEMATICA® has contexts as well, but should probably be left as an advanced topic.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Making Variables Local and Using Switches to Control Procedures

Describes the use of `Module` to "hide" a variable: consider the variable `a` from the first item in the above example—its intermediate values during iteration are not always important. Suppose you wish to use the symbol `a` later, that it played an intermediate role hence was not used, and may easily be forgotten. It is good practice to make such variables 'local' to their own functions.

An example of a logical switch is demonstrated for `If`.

Local Variables

```
xvalue
a
```
1

```
CurrentValueofA = a;
```
2

```
xvalue = SnickerDoodle; a = HappyGoLucky;
Module[
 {xvalue, a, maxiteration = 4, solution, i},
 solution =
     FindMinimum[For[a = xvalue; i = 1,
    i ≤ maxiteration, i++, a = 2 a; a = a^a];
   Log[a], {xvalue, 0.15, 0.25}];
 Print[xvalue /. solution[[2]]]
]
```
3

```
xvalue
a
solution
```
4

Switches: If, Which

```
a = Prime[23] + Prime[62] + Prime[104]
```
5

```
If[PrimeQ[a],
 Print[a , " is a Prime Number"],
 Print[a,
  " is not Prime, its divisors are ",
  Divisors[a]],
 Print["I have no idea what
    you are asking me to do!"]
]
```
6

The above program is ok, but not very useful because it only works for the current value of **a**. *It would be more useful to have something that worked for any value of* **a** *and could use it over again—that is, turn it into a tool. This involves patterns and function definitions.*

1: The symbols `xvalue` and `a` are left over from the last example, even though they played only an intermediate role for the final result. It is not unusual to run the same MATHEMATICA® for a day or more—it would be easy to forget that values have been assigned to symbols.

2: This could lead us to mistakenly use its value later as though it might be undefined. This is a common error.

3: The production of such errors can be reduced with a programming practice known as *localized variables* (also known as variable-scoping). The idea is to hide the variable within its own structure—the variable is said to have a limited *scope*. `Module` provides a function for doing this. Here symbols `xvalue` and `a` have set values before the call to `Module`, but any value that is changed *inside* of `Module` has no effect on its "global" value in the rest of the MATHEMATICA® session.. *Using* `Module` *is good programming practice for creating your own functions.*

4: Even though `Module` changed the symbols `xvalue` and `a`, and used an internal variable `solution`, there should be no effect outside of `Module`.

6: It is useful to build functions that are "smart" (or appear to be so, by applying rules of logic). Here, a simple example of the use of `If` will be applied to a symbol which is the sum of the 23^{rd}, 62^{nd}, and 104^{th} prime numbers.

This is a simple program. First, it checks if `a` is prime using the query-function `PrimeQ`. If the check is true, then it prints a message saying so, and then returns control to the MATHEMATICA® kernel. If the check is false, then it prints out a message and some more useful information about the fact it isn't prime using `Divisors`. If the statement cannot be determined to be true or false, a message to that effect is printed.

3.016 Home
Full Screen
Close
Quit

Patterns are extremely important in mathematics and in MATHEMATICA® . The goal for beginners should be to master how to create your own functions: *understanding how to use patterns is essential to creating your own functions in* MATHEMATICA® .

In MATHEMATICA® , the use of the underscore, _, means "this is a placeholder for something that will be used later."[3] In other words, you may want to perform a predictable action on an object (e.g., find the value of its cosine, determine if it is prime, plot it), but want to create the *action before the object exists*. We create the action using a pattern (_), the arbitrary object, and create fixed operations on the pattern.

Usually, one needs to name the pattern to make it easier to refer to later. The pattern gets named by adding a head to the underscore, such as `SomeVariableName_`, and then you can refer to whatever pattern matched it with the name `SomeVariableName`.

This is a bit abstract and probably difficult to understand without the aid of a few examples. We start with patterns and replacement in the following example, and then build up to functions in the next example.

[3] It is a bit like teaching a dog to fetch—you cock an arm as if to throw _something_, and then when something gets thrown, your dog runs after the "something." The first _something_ is a place holder for an object, say anything from a stick to a ball to the morning paper. The second something is the actual object that is actually tossed, that finally becomes the "something" your dog uses as the actual object in the performance of her ritual response to the action of throwing.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Operating with Patterns

Patterns are identified by the underscore _, and the matched pattern can be named for later use (e.g., `thematch_`).

```
AList = {first, second,
  third = 2 first, fourth = 2 second}
```
1

```
AList /. {2 a_ → a}
```
2

```
Clear[a]
```
3

```
AList /. {2 a_ → a}
```
4

```
AList /.
 {p_, q_, r_, s_} → {p, pq, pqr, pqrs}
```
5

```
{2, 0.667, a/b, Pi} /. {p_Integer → p One}
```
6

_ all by itself stands for anything. x_ also stands for anything, but gives anything a name for later use.

```
AList /. _ → AppleDumplings
```
7

```
PaulieNoMealX = Sum[b[i] x^i, {i, 2, 6}]
```
8

```
PaulieNoMealX /. x^n_ → n x^(n - 1)
```
9

Make the rule work for any polynomial...

```
DerivRule = q_^n_ → n q^(n - 1);
```
10

```
PaulineOMealY = Sum[c[i] z^i, {i, 2, 6}]
```
11

```
PaulineOMealY /. DerivRule
PaulieNoMealX /. DerivRule
```
12

Another problem is that it will not work for first-order and zeroeth-order terms...

```
PaulENoMiel = Sum[c[i] HoneyBee^i, {i, 0, 6}]
```
13

```
PaulENoMiel /. DerivRule
```
14

This could be fixed, but it would be much easier to do so by defining functions of a pattern.

It is also possible to have a pattern apply conditionally.

```
Cases[{{1, 2}, {2, 1}, {a, b}, {2, 84}, 5},
 {first_, second_} /; first < second]
```
15

1: Construct an example `AList = {first, second, 2first, 2second}` to demonstrate use of pattern matching. We will try to replace members that match `2 something` with `something` There is an instructive error in the first try.

2: The rule is applied to `AList` through the use of the operator `/.` (short-hand for `ReplaceAll`). The pattern here is "two multiplied by something." The symbol `a` should a placeholder for *something*, but `a` was already defined and so the behavior is probably not what was wanted: `2 something` was replaced by the current value of `a`. Another (probably better, but better left until later) usage is the *delayed ruleset* `:->`.

4: After `a` has been cleared, the symbol `a` is free to act as a placeholder. In other words, `a` takes on the temporary value of the last match. The effect of applying the rule is 2×*all somethings* are replaced by the pattern represented by `a` which takes a temporary value of *each something*.

5: Here is an example that uses each member of a four-member list, names the members, and then uses a rule to operate on the entire list. Study this example until you understand it.

6: The types of things that get pattern-matched can be restricted by adding a *pattern qualifier* to the end of the underscore. Here, we restrict the pattern matching to those objects that are `Integer`. The first replacement makes sense; however, the third member of the list is understood by considering that the internal representation of a/b is `a×Power[b,-1]`—the -1 is what was matched.

7: It is not necessary to name a pattern, but it is a good idea if the match is to be used again later. Here, the first thing that gets matched (the list itself) is replaced with the new symbol.

8: For a simple (incomplete and not generally useful) example of the use of patterns, an example producing symbolic derivative of a polynomial will be developed. Here, a polynomial `PaulNoMealX` in `x` is defined using `Sum`.

9–10: A rule is applied, which replaces patterns `x` to a power with a derivative rule. If only the power is used later, so it is given a place-holder name `n`. This technique would only work on polynomials in `x`. To generalize (**10**), we need a place-holder for the arbitrary variable and its powers.

13–14: This will not work for the constant and linear terms in a polynomial. This could be fixed, but the example becomes complicated and still not as good as MATHEMATICA® 's built-in differentiation rules.

15: To place more control on the types of patterns that get matched, patterns can also be used in conjunction with `Condition` operator `/;`. Here is an example of its use in `Cases`. The pattern is any two-member list *subject to the condition* that the first member is less than the second. Cases returns those members of the list where the pattern was successfully matched.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Creating Functions using Patterns and Delayed Assignment

Understanding this example is important for beginners to MATHEMATICA® *!*

The real power of patterns and replacement is obtained when defining functions. Examples of how to define functions are presented.

Defining Functions with Patterns

Defining functions with patterns probably combines the most useful aspects of Mathematica. Define a function that takes patten matching x as its first argument and an argument matching n as its second argument and returns x to the n^{th} power:

```
f[x_ , a_] = x^a;
(*This is not a good way to define
 a function, we will see why later*)
```
1

```
f[2, 3]
f[y, z]
```
2

This works fine, but suppose we had defined **x** *ahead of time*

```
x = 4
```
3

```
f[x_ , a_] = x^a;
(*This is not a good way to define
 a function, we will see why later*)
```
4

```
f[2, 3] (*will now be 4^3,
which is probably not what
 the programmer had in mind*)
```
5

```
f[y, z]
```
6

Better Functions with Delayed Assignment (:=)

```
x = 4
a = ScoobyDoo
```
7

```
f[x_ , a_] := x^a
```
8

```
f[2, 5]
```
9

```
f[y, z]
```
10

```
f[x, a]
```
11

```
f[a, x]
```
12

```
Clear[f]
```
13

1: Here is an example of a pattern: a symbol `f` is defined such that if it is called as a function with a pattern of two named arguments `x_` and `a_`, then the result is what ever x^a evaluated to be when the function was defined. **Don't emulate this example—it is not usually the best way to define a function.** In words you are telling MATHEMATICA® , "any time you see f[thing,doodad] replace it with the current value of thing^doodad."

2: Our example appears to work, but only because our pattern variables, `x` and `a` had no previous assignment.

3–6 This shows why this can be a bad idea. `f` with two pattern-arguments, is assigned *when it is defined*, and therefore if either `x` or `a` was previously defined, then the definition will permanently reflect that definition. The `=`-assignment is performed immediately and anything on the right-hand-side will be evaluated with their immediate values.

7–12: What we really want to tell MATHEMATICA® in words is, "I am going to call this function in the future. I want to define the function now, but I don't want MATHEMATICA® to evaluate it until it is called; use the pattern-matching variables when you evaluate it later." This involves use *delayed assignment* which appears as `:=`.

For beginning users to MATHEMATICA® , this is the best way to define functions.

In a delayed assignment, the right-hand-side is *not evaluated until the function is called* and then the patterns become *transitory until the function returns its result.* This is usually what we mean when we write $y(x) = ax^2$ mathematically—if y is given a value x, then it operates and returns a value related to that x *and not any other* x *that might have been used earlier.*

This is the prototype for function definitions.

3.016 Home
Full Screen
Close
Quit

Until you become more familiar with MATHEMATICA® , it is probably a good idea to get in the habit of defining all function with delayed assignment (:=) instead of immediate assignment (=). With delayed assignment, MATHEMATICA® does not evaluate the right-hand-side *until* you ask it to perform the function. With immediate assignment, the right-hand-side is evaluated when the function is defined making it much less flexible because your name for the pattern may get "evaluated away."

Defining functions are essentially a way to eliminate repetitive typing and to "compactify" a concept. This "compactification" is essentially what we do when we define some function or operation (e.g., $\cos(\theta)$ or $\int f(x)dx$) in mathematics—the function or operation is a placeholder for something perhaps too complicated to describe completely, but sufficiently understood that we can use a little picture to identify it.

Of course, it is desirable for the function to do the something reasonable even if asked to do something that might be unreasonable. No one would buy a calculator that would try to return a very big number when division by zero occurs—or would give a real result when the arc-cosine of 1.1 is demanded. Thus, a bit of care is advisable when defining functions: you want them to behave reliably in the future when you have forgotten what you have done. Functions should probably be defined so that they can be reused, either by you or someone else. The conditions for which the function can work should probably be encoded into the function. In MATHEMATICA® this can be done with restricted patterns.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Functional Programming with Recursion: Functions that are Defined by Calling Themselves

This is an example of how one might go about defining a function to return the factorial of a number. Instructive mistakes are introduced and, in the following example, we will make the function behave better with incremental improvements.

We will also show how to speed up programs by trading memory for speed.

The canonical programming example is the factorial function n! = (n)x(n-1) x(n-2)x...x(1) where 0! = 1; here is a reasonably clever way to use the fact that (n+1)! = (n+1)xn!

```
factorial[n_] := n factorial[n - 1]
```
1

```
factorial[8]
```
2

Ooops, This isn't what was expected, but upon reflection it is correct--we forgot to define a part of the rule. (Note also that the message window produced an error about recursion limits) Add the second part of the definition. Here, we don't use delayed evaluation (:=) because we want to assign a value immediately.

```
factorial[0] = 1;
```
3

```
factorial[120]
```
4

```
factorial[257]
```
5

Here is where the recursion limit comes in : our function keeps on calling itself (i.e., recursively). Unless a limit is set the program would keep running forever. Mathematica builds in a limit to how many times a function will call itself:

```
$RecursionLimit
```
6

```
$RecursionLimit = 2^11
```
7

Speed versus Memory in Functions

```
Timing[factorial[2000]][[1]]
```
8

Using immediate assignment in a function: spending memory to buy time: Each time the function is called, it makes an extra assignment so that previous values can be recalled if needed.

```
factorial[n_] :=
 factorial[n] = n * factorial[n - 1]
```
9

This version takes a bit longer the first time, because we are storing data in memory ...

```
Timing[factorial[2000]][[1]]
```
10

But, the next time it is called, the result is much faster.

```
Timing[factorial[2001]][[1]]
```
11

```
Clear[factorial]
```
12

1: This is a functional definition that will produce the factorial function by recursion because $(n+1)! = (n+1)n!$—the result for $n+1$ is obtained by using the previous result for n.

2: However, trying this function now will produce an advisory in the MATHEMATICA® 's *Message Window*, and will not give a satisfactory result because...

3: It is necessary to define a place for the recursion to stop. This is done by *assigning* the factorial of zero to be unity.

5: So that recursive functions don't run for ever, leaving no way, a sensible limit is placed on the number of times a function can call itself. MATHEMATICA® sets a number of variables such as `$RecursionLimit`, that control global behavior.

7: However, the user is free to subvert the defaults.

8: We will now examine the role of memory and speed, to do this we will need the time it takes MATHEMATICA® to do a computation; this can be obtained with `Timing`. `Timing` returns a list of two elements: the first is the time for the computation; the second is the result of the computation. We will only be interested in the first element.

9: Consider using the function to find the factorial of 2000, the currently-defined function must call itself about 2000 times to return a value. Suppose a short time later, the value of 2001! is requested. The function must again call itself about 2000 times, even though all the factorials less than 2001's were calculated previously. If *you* were the CPU, you might say "why are you asking me to do this all again? Can't you remember anything?" Unless computer memory is abundant, it seems like a waste of effort to repeat the same calculations over and over.

Here is an example where computation speed is purchased at the cost of memory. The definition of the function uses a delayed assignment (`:=`) as well as an immediate assignment (`=`). The delayed assignment defines the function with a pattern—the immediate assignment assigns and stores the value of a symbol. Thus, when the function is called, it makes an assignment as well as the computation.

10–11: Here, we see that it takes a little longer to calculate 2000! (because the CPU is doing memory storage operations), but it takes significantly less time to calculate 2001!.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Restricted and Conditional Pattern Matching

Here are demonstrations of how to restrict whether a pattern gets matched by the *type of the argument* and how to place further restrictions on pattern matching.

Restrictions on Patterns

The factorial function is pretty good, but not foolproof as the next few lines will show.

```
Clear[factorial]
```
1

```
factorial[0] = 1;
factorial[n_] := n * factorial[n - 1]
```
2

The next line will cause an error to appear on the message screen.

```
factorial[Pi]
```
3

The remedy is to restrict the pattern:

```
Clear[factorial]
```
4

```
factorial[0] = 1;
factorial[n_Integer] := n * factorial[n - 1]
```
5

This time it doesn't produce an error, and returns a value indicating that it is leaving the function in symbolic form for values it doesn't know about.

```
factorial[Pi]
```
6

Functions and Patterns with Tests

However, the definition of factorial still needs some improvement--the next line will cause an error.

```
factorial[-5]
```
7

```
Clear[factorial]
```
8

```
factorial[0] = 1;
factorial[n_Integer?Positive] :=
 n * factorial[n - 1]
```
9

```
factorial[12]
```
10

```
factorial[Pi]
```
11

3: However, what if the previously-defined factorial function were called on a value such as π? It would recursively call $(\pi - 1)!$ which would call $(\pi - 2)!$ and so on. Thus, this execution would be limited by the current value of `$RecursionLimit`.
This potential misuse can be eliminated by placing a pattern restriction on the argument of *factorial* so that it is only defined for integer arguments.

5: Here is an improved definition for the *factorial* function using a pattern type: `_Integer`. The type-qualifier at the end of the "_" is the internal representation of whatever the argument was (e.g., `Integer`, `Real`, `Complex`, `List`, `Symbol`, `Rational`, etc.). In this case, the factorial function is *only* defined for integer arguments.

6: Now the function should indicate that it doesn't have anything further to do with a non-integer argument.

7: However, the definition is still not fool-proof because negative integers will not terminate the recursion properly.

9: A pattern can have conditional matching indicated by the `_?Query` where `Query` returns true for the conditions that the pattern can be matched (e.g., `Positive`[2], `NonNegative`[0], `NumberQ`[1.2], `StringQ`["harpo"] all return `True`.) In this example, the function's pattern—`n_Integer?Positive`—might be understood in words as "Match any integer and then test and see if that integer is positive; if so use `n` as a temporary placeholder for that positive integer."

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Further Examples of Conditional Pattern Matching; Conditional Function Definitions

A simple example of patterns is demonstrated with graphics. Another method of using conditions is demonstrated.

As a another example, let's define the Sign function. It should be -1 when its argument is negative, 0 when its argument is zero, and +1 when its argument is positive. There are lots of ways to write this function, there is no best way. Whatever works is good.

```
? Sign
```
1

Here we write our own version, we don' t "name" the pattern because it is not needed in the function definition. It is a bit harder to read this way, but I use it here to be instructive.

```
HeyWhatsYourSign[0] = 0;
HeyWhatsYourSign[0.0] = 0;
HeyWhatsYourSign[_?Positive] := 1;
HeyWhatsYourSign[_?Negative] := -1;
```
2

```
Plot[HeyWhatsYourSign[argument],
 {argument, -π, e}, PlotStyle → Thick]
```
3

```
Plot[{1 / x, HeyWhatsYourSign[x] / x},
 {x, -1, 1},
 BaseStyle →
  {FontSize → 18, FontFamily → "Helvetica"},
 PlotStyle → {{Hue[1], Thickness[0.02]},
   {Hue[0.66], Thickness[0.01]}}]
```
4

Functions with Conditional Definitions

In thermodynamics, x ln(x) appears frequently in expressions that involve entropy. The variable x is restricted to 0 ≤x≤1.

```
XLogX[x_] := x Log[x] /; (x > 0 && x ≤ 1)
XLogX[0] = XLogX[0.0] =
  Limit[xsmall Log[xsmall], xsmall → 0]
```
5

```
XLogX[1.2]
```
6

```
Plot[XLogX[x] + XLogX[1 - x],
 {x, -1, 2}, PlotStyle → Thick]
```
7

1: As an example, we will try to duplicate MATHEMATICA® 's definition of `Sign`.

2: Because we want our function to return zero when it gets called with an argument of zero—exact or numerical, immediate assignment is used in the first two lines. (It would probably better to use the `_? PossibleZeroQ` pattern match here, but slower.)

Because, we don't need to use the value of the matched pattern we can get by without naming it (i.e., `_?Positive`). I include this for instruction purposes—if I were writing this function for later use, I'd probably go ahead and name the pattern for readability.

3: We `Plot` our function to see if it behaves properly. We use `Plot's` option `PlotStyle->Thick` to make the curve easier to see.

4: Here is an example using our function and plotting two curves with more plotting options.

5: The ideal molar entropy of mixing is the sum of $X_i \ln X_i$ for each component i with composition X_i. Because the composition variables are limited to $0 \leq X_i \leq 1$, our example ideal molar entropy function should reflect this constraint.

Here we use a *conditional definition (/;)*, to ensure that our $X \ln X$ function is never called for any X that are out-of-bounds. The delayed assignment statement `LHS := RHS/;test` might be read as, "If the symbol `LHS` is called, then evaluate (using whatever patterns might appear in `LHS`) whether `test` is true; if true, then evaluate RHS with the appropriate pattern replacements." Note that here, we make $X = 0$ a special case and not included in our delayed assignment of the function.

Because $\ln x \to -\infty$ as $x \to 0$, it may not be obvious that $x \ln x \to 0$ as $x \to 0$. We use `Limit` to determine this behavior and use *immediate assignment* in our function definition. (This is a case where immediate assignment makes sense; with delayed assignment the `Limit` function would be called each time that *XLogX* is called on a zero-argument.

7: This is a plot of the ideal molar entropy of mixing for a binary alloy.

3.016 Home
Full Screen
Close
Quit

Lecture 4: Introduction to Mathematica III

Simplifying and Picking Apart Expressions, Calculus, Numerical Evaluation

A great advantage of using a symbolic algebra software package like MATHEMATICA® is that it reduces or even eliminates errors that inevitably creep into pencil and paper calculations. However, this advantage does come with a price: what was once a simple task of arranging an expression into a convenient form is something that has to be negotiated with MATHEMATICA® . In fact, there are cases where you cannot even coerce MATHEMATICA® into representing an expression the way that *you* want it.

A MATHEMATICA® session often results in very cumbersome expressions. You can decide to live with them, or use one of MATHEMATICA® 's many simplification algorithms. The "Algebraic Calculations" topics in the Tutorial Overviews section of the Helper Palette provides a nice summary of frequently used simplification algorithms. Another method is to identify patterns and replace them with your own definitions.

MATHEMATICA® has its own internal representation for rational functions (i.e., $\frac{\text{numerator expression}}{\text{denominator expression}}$) and has special operations for dealing with these. Generally, advanced simplification methods usually require a working knowledge of of MATHEMATICA® 's internal representations.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Operations on Polynomials

There are built-in simplification operations, such as `Simplify`, but they will not always result in a form that is most useful to the user. Crafting an expression into a pleasing form is an art.

```
PaulENomeal = (1 + 2 a + 3 x + 4 z)^4                                    1
FatPEN = Expand[PaulENomeal]                                             2
Factor[FatPEN]                                                           3
PaulinX = Collect[FatPEN, x]                                             4
Coefficient[PaulinX, x, 2]                                               5
PaulSpiffedUp = Sum[
  Simplify[Coefficient[PaulinX, x, i]] x^i,                              6
  {i, 0, 20}]
Simplify[PaulSpiffedUp]                                                  7
RashENell = (x + y)/(x - y) + (x - y)/(y + x)                            8
Apart[RashENell]                                                         9
Together[RashENell]                                                      10
Numerator[Together[RashENell]]                                           11
Simplify[RashENell]                                                      12
Factor[RashENell]                                                        13
```

Simplfiying Expressions with Square Roots

```
RootBoy = Sqrt[(x + y)^2]                                                14
Simplify[RootBoy]                                                        15
Simplify[RootBoy, x ∈ Reals && y ∈ Reals]                                16
Simplify[RootBoy, x ≥ 0 && y ≥ 0]                                        17
Simplify[RootBoy, x < 0 && y < 0]                                        18
RootBoy /. Sqrt[(expr_)^2] → expr                                        19
```

1: We will use this simple expression to demonstrate some of MATHEMATICA® 's algebraic manipulations.

2: `Expand` performs all multiplication and leaves the result as a sum.

3: `Factor` has an algorithm to find common terms in a sum and write the result as a factor and a cofactor—but in this case, it will return the original form.

4: `Collect` will turn in an expression into a polynomial of a user-selected variable.

5: `Coefficient` picks out coefficients of user-specified powers of a variable—this will return the coefficient of x^2 in the polynomial.

6: This is an example of using `Simplify` together with `Coefficient` to simplify only the coefficients of each power of x, and then return the original result by multiplying by the appropriate power and summing.

7: `Simplify` tries to produce a simple result (based on an internal measure of simplicity). Here it returns the same result as `Factor`, but this will not always be the case.

8: Besides polynomials, other frequently encountered forms are *rational forms*—we will use this sum of rationals as an example.

9: `Apart` will re-express a rational form as a sum with simple denominators.

10: `Together` will collect all terms in a sum into a single rational form.

11: `Numerator` returns the numerator of a single rational form.

12–13: In this case, `Simplify` and `Factor` do not produce the same form.

14: MATHEMATICA® is fastidious about simplifying roots and makes no assumptions—unless they are specified— about whether a variable is real, complex, positive, or negative.

15: Many users become frustrated that `Simplify` doesn't do what the user thinks must be correct... *If you think it is obvious that $\sqrt{x^2}$ should always simplify to x, then consider that both $x = \pm 1$ satisfy $\sqrt{x^2} = 1$—picking only $x = 1$ will miss the minus-solution. Or, consider that $\sqrt{x^2} \neq x$ for $x < 0$*

16: `Simplify` will accept `Assumptions` as a second argument, or as an option.

17–18: This demonstrates why it is not a good idea to automatically simplify $\sqrt{x^2}$.

19: This is brute force—and not really a good idea.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

A Second Look at Calculus: Limits, Derivatives, Integrals

Examples of `Limit` and calculus with built-in assumptions

```
AMessyExpression = Log[x Sin[x]] / (1/x)                                   1
Limit[AMessyExpression, x → 0]                                             2
DMess = D[AMessyExpression, x]                                             3
Integrate[DMess, x]                                                        4
DefInt1 = Integrate[DMess, {x, 0, e}]                                      5
(AMessyExpression /. x → e) -
 (AMessyExpression /. x → 0)                                               6
DefInt2 = (AMessyExpression /. x → e) -
  Limit[AMessyExpression, x → 0]                                           7
DefInt1
DefInt2
DefInt1 == DefInt2                                                         8
Integrate[Sin[x] / Sqrt[(x^2 + a^2)], x]                                   9
Integrate[Sin[x] / Sqrt[(x^2 + a^2)],
 x, Assumptions → Re[a^2] > 0]                                             10
UglyInfiniteIntegral =
 Integrate[Sin[x] / Sqrt[(x^2 + a^2)],
  {x, 0, ∞}, Assumptions → Re[a^2] > 0]                                    11
N[UglyInfiniteIntegral /. a → 1]                                           12
Series[AMessyExpression, {x, 0, 4}]                                        13
FitAtZero =
 Series[AMessyExpression, {x, 0, 4}] // Normal                             14
Plot[
 {AMessyExpression, FitAtZero}, {x, 0, 3},
 PlotStyle → {{Thickness[0.02], Hue[1]},
   {Thickness[0.01], Hue[0.5]}}]                                           15
```

1–2: This would be a challenging limit to find for many first-year calculus students (*try it!*).

3–4: Here, do a quick verification using differentiation and integration to check if MATHEMATICA® agrees with the fundamental theorem of calculus (`Integrate[ D[expr,x],x]==x`). Note, MATHEMATICA® does not add the arbitrary constant to the indefinite integral.

5: This definite integral *should* the value of `AMessyExpression` at $x = e$, but is not obvious by inspection.

6: Simply evaluating (via application of rules) the integral at the ends of the integration domain does not produce the correct result because of a possible division by zero.

7: Using `Limit` instead of direct evaluation produces the expected result.

8: Although they have different forms (and one can probably see that they are the same expression), testing equality shows that the two different forms of the definite integral are the same.

9-10: Some indefinite integrals do not have closed-form solutions as in **9**, even with extra assumptions as attempted in **10**.

12: But, in some cases even if the indefinite integral does not have a closed-form solution, the definite integral will have one.

13: `Series` is one of the most useful and powerful MATHEMATICA® functions; especially to replace a complicated function with a simpler approximation in the neighborhood of a point.

`Series` returns a `SeriesData`-form which is indicated by the *trailing order* function `O`. Subsequent operations, such as `Simplify`, won't work on a `SeriesData`-form, but `Normal` converts a `SeriesData` to a normal expression by chopping off the `O`.

14–15: In this example, `FitAtZero` is a fourth-order approximation to `AMessyExpresssion` at $x = 0$ and has been converted with `Normal` so it can be plotted in **15** alongside the exact expression.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Solving Equations

`Solve`, its resulting rules, and how to extract solutions from the rules.

Solving Equations

```
TheEquation = a x^2 + b x + c                                   1
```

*Note the use of **Equal** (==) rather than **Set** (=) in the following; using "=" will produce an error message.*

```
TheZeroes = Solve[TheEquation == 0, x]                          2
```

*Note that the roots are given as **Rules**. Now we ask Mathematica to verify that the solutions it found are indeed roots to the specified equation. Here is a prototypical example of using **Replace** (/.) to accomplish this:*

```
TheEquation /. TheZeroes                                        3
```

```
Simplify[TheEquation /. TheZeroes]                              4
```

*More examples of using **Solve**:*

```
a[i_] := i + 1                                                  5
```

```
TheQuinticEquation = Sum[a[i] x^i, {i, 0, 5}]                   6
```

```
TheFiveSols = Solve[TheQuinticEquation == 0, x]                 7
```

```
N[TheFiveSols]
x /. N[TheFiveSols]                                             8
```

```
Quad1 = a x^2 + y + 3
Quad2 = a y^2 + x + 1                                           9
```

```
Solve[{Quad1 == 0, Quad2 == 0}, {x, y}]                         10
```

1: We assign the familiar quadratic equation to `TheEquation` as a demonstration of how to solve equations and extract solutions.

2: `Solve` takes a *logical equality* (or a list of logical equalities for simultaneous equations) as a first argument. It returns a list of solutions in the form of rules. Here, the list of rules is assigned to `TheZeroes`. There will be one rule for every solution found—if no solutions are found then `Solution` will either return an empty list, or a symbolic list of pure functions that the solutions must satisfy for subsequent use in numerical functions (this case qualifies as an advanced topic).
For the general quadratic case, `Solve` returns a list with two rules of the form `{{x->solution1},{x->solution2}}`—*it is a list of lists.*
We will see why it is a list of lists when we examine the solution to simultaneous equations in two variables in **9**.

3: To evaluate the original equation at the values of x that solve it, one uses the rules (`TheZeros`) as a list of replacements: `TheEquation/.TheZeros` returns a list of the two values with x replaced by the solutions.

4: Using `Simplify` on this result produces the expected zeroes.

5–6: To see what `Solve` might do with higher-order polynomials, we set up a simple function for the coefficients of a particular quintic equation and create it using `Sum`.

7: The zeroes of a quintic polynomial do not have general closed forms. Here MATHEMATICA® will return a symbolic representation of the solution rules—which we assign to `TheFiveSols`. This representation indicates that the solution doesn't have a closed form, but the form is suitable for subsequent numerical analysis.

8: To extract the numerical solution to `TheQuinticEquation==0`, the first line will return a list of rules for `x`; the second line returns a list of `x` with those rules used as a replacement.

10: This is an example of a solution to coupled quadratic equations. There are four solutions with the form: `{{x->xsol1,y->ysol1},...,{x->xsol4,y->ysol4}}`. Each member of the list must contain a rule for each variable; that is why the solution has the form of a list of a list.

3.016

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

Sometimes, no closed-form solution is possible. MATHEMATICA® will try to give you rules (in perhaps a seemingly strange form) but it really means that you don't have a solution to work with. One usually resorts to a numerical technique when no closed-form solution is possible— MATHEMATICA® has a large number of built-in numerical techniques to help out. A numerical solution is an approximation to the actual answer. Good numerical algorithms can anticipate where numerical errors creep in and accounts for them, but it is always a good idea to check a numerical solution to make sure it approximates the solution to the original equation.

Of course, to get a numerical solution, the equation in question must evaluate to a number. This means if you want to know the numerical approximate solutions $x(b)$ that satisfy $x^6 + 3x^2 + bx = 0$, you have to iterate over values of b and "build up" your function $x(b)$ one b at a time.

The "Numerical Equation Solving" topic in the "Numerical Mathematics" within "Tutorial Overviews" section of the Helper Palette provides a nice summary.

Examples of numerical algorithms `NIntegrate` `FindRoot`

Numerical Solutions

```
Integrate[Sin[x] / Sqrt[(x^2 + a^2)], x]                                    1

Integrate[
 Sin[x] / Sqrt[(x^2 + a^2)], {x, 0, 1}]                                     2

NIntegrate[
 (Sin[x] / Sqrt[(x^2 + a^2)]) /. a → 1,
 {x, 0, 2 Pi}]                                                              3

Plot[
 NIntegrate[Sin[x] / Sqrt[(x^2 + a^2)],
  {x, 0, 2 Pi}], {a, 0, 10}, PlotStyle → Thick,
 BaseStyle → {Large, FontFamily → "Helvetica"}]                             4

Plot[{AMessyExpression, FitAtZero}, {x, 0, 3},
 PlotStyle → {{Thickness[0.02], Hue[1]},
   {Thickness[0.01], Hue[0.5]}}]                                            5

NSolve[AMessyExpression == 0, x]                                            6

FindRoot[AMessyExpression == 0, {x, .5, 1.5}]                               7

FindRoot[FitAtZero == 0, {x, .5, 1.5}]                                      8

FindRoot[AMessyExpression == 0, {x, 2.5, 3}]                                9
```

3: `NIntegrate` can find solutions in cases where `Integrate` cannot find a closed-form solution. It is necessary that the integrand should evaluate to a number at all points in the domain of integration (it is possible that the integrand could have singularities at a limited set of isolated points). Thus, a rule and replacement for `a` has to be used for the integrand that appears in **2**. Along with the numerical integrand, the bounds of the definite integral must also be specified.
Like most numerical algorithms, `NIntegrate` can return wrong results (viz `NIntegrate[1/x,{x,1,∞}]`). However, in practice these cases are rare; but, be wary.

4: `NIntegrate` is sufficiently fast that we can treat the integrand in **2** as a function of `a`. Here, we let plot vary `a` like the x-axis and plot the results of the numerical integrand from 0 to 2π as a function of `a`.

5: Here we use `Plot` to compare our previous fourth-order polynomial approximation (`FitAtZero`) to the exact result (`AMessyExpression`).

6: `NSolve` will find roots to *polynomial forms*, but not for more general expressions.

7: `FindRoot` will operate on general expressions and find solutions, but additional information is required to inform where to search.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

1. You will want to save your work.
2. You will want to modify your old saved work
3. You will want to use your output as input to another program
4. You will want to use the output of another program as input to MATHEMATICA® .

You have probably learned that you can save your MATHEMATICA® notebook with a menu. This is one way to take care of the first two items above. There are more ways to do this and if you want to do something specialized like the last two items, then you will have to make MATHEMATICA® interact with files. Because an operating system has to allow many different kinds of programs to interact with its files, the internal operations to do input/output (I/O) seem somewhat more complicated than they should be. MATHEMATICA® has a few simple ways to do I/O—and it has some more complex ways to do it as well.

It is useful to have a few working examples that you can modify for your purposes. The examples will serve you well about 90% of the time. For the other 10%, one has to take up the task of learning the guts of I/O—hopefully, beginners can ignore the gory bits.

The "Files and Streams" overview within the "Tutorial Overviews" section of the Helper Palette is useful. Data reading is also integrated into MATHEMATICA® —see the "Data Handling & Data Sources" section at the top level of the help browser.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Interacting with the Filesystem

Reading and writing data directly and through the use of a *filestream*. A user should check and (sometimes) change the *working directory* to interact with files using `Directory` or `SetDirectory`. Otherwise, the full path to a file must be given.

File Input and Output

```
Directory[]                                                    1
AMessyExpression >> AFile.m                                    2
Clear[AMessyExpression]                                        3
<< Afile.m                                                     4
```

The previous statement reads in the expression, but it is not assigned to its previous symbol

```
AMessyExpression                                               5
AMessyExpression = << AFile.m                                  6
AMessyExpression                                               7
FilePrint["Afile.m"]                                           8
Close["ANewFileName"]                                          9
AFileHandle = OpenWrite[
  "ANewFileName", FormatType → OutputForm]                     10
RandomPairs =
 Table[RandomReal[{0, 1}, 2], {i, 10}]                         11
Write[AFileHandle, RandomPairs]                                12
FilePrint["ANewFileName"]                                      13
Write[AFileHandle, MatrixForm[RandomPairs]]                    14
FilePrint["ANewFileName"]                                      15
Close[AFileHandle]                                             16
```

1: `Directory` will print the *current directory* into which, and from which, files will be read (if that directory is writable and readable). To change MATHEMATICA® 's current directory, use `SetDirectory`.

2: Simple redirection of an expression into a file is achieved with `>>` The working directory must be *writable*. Selected symbols can be saved in files all at once using `Save`.

4: A file containing a MATHEMATICA® expression can be read in with `<<` The file must be *readable*.

5: Only the expression was saved using `>>`, not the symbol it was assigned to.

8: The contents of a file can be displayed using `FilePrint`.

10: This opens a filestream for subsequent use. Note that the filestream (here called `AFileHandle`) is associated with a filename (here `ANewFileName`). Filestreams give the user much more control over the way the file is written. The use of filestreams is useful for cases where data is written incrementally during a calculation and this method can be generalized to different kinds of devices. Another use of file streams is when the user wants to have the program compute the file name as a string value, and then use the filestream to write to a file with a meaningful string (e.g., name the file from a computed string "`x=3_y=2.dat`")

11: We use `RandomReal` to create some example data (a list of ten pairs of random numbers) to write to the filestream.

12: An example of writing data directly with a filestream.

13: Subsequent writes to the filestream get appended to the end of the file. Here we write the `MatrixForm` of the data.

16: It is good practice to close open file streams when writing is finished.

3.016

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Using Packages

There are a number of packages that come with MATHEMATICA® (and more that can be bought for special purposes). The packages contain functions and data that can be added to a MATHEMATICA® session as desired, and not loaded beforehand. This helps regulate the amount of memory required to run MATHEMATICA® . You should look through the various packages in the help browser to get an idea of what is there—it is also a good idea to take a look at the inside of a package by editing a package file with an editor. By doing this, you will see some of internal structure of MATHEMATICA® and good examples of professional programming.

Using Packages

Fortunately, others have gone to the trouble of writing files full of useful stuff--and you can load this stuff into Mathematica for your very own use. Some people produce useful stuff and you can buy it, which is nice if you find it valuable--and you can write stuff and gain value by selling it, which might be even more nice.
Mathematica comes with a group of Standard Packages, that you can load in to do special tasks. The Package documentation can be found with the Helper Palette, available at http://puffle.mit.edu/3.016/Help-Pallette-Builder.nb. For example, take a look at the specialized package Calendar:

```
<< Calendar`
```
1

```
DayOfWeek[{1929, 9, 30}]
```
2

```
DateList[]
```
3

```
CalendarChange[DateList[], Gregorian, Islamic]
```
4

```
DateString[CalendarChange[
  DateList[], Gregorian, Islamic]]
```
5

1: A package is read in using the input operator `<<` or with `Needs`. Here is an example of how `Calendar` is read.

2: `DayOfWeek` is one of the functions available in `Calendar`.

3: `DateList` is part of the standard MATHEMATICA® kernel, without arguments it returns the current date and time.

4: We use the Gregorian calendar–here is the current date in the Islamic calendar.

5: Here, we print the Islamic date in a more readable form. It would be nice to have a little function to translate the day and the month into Arabic...

3.016 Home
Full Screen
Close
Quit

Lecture 5: Introduction to Mathematica IV

Graphics

Graphics are an important part of exploring mathematics and conveying its results. An informative plot or graphic that conveys a complex idea succinctly and naturally to an educated observer is a work of creative art. Indeed, art is sometimes defined as "an elevated means of communication," or "the means to inspire an observation, heretofore unnoticed, in another." Graphics are art; they are necessary. And, I think they are fun.

For graphics, we are limited to two and three dimensions, but, with the added possibility of animation, sound, and perhaps other sensory input in advanced environments, it is possible to usefully visualize more than three dimensions. Mathematics is not limited to a small number of dimensions; so, a challenge —or perhaps an opportunity—exists to use artfulness to convey higher dimensional ideas graphically.

The introduction to basic graphics starts with two-dimensional plots.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Simple Plots

Here are some examples of simple x-y plots and how to decorate them. We start with very simple examples and add a little more at each step to show how a plot can be developed incrementally. We leave all the steps in as cut-and-paste examples.

```
Plot[Sin[x] / x, {x, -5 Pi, 5 Pi}]
```
1

```
Options[Plot]
```
2

```
Plot[Sin[x] / x, {x, -5 Pi, 5 Pi}, PlotRange →
  {-0.25,
   1.25}, PlotStyle → {Red, Thick}]
```
3

```
Plot[Sin[x] / x, {x, -5 Pi, 5 Pi}, PlotRange →
  {-0.25,
   1.25}, PlotStyle → {Red, Thick}, AxesLabel →
  {"x",
   "Sin (x) / x"}]
```
4

```
Plot[Sin[x] / x, {x, -5 Pi, 5 Pi}, PlotRange →
  {-0.25,
   1.25}, PlotStyle → {Red, Thick}, AxesLabel →
  {"x",
   "Sin (x) / x"}, BaseStyle → {Large, FontFamily →
    "Helvetica",
   Italic}]
```
5

```
Plot[Sin[x] / x, {x, -5 Pi, 5 Pi}, PlotRange →
  {-0.25,
   1.25}, PlotStyle → {Red, Thick}, AxesLabel →
  {"x",
   "Sin (x) / x"}, BaseStyle → {Large, FontFamily →
    "Helvetica",
   Italic}, TicksStyle → {{Medium, Blue},
   {Medium,
    RGBColor[0.5, 0.2, 0]}}]
```
6

1: This is the simplest version of `Plot`: all it requires is an expression depending on a variable and a range over which to plot that variable. MATHEMATICA® has algorithms to select the region which is most likely to be of interest.

2: Tweaking the appearance of a plot will usually involve changing one of `Plot`'s options.

3: Here we change `PlotRange` and `PlotStyle` explicitly. `PlotStyle` takes a list of graphics directives, and the type of `PlotStyle` directives will generally depend on what is being plotted (i.e., lines, points, surfaces).

4: The `AxesLabel` option is used here. The `BasicMathInput`-palette is useful to typesetting mathematical expressions.

5: The option `BaseStyle` can be used to specify the basic size, font, font-shape, etc for the entire plot.

6: As a last example, we use a list of two styles for `TickStyle` to specify both x- and y-axis ticking characteristics.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Plotting Precision and an Example of Interaction

Even for continuous functions, a graphical representation is a discrete object. The level of precision is associated with the *mesh*—which is the set where numerical evaluations are performed. More mesh points generally results in a smoother representation, but at the cost of longer computation and memory.

Mesh and MeshStyle

```
Plot[Sin[x] / x, {x, -5 Pi, 5 Pi}, PlotRange →
  All,
 PlotStyle → {Red, Thick}, Mesh → All,
 MeshStyle →
  {Black, PointSize[0.015]}]
```

1

MaxRecursion and PlotPoints

```
Plot[Sin[x] / x, {x, -5 Pi, 5 Pi}, PlotRange →
  All,
 PlotStyle → {Red, Thick}, , Mesh → All,
 MeshStyle →
  {Black, PointSize[0.015]}, MaxRecursion → 2,
 PlotPoints →
  8]
```

2

Interactive Graphics: An Example of Manipulate

```
Manipulate[Plot[Sin[x] / x, {x, -5 Pi, 5 Pi},
  PlotRange →
   All, PlotStyle → {Red, Thick}, AxesLabel →
   {"x",
    "sin(x)/x"}, BaseStyle → {Large,
    FontFamily →
     "Helvetica", Italic}, TicksStyle →
   {{Medium,
     Blue}, {Medium, RGBColor[0.5, 0.2, 0]}},
  Mesh →
   All, MeshStyle → {Black, PointSize[0.015]},
  MaxRecursion →
   recursion, PlotPoints → plotpoints],
{{recursion,
  3}, 1, 15, 1}, {{plotpoints, 4}, 2, 12, 1}]
```

3

1: The option `Mesh→All` shows the points where `Plot` made numerical evaluations. Note that the points are not equally spaced, but are adapted to the plot (in this case, to the curvature). `MeshStyle` permits specification of how the mesh is visualized.

2: A simple way to control the mesh is with `PlotPoints` (which specifies how many points to sample initially) and `MaxRecursion` (which specifies how many times to try to optimize the adaptation of the points on the curve).

3: This is a simple example of using `Manipulate` to change `PlotPoints` and `MaxRecursion` interactively. Here, both of the options point to variables (`recursion` and `plotpoints`) that can be adjusted via a graphical interface.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Multiple Curves, Filling, and Excluding Points

Here, simple examples of plotting several curves at the same time, of filling between curves, or between curves and the axis, and of telling plot to ignore certain points, are demonstrated.

```
Plot[Sin[x] / x, {x, -5 Pi, 5 Pi},
 PlotRange → {-0.25, 1.25},
 PlotStyle → {Red, Thick},
 TicksStyle → {{Medium, Blue},
   {Medium, RGBColor[0.5, 0.2, 0]}},
 Filling → Automatic]
```

1

Combining several curves

```
Plot[{Sin[x] / x, Tan[x] / x},
 {x, -5 Pi, 5 Pi}, BaseStyle → {Thick}]
```

2

```
Plot[{Sin[x] / x, Tan[x] / x},
 {x, -5 Pi, 5 Pi}, PlotStyle → {{Red, Thick},
   {Hue[0.3, 1, .5], Thickness[0.005]}}]
```

3

Removing points with Exclusions

```
Plot[Tan[x] / x, {x, -5 Pi, 5 Pi},
 BaseStyle → {Thick, Medium},
 Exclusions → {-Pi / 2, Pi / 2}]
```

4

```
Plot[Tan[x] / x, {x, -5 Pi, 5 Pi},
 BaseStyle → {Thick, Medium}, Exclusions →
  Table[p, {p, -9 Pi / 2, 9 Pi / 2, Pi}]]
```

5

Multiple curves with exclusions

```
Plot[{Sin[x] / x, Tan[x] / x}, {x, -5 Pi, 5 Pi},
 PlotStyle → {{Red, Thick}, {Hue[0.3, 1, .5],
    Thickness[0.005]}}, Exclusions →
  Table[p, {p, -9 Pi / 2, 9 Pi / 2, Pi}]]
```

6

Filling between curves

```
Plot[{Sin[x] / x, Tan[x] / x},
 {x, -5 Pi, 5 Pi}, PlotStyle → {{Red, Thick},
   {Hue[0.3, 1, .5], Thickness[0.005]}},
 PlotRange → {-0.25, 1.25}, Exclusions →
  Table[p, {p, -9 Pi / 2, 9 Pi / 2, Pi / 2}],
 Filling → {2 → {{1}, {RGBColor[1, 0, 0, 0.2],
      RGBColor[0, 0, 1, 0.2]}}}]
```

7

1: Simple filling to the x-axis can be produced with `Filling→Automatic`.

2: When `Plot` gets a list of expressions as its first argument, it will superimpose the curves obtained from each. The curves' colors are chosen automatically, but can be specified. (n.b., if you find that the colors are not changing as you'd expect, try calling `Evaluate` on the list.) In this example, a vertical line appears for the $\tan(x)/x$ function where the values change as $\pm\infty$. To change the appearance of each curve, a list containing a style-directive list for each curve is used for the `PlotStyle` option. The first style, `{Red,Thick}`, uses simple directives for basic, easy-to-remember, control; the second style uses higher precision control with `Hue` and `Thickness`.

3: The singularities in the function produce vertical lines in the above plots. To remove these features, the option `Exclusions` can get a list of points where the curve should be sliced and not evaluated.

4: Here, we use `Table` to produce a list of all the singularities in $\tan(x)/x$. This list is passed via `Exclusions`.

7: This is a more complex example of filling: here we ask for the filling to take place between the second curve and the first—and to use different filling styles when the first curve lies above or below the second curve.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Plotting Two Dimensional Parametric Curves and Mapped Regions

Here are simple examples of using `ParametricPlot` to plot functions for curves in the form $(x(t), y(t))$ and regions in the form $(x(s,t), y(s,t))$.

1
```
? ParametricPlot
```

2
```
MagicCircles[ t_, n_] :=
 { Cos[n t - Pi + 2 Pi Quotient[n t, 2 Pi] / n] +
   Cos[2 Pi Quotient[n t, 2 Pi] / n],
  Sin[n t - Pi + 2 Pi Quotient[n t, 2 Pi] / n] +
   Sin[2 Pi Quotient[n t, 2 Pi] / n]}
```

3
```
ParametricPlot[
 MagicCircles[t, 5], {t, 0, 2 Pi},
 PlotStyle → Thick, PlotRange → All]
```

4
```
Manipulate[
 ParametricPlot[MagicCircles[t, ncirc],
  {t, 0, lastp}, PlotStyle → Thick,
  PlotPoints → 6 ncirc, Axes → False],
 {{ncirc, 3}, 1, 36, 1},
 {{lastp, 2 Pi}, 0.0001, 2 Pi}]
```

5
```
OrbitOrbit[ r_, t_, n_] :=
 { r Cos[n t] + Cos[t], r Sin[n t] + Sin[t]}
```

6
```
ParametricPlot[
 Evaluate[OrbitOrbit[.5, t, 12]],
 {t, -Pi, Pi}, PlotStyle → Thick]
```

Now we let both r and t vary. Some regions in the disk r ∈ (0.25,0.75) don't get covered, and others get covered one or more times.

7
```
ParametricPlot[Evaluate[OrbitOrbit[r, t, 12]],
 {t, -Pi, Pi}, {r, .25, .75},
 PlotStyle → {Thick, Red},
 Mesh → False, PlotPoints → 72]
```

8
```
ParametricPlot[Evaluate[OrbitOrbit[r, t, 6]],
 {t, -Pi, Pi}, {r, .25, .9},
 PlotStyle → {Thick, Red},
 Mesh → False, PlotPoints → 36,
 ColorFunction → (Hue[#3, 1, 1, 0.25] &)]
```

2: A function, *MagicCircles[t,n]* , is defined to produce some interesting parametric plots. It returns data in the form `{x(t),y(t)}` where $t \in (0, 2\pi)$. The second argument, `n`, is a parameter which will determine how many circles get drawn.

3: `ParametricPlot` is used with the `PlotStyle` option set for thick curves, and `PlotRange` set to `All`.

4: Here, we make `ParametricPlot` the first argument to `Manipulate` so that the number of circles can be varied (note, that we force `n` to iterate over integers). The trajectory of the curve can be visualized here by interactively changing the upper bound of t with `lastp`.

5: We cook up another function, *OrbitOrbit[r,t,n]* , to demonstrate filling a region. Data is returned in the form `{x(r,t),y(r,t)}`, and `n` is a parameter.

6: If `r` is fixed, `ParametricPlot` produces a curve as before.

7: Letting both `r` and `t` vary, produces a two-dimensional region—one might think of the region as the set of all the curves for different `r`.

8: This is a slightly advanced example where we use a *pure function* for the `ColorFunction` option. I'm including this example because I think it's pretty.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Simple Plots of Data

One of MATHEMATICA® 's integrated data resources, `ElementData`, is used to demonstrate plotting of discrete data.

The next command uses Mathematicas Integrated Data Resources, it will not retrieve the data unless you have an active internet connection

```
ElementData[]
```
1

Here is a list of properties that we can access from ElementData

```
ElementData["Properties"]
```
2

However, one should always question the provenence and accuracy of data... Let's make a sanity check: the stable phase of carbon at STP is graphite which is hexagonal (but not close packed).

```
ElementData[6, "StandardName"]
ElementData[6, "CrystalStructure"]
```
3

We create a list of the densities of the first one hundred elements. Data that is missing is reported with Missing[NotAvailable] or Missing[Unknown].

```
Densities =
 Table[ElementData[i, "Density"], {i, 1, 100}]
```
4

```
ListPlot[Densities]
```
5

```
ListPlot[Densities,
 BaseStyle → {Large, FontFamily → "Helvetica",
   PointSize[0.025]}]
```
6

```
ListLinePlot[Densities,
 BaseStyle → {Large, FontFamily → "Helvetica",
   PointSize[0.025]}]
ListPlot[Densities, BaseStyle →
  {Large, FontFamily → "Helvetica",
   PointSize[0.025]}, Joined → True]
```
7

To see the data, we use the PlotMarkers Option.

```
ListLinePlot[Densities,
 BaseStyle → {Large, FontFamily → "Helvetica",
   PointSize[0.025]},
 PlotMarkers → Automatic, AxesLabel →
  {"Element Number", "Density (MKS)"},
 ImageSize → Large]
```
8

1: `ElementData` will download physical data for the elements via an internet connection. *This command won't work if you do not have an active connection.* However, similar data remain in the now obsolete `ChemicalElements` package.

2: This produces a list of properties that are available. One should always suspect data sources! The stable form of carbon and graphite, is hexagonal but not close-packed.

3: For example, this is how to access properties for carbon.

4: `Table` is used with `ElementData` to produce a list, `Densities`, of the first 100 elements for subsequent use. Missing data are indicated with the function `Missing`.

5: Simply using `ListPlot` produces an indexed scatter plot.

6: Like `Plot`, we can use options in `ListPlot` and `ListLinePlot` to change the appearance of the graphic.

7: A set of line segments are drawn (approximating a curve) in `ListLinePlot`—which is equivalent to using `ListPlot` with the option `PlotJoined` set to `True`.

8: Using the `PlotMarkers` option, both the data and the line segments are visualized.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Getting More out of Displayed Data: Screen Interaction

Putting too much information on a single data graphic can make it difficult to understand. Using pop-up windows with the mouse can be a nice way to improve graphical information flow. Here, we show how this can be done using `Tooltip`. In these examples, *where* the extra information appears can be altered by replacing `Tooltip` with `StatusArea`, `Annotation`, or `PopupWindow`.

Example with Tooltip to make graphics interactive----put your mouse over a point and you get a pop-up with more information

```
ListLinePlot[Tooltip[Densities],
 BaseStyle → {Large, FontFamily → "Helvetica",
   PointSize[0.025]},
 PlotMarkers → Automatic, AxesLabel →
  {"Element Number", "Density (MKS)"},
 ImageSize → Large]
```

1

This is a slightly more complicated example of Tooltip. We create a data structure with {x(i),y(i)} = {density(i), bulkmodulus(i)} and then tell Tooltip to pop-up the element's symbol when the mouse is over it.

```
ListPlot[
 Table[Tooltip[{ElementData[i, "Density"],
    ElementData[i, "BulkModulus"]},
   ElementData[i, "Abbreviation"],
   LabelStyle → {Large}], {i, 1, 100}],
 BaseStyle → {Large, FontFamily → "Helvetica",
   PointSize[0.025]}, PlotMarkers → Automatic,
 AxesLabel → {"Density", "Bulk Modulus"},
 PlotLabel → "MKS Units",
 ImageSize → Full]
```

2

1: This is a simple example of `Tooltip`: wrapping the first argument to `ListPlot` or `ListLinePlot` inside `Tooltip` will show the value of each data point when the mouse is over it.

2: I like this example which uses `Tooltip[{xi,yi},labeli]` to produce an interesting way to pick material properties. Suppose we were interested in finding materials that are very stiff (large bulk modulus) but not very heavy (low density)—plotting modulus versus density will identify "interesting" elements in the northwest region of the plot. Using `Tooltip` with `ElementData[i,''Abbreviation'']` allows us to explore element properties without cluttering up the plot. I use `LabelStyle` as an option for `Tooltip` and `ImageSize` as an option for `ListPlot` to make things readable on the display.

3.016

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Graphical Data Exploration, continued

We use `BarChart` and `PieChart` in the `BarCharts` and `PieCharts` packages to explore the relative abundances of different crystal structures among the elements. A three-dimensional histogram of elements selected by their melting points and densities is produced with `Histogram3D` from the `Histograms` package.

Here we do a small exercise to get a graphical representation of which Crystal Structures the elements form, and represent the frequency of each type. First we create a list of known elemental crystal structures for the first 100 elements.

```
CrystalStructures = Table[ElementData[
   i, "CrystalStructure"], {i, 100}]
```
1

```
UniqueStructures = Tally[Cases[
   CrystalStructures, Except[Missing[_]]]]
MatrixForm[UniqueStructures]
```
2

Here is a bar chart showing the frequency of crystal structures.

```
Needs["BarCharts`"]
BarChart[Transpose[UniqueStructures][[2]],
 BarLabels ->
  Transpose[UniqueStructures][[1]],
 BaseStyle → {Large, FontFamily → "Helvetica"},
 BarOrientation → Horizontal, ImageSize → Full]
```
3

```
Needs["PieCharts`"]
PieChart[Transpose[UniqueStructures][[2]],
 PieLabels ->
  Transpose[UniqueStructures][[1]],
 BaseStyle → {Large, FontFamily → "Helvetica"},
 ImageSize → Full]
```
4

As a last example, we produce a 3D histogram. The height of each bar corresponds to the number of elements in a range of melting points and range of densities.

```
Needs["Histograms`"]
histdata = DeleteCases[Table[
   {ElementData[i, "AbsoluteMeltingPoint"],
    ElementData[i, "Density"]}, {i, 100}],
  {Missing[_], _} | {_, Missing[_]}]
Histogram3D[histdata, AxesLabel →
  {"Melting Point", "Density", "Number"},
 HistogramCategories → {16, 24}]
```
5

1: `CrystalStructures` will be a list of the crystal structures of the most stable solid phase. (I am not sure what is meant by most stable—this is ambiguous, but that is what it says in the documentation)

2: `UniqueStructures` will be a list of pairs—each item will be comprised of a crystal structure and how many times it appears. We use `Cases` to remove missing data by using a pattern, and then use `Tally` to create the data structure.

3: Because `BarChart` needs data of the form `{y1, y2, ...}`, we need to manipulate the data. To get the data, `Transpose` will put the abundances into the second row, which is also the list we need. We use the first row of the transpose for the `BarLabels` option. The plot is easier to read if horizontal, so we use the `BarOrientation` option.

4: Here we simply replace the barchart with `PieChart`.

5: As a final example, we create a histogram of elements with similar densities and melting points. We use a pattern with an "or" in `Cases` to remove missing data with `DeleteCases`, because we cannot plot data where either the density *or* the melting point is missing.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Three-Dimensional Graphics

Here we show examples of three-dimensional graphics, although it would be better to say, 3D graphics projected onto a 2D screen.

```
EPot[x_, y_, z_, xo_, yo_] :=
                1
-------------------------------------
√((x - xo)^2 + (y - yo)^2 + z^2)
```
1

```
SheetOLatticeCharge[x_, y_, z_] :=
 Sum[EPot[x, y, z, xo, yo],
  {xo, -5, 5}, {yo, -5, 5}]
```
2

SheetOLatticeCharge *represents the electric field produced by an 11 by 11 array of point charges arranged on the x-y plane at z = 0. The following command evaluates and plots the field variation in the plane z = 0.25:*

```
Plot3D[
 Evaluate[SheetOLatticeCharge[x, y, 0.25]],
 {x, -6, 6}, {y, -6, 6}]
```
3

Note below how ***theplot*** *is set to contain the output of the Plot3D command---it is now a symbol assigned to a graphics object. The number of plotpoints is increased so that we can resolve all the bumps. This will take a while to compute on most machines.*

```
theplot = Plot3D[
  Evaluate[SheetOLatticeCharge[x, y, 0.25]],
  {x, -6, 6}, {y, -6, 6}, PlotPoints → 60]
```
4

This demonstrates the use of RegionFunction plot option which is pure function. Here, only the region inside a cylinder with radius 9 ($x^2 + y^2 \le 9^2$) is plotted.

```
Plot3D[
 Evaluate[SheetOLatticeCharge[x, y, 0.25]],
 {x, -9, 9}, {y, -9, 9}, PlotPoints → 60,
 RegionFunction → (#1^2 + #2^2 ≤ 81 &)]
```
5

This demonstrates the use of the ColorFunction plot option which is pure function. Here we use one of Mathematica ColorData functions.

```
Plot3D[
 Evaluate[SheetOLatticeCharge[x, y, 0.25]],
 {x, -9, 9}, {y, -9, 9}, PlotPoints → 60,
 RegionFunction → (#1^2 + #2^2 ≤ 81 &),
 ColorFunction →
  (ColorData["TemperatureMap"][#3] &)]
```
6

1: This is the electrostatic potential as a function of (x, y, z) due to a single positive charge located at $(x_o, y_o, z = 0)$ (i.e., anywhere on the $z = 0$ plane).

2: By summing over a square lattice of unit charges, this function (*SheetOLatticeCharge*) computes the electrostatic potential over a 11×11 square-lattice of point-charges centered on the z-plane as a function of x, y, and z.

3: `Plot3D` plots data of the form $f(x, y)$ (f is the height above a point (x, y)). Therefore, we must fix one of the coordinates; here we visualize the electrostatic potential at a fixed height ($z = 0.25$). Note that the bounds for both the "horizontal" and "into-screen" coordinates need to be specified.

You can rotate the graphics by dragging the mouse over the surface, translate by dragging with the shift-key held down, and zoom with the alt-key held down.

4: With sufficiently many `PlotPoints`, the structure of the potential at a fixed distance $z = 0.25$ is made apparent. The finer details are not resolved at lower resolutions, but using 60 points in each direction may be overkill and this will be slow on older computers and may not fit on machines with little memory.

5: `RegionFunction` is new as of MATHEMATICA® 6. This is an advanced examples, but it demonstrates how one can plot over non-rectangular domains.

6: As a last example, the use of the new `ColorData` functions for the `ColorFunction` option is demonstrated.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

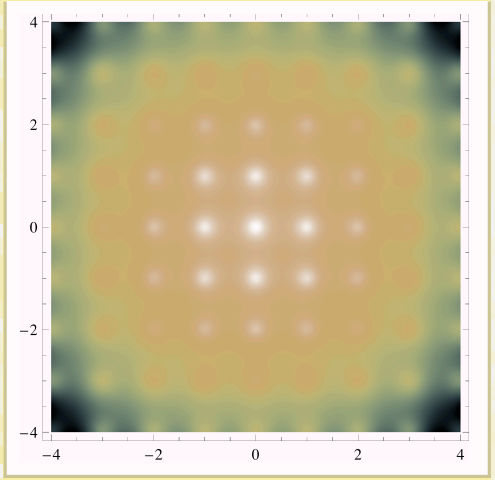
Colors and Contours: Three-Dimensional Graphics in Two Dimensions

Three dimensions can also be visualized by drawing level sets (as in a topographical map) or by drawing colors (as in a relief map). The data burden is usually much smaller than a 3D graphics object, is sometimes easier to interpret, and is certainly easier to publish.

```
theconplot = ContourPlot[
  Evaluate[SheetOLatticeCharge[x, y, 0.25]],
  {x, -6, 6}, {y, -6, 6}, PlotPoints → 32]
```
1

```
theconplot = ContourPlot[
  Evaluate[SheetOLatticeCharge[x, y, 0.25]],
  {x, -4, 4}, {y, -4, 4}, PlotPoints → 50,
  ColorFunction → Hue, Contours → 24]
```
2

```
thedenplot = DensityPlot[
  Evaluate[SheetOLatticeCharge[x, y, 0.25]],
  {x, -4, 4}, {y, -4, 4},
  PlotPoints → 50, ColorFunction →
   ColorData["GreenBrownTerrain"]]
```
3



1: We reproduce the 3D graphics object for the sheet of electric charges using `ContourPlot`. Here, the number of contours are picked arbitrarily, but `PlotPoints` has to be increased to resolve details of the function. Moving the mouse over one of the contours will give a pop-up window for the value along that contour.

2: In the representation above, we might conclude that a positive charge (such as a hole) confined to $z = 0.25$ could not be "trapped" because no minima are obvious. Increasing the number of contours with the `Contours` option improves the resolution so that local minima can be observed. Here we pass `Hue` to the `ColorFunction` option; however, I don't find this satisfactory because both the largest and the smallest values are red. In other words, the color scaling runs completely around the outside of a color wheel and ends up where it started.

Unless options are sent requesting otherwise, the values of the plot will be scaled so that the maximum and minimum values are 1 and 0. Thus, two plots would look the same whether the differences are very small or very large. This feature is controlled by `ColorFunctionScaling`.

3: Here, instead of a single color decorating the region between two neighboring contours, a color is plotted directly indicating the "height" of the function. `ColorData` is used with `GreenBrownTerrain` so that the high potentials look like snow-covered peaks and lower potentials look like green river-deltas.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Graphics Primitives, Drawing on Graphics, and Combining Graphical Objects

Here, examples of placing *Graphics Primitives* into a *Graphics Object* are demonstrated by direct means: by a drawing tool, and by sequential combination.

It can be useful to be able to build up arbitrary graphics objects piece-by-piece using simple "graphics primitives" like ***Circle****:*

```
thecirc = Graphics[Circle[{2, 2}, 1.5]]
```
1

```
Show[thecirc, Axes -> True]
```
2

```
Show[thecirc, Axes -> True,
 AxesOrigin → {0, 0}, AspectRatio → 1]
```
3

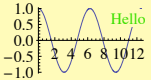
Now we take a simple plot...

```
cosplot = Plot[Cos[x], {x, 0, 4 Pi}]
```
4

Adding Graphics Primitives to Plots (or other graphics objects) using the built-in Drawing Tool

Mathematica6 now has a simple drawing editor that allows you add text, arrows, lines, and shapes to existing graphics. To do this, select the previous graphics output for the cosine plot. While the graphics are selected, use the Menu Item "Drawing Tools" under Graphics. After you have added shapes, text, etc.. move the cursor to the left of the selected graphics object and type a symbol (below, I used "thenewplot") for the new (combined) graphics object to be assigned to.

```
thenewplot =
```



5

```
thenewplot
```
6

Combining Graphical Objects using Show.

and overlay some text in places of our own choosing...

```
Show[cosplot, Graphics[
  Text["One Wavelength", {2 Pi, 0.5}]],
 Graphics[Text["Two\nWavelengths",
   {4 Pi, 0.5}]], PlotRange → All]
```
7

```
Show[thenewplot, Graphics[
  Text["One Wavelength", {2 Pi, 1.1}]],
 Graphics[Text["Two Wavelengths",
   {4 Pi, 1.1}]], PlotRange → All]
```
8

1: A `Circle` is a graphics primitive, and making a primitive an argument to `Graphics` returns a "Graphics Object." When a graphics object is output, graphics appear. The graphical output can be suppressed by a trailing semicolon. In this case, `thecirc` is assigned to the graphics object and it is displayed. If a trailing semicolon appears (e.g., a unit circle `thecirc = Graphics[Circle[]];`), then the assignment is made to `thecirc`, but no graphics are sent to the display.

2–3: Additional options can be added to a graphics object with `Show`. The result is a new graphics object.

4: Here we create a graphics object and assign it to the symbol `cosplot` by simply using `Plot`.

5: If the mouse is clicked on the display of the graphics object, then it can be edited just like input. Clicking to the left of the object allows you to type a symbol for assignment to the graphics object. Shown here is the result of assigning a graphic to `thenewplot`. If the graphic is selected, then a *Drawing Tools Widget* can be pulled up under the Graphics menu item. With the widget, other primitives such as text, lines, arrows, and shapes can be combined. When the expression is evaluated, the combined graphics will be assigned to `thenewplot`.

7–8: Here, `Show` is used to add text via a graphics primitive to the original plot and to the new combined graphics object.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

A Worked Example: The Two-Dimensional Wulff Construction

The Wulff construction is a famous thermodynamic construction that predicts the equilibrium enclosing-surface of an anisotropic isolated body. The anisotropic surface tension, $\gamma(\hat{n})$, is the amount of work (per unit area) required to produce a planar surface with outward normal $\hat{n}$. The construction proceeds by drawing a bisecting plane at each point of the polar plot $\gamma(\hat{n})\hat{n}$. The interior of all bisectors is the resulting *Wulff shape*.

A working example of the Wulff construction for a $\gamma(\theta)$ in two dimensions is produced here.

This next example shows a clever way to perform a famous thermodynamic graphical construction called the Wulff construction.

```
wulffline[{x_, y_}, wulfflength_] :=
 Module[{θ, wulffhalf = wulfflength * 0.5,
   x1, x2, y1, y2}, θ = ArcTan[x, y];
  x1 = x + wulffhalf * Cos[θ + Pi / 2];
  x2 =  x + wulffhalf * Cos[θ - Pi / 2];
  y1 = y + wulffhalf * Sin[θ + Pi / 2];
  y2 = y + wulffhalf * Sin[θ - Pi / 2];
  Graphics[Line[{{x1, y1}, {x2, y2}}]]
 ]
```

1

```
gammaplot[ theta_ , anisotropy_ , nfold_] :=
 {Cos[theta] + anisotropy *
    Cos[(nfold + 1) * theta], Sin[theta] +
   anisotropy * Sin[(nfold + 1) * theta]}
```

2

```
GammaPlot =
 ParametricPlot[gammaplot[t, 0.1, 4],
  {t, 0, 2 Pi}, PlotStyle →
   {{Thickness[0.01], RGBColor[1, 0, 0]}}]
```

3

```
Show[Table[wulffline[gammaplot[t, 0.1, 4], 2],
  {t, 0, 2 Pi, 2 Pi / 100}], GammaPlot]
```

4

```
ToutesDesLoups[anisotropy_, nfold_] :=
 Module[{GammaPlot }, GammaPlot =
   ParametricPlot[gammaplot[t, anisotropy,
     nfold], {t, 0, 2 Pi}, PlotStyle →
    {{Thickness[0.01], RGBColor[1, 0, 0]}}];
  Show[Table[wulffline[gammaplot[
     t, anisotropy, nfold], 3],
   {t, 0, 2 Pi, 2 Pi / 100}], GammaPlot]]

Manipulate[ToutesDesLoups[aniso, nfold],
 {{aniso, 0.1}, -0.9, 0.9},
 {{nfold, 6}, 2, 16, 1}]
```

5

1: This function takes a point `{x,y}` as an argument and then returns a graphics object of a line of specified length. The line is the perpendicular bisector required by the Wulff construction.

2: This is an example $\gamma(\hat{n})$ with the surface tension being smaller in the $\langle 11 \rangle$-directions (if the `anisotropy` parameter is positive).

3: A particular instance of a γ-plot is assigned to `GammaPlot`.

4: `Table` is used to produce a list of graphics objects by calling *wulffline* function at one hundred points on the γ-plot. The equilibrium shape is the interior of all the curves and the γ-plot from which it derives is superimposed by collecting all the graphics together with `Show`.

5: All the above steps are collected together and bundled into a `Module` to produce a single visualization function, *ToutesDesLoups* . The function depends on the prior definition of `gammaplot[t,α,n]`.

6: Here, *ToutesDesLoups* is used as the argument to `Manipulate` to visualize the effect of changing the anisotropy factor and the n-fold axis.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Graphical Animation: Using Time as a Dimension in Visualization

Animations can be very effective tools to illustrate time-dependent phenomena in scientific presentations. Animations are sequences of multiple images—called frames—that are written to the screen interatively at a constant rate: if one second of real time is represented by N frames, then a real-time animation would display a new image every $1/N$ seconds.

There are two important practical considerations for computer animation:

frame size An image is a an array of pixels, each of which is represented as a color. The amount of memory each color requires depends on the current image depth, but this number is typically 2-5 bytes. Typical video frames contain 1024×768 pixel images which corresponds to about 2.5 MBytes/image and shown at 30 frames per second corresponding to about 4.5 GBytes/minute. Storage and editing of video is probably done at higher spatial and temporal resolution. Each frame must be read from a source—such as a hard disk—and transfered to the graphical memory (VRAM) before the screen can be redrawn with a new image. Therefore, along with storage space the rate of memory transfer becomes a practical issue when constructing an animation.

animation rate Humans are fairly good at extrapolating action between sequential images. It depends on the difference between sequential images, but animation rates below about 10 frames per second begin to appear jerky. Older Disney-type cartoons were typically displayed at about 15 frames per second, video is displayed at 30 frames per second. Animation rates above about 75 frames per second yield no additional perceptable "smoothness." The upper bound on computer displays is typically 60 hertz.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Animation

Animations are a nice way to visualize an extra dimension, like time. An animation is composed of a sequence of displayed graphics (frames) that are displayed iteratively. Animations are fairly easy to create–and can be great fun.

```
Fxt[x_, t_] :=
 Sin[3 (x + 10 - t)] Exp[-(x + 10 - t)^2] -
  Sin[3 (x - 10 + t)] Exp[-(x - 10 + t)^2]
```

1

```
Animate[
 Plot[Fxt[xvar, timevar], {xvar, -15, 15},
  PlotRange → {-1, 1}, PlotStyle → {Thick, Red},
  Filling → Axis, FillingStyle →
   {RGBColor[0, 0.5, 0, 0.5], RGBColor[
     0, 0, 0.5, 0.5]}], {timevar, 0, 25}]
```

2

This is the solution to the temperature evolution equation (the diffusion equation) for a square of length L initially at 500K embedded in a plate initially at 100K , κ is the themal diffusivity (units length^2/time). We introduce a "normalized" time and space variables variable $\tau = \kappa t/L^2$ and ξ = x/L and η=y/L

```
TempSquare =

 100 + 400 Integrate[Exp[-((x-xo)^2 + (y-yo)^2)/(4 κ t)]/(4 π κ t),

   {xo, -L/2, L/2}, {yo, -L/2, L/2}]

NormalizeRules = {t → τ L^2/κ, x → ξ L,
   y → η L, xo → ξo L, yo → ηo L};
TempSquare = Simplify[TempSquare /.
   NormalizeRules, Assumptions → κ > 0 && L > 0]
```

3

We divide by 500 so that the temperatures should scale between zero and one, and then use ColorFunctionScaling->False so that the colors are consistent over time.

```
ListAnimate[
 Table[Plot3D[TempSquare / 500, {η, -1, 1},
   {ξ, -1, 1}, PlotRange → {0, 1}, PlotPoints →
    50, ColorFunction → "TemperatureMap",
   ColorFunctionScaling → False],
  {τ, 0.001, .1, 0.002}]]
```

4

1: We will create a simple animation by cooking up a function $f(x,t)$ and then plotting it for a range of x and for a sequence of t's.

2: This plot would be the frame associated with $t = 0$.

3: Using `Plot` as the argument to `Animate` produces the animation. Note, `xvar` 'belongs' to `Plot` while `timevar` belongs to `Animate`.
Can you imagine what the animation would look like if we animated over x and plotted over t? No? Try it!

4: We will produce a three-dimensional animation of how the temperature would change in a flat plate, if at time $t = 0$ there is a square at a different temperature than the rest of the plate. The governing partial differential equation is $\partial T/\partial t = \kappa \nabla^2 T$ and for initial conditions $T(x, y, t = 0) = 500$ when $-L/2 < x, y < L/2$ and $T = 100$ otherwise, the closed form solution can be expressed as an integral. To make a plot, we must send a function that can be evaluated numerically. To do this, we must *non-dimensionalize variables* (also known, as *dimensional scaling* or *normalizing variables*). This is done by dividing variables having physical units (such as x), with a characteristic quantity in the model that has the same physical units (here, we will use the model's length L to produce a dimensionless variable $\xi = x/L$) *NormalizeRules* is a set of rules that can be applied to our physical problem. After the normalization rules are applied, the properly scaled solution should be a non-dimensional temperature-quantity as a function of non-dimensional space- and time-quantities.

5: Finally, we will use `Plot3D` inside `ListAnimate`. `Plot3D`'s argument is scaled by dividing by the maximum temperature, so that all temperature-like quantities scale between zero and one. We turn off `ColorFunctionScaling` so that the 'meaning' of each color remains constant in the animation. `ListAnimate` takes a list of frames that are produced via `Table`.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

An Example of Animating a Random Walk

A *random walk* process is an important concept in diffusion and other statistical phenomena. Functions to simulate a random walk in two dimensions are constructed and then visualized with animations.

```
randomwalk[0] = {0, {0, 0}}
```
1

```
randomwalk[nstep_Integer?Positive] :=
 randomwalk[nstep] =
  {nstep, randomwalk[nstep - 1][[2]] +
    RandomReal[0.5] {Cos[
       theta = RandomReal[2 π]], Sin[theta]}}
```
2

Create a function that returns a graphic object putting the step number at the correct place:

```
gtext[nstep_Integer?NonNegative] :=
  gtext[nstep] = Graphics[
    Text[ToString[randomwalk[nstep][[1]]],
     randomwalk[nstep][[2]]]];
```
3

```
locations = Show[Table[gtext[i], {i, 0, 100}],
  PlotRange → All, AspectRatio → 1]
```
4

```
gline[nstep_Integer] := gline[nstep] =
   Graphics[Line[{randomwalk[nstep - 1][[2]],
      randomwalk[nstep][[2]]}]];
```
5

```
Show[Table[gtext[i], {i, 0, 100}],
 Table[gline[j], {j, 1, 100}],
 PlotRange → All, AspectRatio → 1]
```
6

```
Animate[Show[gtext[i], gline[i]],
 {i, 1, 49, 1}]
```
7

If we use the PlotRange from a graphical object that contains all the points, we can fix the framesize, we use AbsoluteOptions

```
prange =
 PlotRange /. AbsoluteOptions[locations]
```
8

```
Animate[Show[gtext[i], gline[i],
  PlotRange → prange], {i, 1, 100, 1}]
```
9

```
Animate[
 Show[Table[{gtext[i], gline[i]}, {i, 1, j}],
  PlotRange → prange], {j, 2, 100}]
```
10

1–2: This is a recursive function that simulates a random walk process. Each step in the random walk is recorded as a list structure, { {`iteration number`}, { `x` , `y` }}, and assigned to *randomwalk* [iteration number]. For each step (or iteration), a number between 0 and 1/2 is selected (for the magnitude of the displacement), and an angle between 0 and 2π is selected (for the direction), with each of these numbers being selected randomly from a uniform distribution (using `RandomReal`). The function includes an assignment, so all previous values are stored in memory.

3: The function *gtext* calls *randomwalk* to create a text graphics-object located at the position corresponding to `nstep`.

4: This shows the history of a random walk after 50 iterations by combining the graphics objects created by *gtext* . The resulting graphics object gets assigned, because we will use some information contained in it later.

5: To improve the physical interpretation of the previous graphic, it would be an aid to the eye if the individual jumps were indicated. To do this, the function *gline* calls *randomwalk* to create a line graphics-object connecting the position corresponding to `nstep` to its previous position.

7: Thus, we could animate by combining the line and the text with `Show` and using that as the argument to `Animate`. *However, this result will be unsatisfactory because the "length scale" of each frame will not be consistent.*

8: To solve this problem, we find the bounds of a graphics object (`locations`) that contains all the points, and then query its `PlotRange` using `AbsoluteOptions` and this is assigned to a symbol `prange`.

9: The animation is consistent now, but could still use some improvement.

10: Here, we animate the graphics object that also contains the history of prior jumps. This is not a terribly efficient way to do this because we recreate the early steps many times over, but it works for our purposes.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Worked Example (part A): Visualizing the Spinodal and Common Tangent Construction

The spinodal and common tangent construction is a fundamental thermodynamic concept used for the creation of an alloy phase diagram from molar-free energies. This construction appears repeatedly in studies of materials.

An example of visualizing this construction as a function of temperature will be worked out in detail for the case of a single curve and a binary alloy.

First, we will work out all the steps in detail that are used to build up a single visualization, and then we will collect it all together in a reusable function.

A prototype molar free energy of mixing using the same xlogx function for the ideal entropy of mixing terms. The temperature term is a scaled energy (RT), and it is assumed that enthalpies have been scaled so that the temperatures of interest (if there are any) are between T=0 and T=10.

```
xlogx[0] =
 xlogx[1] = xlogx[0.0] = xlogx[1.0] = 0;
xlogx[x_] := x Log[x]
Gmolar[X_, T_] :=
 5 X (1 - X) + T (xlogx[X] + xlogx[1 - X]) + X / 2
```

1

Here is the shape of our prototype free energy at T=3/2

```
p1 = Plot[Gmolar[x, 3 / 2],
  {x, 0, 1}, PlotStyle → Thick]
```

2

We will need the bounds of the above graphics object:

```
{{graphxmin, graphxmax},
  {graphymin, graphymax}} =
 PlotRange /. AbsoluteOptions[p1, PlotRange]
```

3

First let's determine where the spinodal region (by finding where the second derivative with respect to composition is negative

```
ddg = D[Gmolar[x, 3 / 2], {x, 2}]
```

4

Then, use RegionPlot to illustrate the range over which spinodal decomposition is spontaneous

```
p2 = RegionPlot[ddg < 0,
  {x, graphxmin, graphxmax},
  {T, graphymin, graphymax},
  PlotStyle → RGBColor[0, 1, .5, 0.1]]
```

5

Show them both together to identify the spinodal region

```
Show[p1, p2]
```

6

1: We cook up a prototypical molar free-energy as a function of molar composition, `X`, and temperature `T`. The $x \log x$ terms are calculated with a handy function, *xlogx* , which will handle the zeroes without numerical difficulty at `0 Log[0]`.

2: The molar free-energy is plotted at a particular temperature ($T = 1.5$) and assigned to a symbol, `p1`.

3: We will need the bounds of the plot to create other graphical objects. We grab the bounds with `AbsoluteOptions` and assign them to variables using a handy assignment construction `{a,b} = List`.

4: The spinodal region is the easiest to visualize—it is the region where the second derivative of the molar free-energy is negative. The second derivative is assigned to `ddg`.

5: `RegionPlot` evaluates its first argument over a square region and fills where the argument is true. It is exactly what we need in order to visualize the spinodal region. We use the bounds that we calculated from the free energy curve as the bounds for `RegionPlot`.

6: Showing both plots together, we visualize the spinodal region.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Worked Example (part B): Visualizing the Spinodal and Common Tangent Construction

The common tangent is any finite line segment that touches the molar free-energy at two points which have the same derivative. For phase diagrams, we are interested only in lower common tangents (i.e., lines that touch the molar free-energy, but always lie below all values). One can picture the common tangent by imagining that an elastic string is stretched along a molar free-energy curve; the common tangents are where the string pulls away from the the curves.

The common tangent is related to the *convex hull* that appears in computational geometry.

We can use the ConvexHull to find the common tangent lines; this function is in the Computational Geometry Package.

```
<< ComputationalGeometry`
```
1

First we compute a list of values along the molar free energy curve, then compute those that lie outside the common tangent(s) (i.e., the convex hull). Because the points are given in order, we might as well sort them on the way back out. Note, the convex hull program gives the indices of the vertices that are on the hull.

```
npoints = 100;
gvals = Table[{x, Gmolar[x, 3 / 2]},
   {x, 0, 1, 1 / N[npoints - 1]}];
```
2

We only want the lower convex hull; therefore we add some "fictive" points to the beginning and the end of the data. The the fictive points add a rectangle to the top of the curve that should be part of the computed convex hull.

```
gmax = Max[Transpose[gvals][[2]]];
PrependTo[gvals, {0, 10 * Abs[gmax]}];
AppendTo[gvals, {1, 10 * Abs[gmax]}];
```
3

After we compute this hull, we shift the hull by one and take off its first and last element. We strip the first and last element from the discrete values of free energy as well.

```
chull = Sort[ConvexHull[gvals]];
chull = Drop[Drop[chull - 1, 1], -1]
gvals = Drop[Drop[gvals, 1], -1]
```
4

The common tangent(s) correspond to gaps in the vertex list of the common tangent. We will use Split to find the set of continous sequences.

```
convexparts = Split[chull, (#2 - #1 < 2) &]
```
5

```
{{1, 2, 3, 4, 5, 6}, {95, 96, 97, 98, 99, 100}}
```

1: To calculate convex hulls, the `ComputationalGeometry` package is needed.

2: `ConvexHull` operates on discrete data. Discrete data are created by evaluating *Gmolar* at `npoints` evenly-spaced mesh-points. We use `Table` and assign the discrete data list to `gvals`.

3: `ConvexHull` calculates the *entire hull* (i.e., the polygon that encloses all other points), and we are only interested in the lower hull. Thus, we add a rectangle to the top of the data which is guaranteed to be part of the hull, calculate the hull and discard the upper parts. Here we use `PrependTo` to add a point ten times higher than the maximum value on the left side of the region, and use `AppendTo` to add a corresponding point to the right side of the region. We have thus added a known rectangle that we will remove later.

4: `ConvexHull` returns a *list of indices of points* from the original data. Because the original data was created in an orderly left-to-right way, we can use `Sort` to put the data in a predictable form. Because there was an additional point added at the beginning of `gvals`, we will need to shift the indices down by one (by subtracting 1 from each index), and then we use `Drop` to remove the first and last elements of both `chull` and `gvals`.

5: Thinking about the indices on the convex hull, any ordered sequence of the sorted list must be part of original discrete data and also part of the convex hull. We are interested in connecting the last point of any isolated sequence to the first point of the next sequence. We can use `Split` to find the isolated sequences.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Worked Example (part C): Visualizing the Spinodal and Common Tangent Construction

With the information contained in the convex hull data, graphical objects are created to represent the gaps in that data. The gaps coincide with the common tangents.

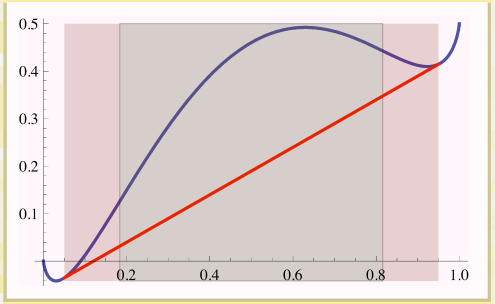
Now we create graphics objects for each of the two-phase regions (i.e., the gaps in the convex hull) and collect them all into a graphics list for subsequent display.

```
len = Length[convexparts];
graphicslist = {};
i = 1;
While[i + 1 ≤ len, leftpoint =
  gvals[[Last[ convexparts[[i]] ]]];
 rightpoint = gvals[[
   First[ convexparts[[i + 1]] ]]];
 ctline = {Red, Thick,
   Line[{leftpoint, rightpoint}]};
 twophaseregion = {RGBColor[0.5, 0, 0, 0.2],
   Rectangle[{leftpoint[[1]], graphymin},
    {rightpoint[[1]], graphymax}]};
 AppendTo[graphicslist, ctline];
 AppendTo[graphicslist, twophaseregion];
 i++
]
p3 = Graphics[graphicslist]
```

1

```
Show[p1, p2, p3]
```

2



1: We traverse the list `convexparts` and construct graphical objects corresponding to the regions of isolated sequences. Because it is possible that a curve may have any number of common tangents, we accumulate graphics primitives in a list as we encounter common tangents. A graphics object is created from the list of graphics primitives.

The number of isolated sequences is assigned to `len` and we start with an empty list `graphicslist`. Then, we loop over the list of length `len`. At each iteration in the loop, we identify the last vertex on the previous point of the convex hull sequence and the first part of the next sequence. We use those indices to extract the points on the curve that have been stored in `gvals`. With the two points, we create red lines for the common tangents—and with the extra graphical information about the original plot, draw a rectangle for the region.

Finally, a new graphics object (`p3`) is created.

2: Our final visualization is obtained by showing all three graphics objects together.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Worked Example (part D): Visualizing the Spinodal and Common Tangent Construction

The previous three parts illustrate how one might actually go about developing a complex visualization: create simple working parts and then integrate them together into something more complex. (Don't get the impression that I didn't make any errors or silly conceptual mistakes as I created this example! It was very time consuming and, while it looks fairly straightforward in hindsight, it was a challenge to build.) However, once finished, it is useful to collect everything into a single function that can be reused.

```
Needs["ComputationalGeometry`"];
CommonTangentConstruction[
  Gm_, T_, npts_:100] :=
 Module[{x, y, p1, p2, p3, gxmin, gxmax,
   gymin, gymax, ddg, gvals, gmax,
   chull, conprts, len, glist = {}, i = 1,
   lftpt, rtpt, ctline, twophasreg},
  p1 = Plot[Gm[x, T], {x, 0, 1},
    PlotStyle → Thick];
  {{gxmin, gxmax}, {gymin, gymax}} =
   PlotRange /.
    AbsoluteOptions[p1, PlotRange];
  ddg = D[Gm[x, T], {x, 2}];
  p2 = RegionPlot[ddg < 0,
    {x, gxmin, gxmax}, {y, gymin, gymax},
    PlotStyle → RGBColor[0, 1, .5, 0.1]];
  gvals = Table[{x, Gm[x, T]},
    {x, 0, 1, 1 / N[npts - 1]}];
  gmax = Max[Transpose[gvals][[2]]];
  PrependTo[gvals, {0, 10 * Abs[gmax]}];
  AppendTo[gvals, {1, 10 * Abs[gmax]}];
  chull = Sort[ConvexHull[gvals]];
  chull = Drop[Drop[chull - 1, 1], -1];
  gvals = Drop[Drop[gvals, 1], -1];
  conprts = Split[chull, (#2 - #1 < 2) &];
  len = Length[conprts];
  While[i + 1 ≤ len,
   lftpt = gvals[[Last[ conprts[[i]] ]]];
   rtpt = gvals[[First[ conprts[[i + 1]] ]]];
   ctline =
    {Red, Thick, Line[{lftpt, rtpt}]};
   twophasreg = {RGBColor[0.5, 0, 0, 0.2],
     Rectangle[{lftpt[[1]], gymin},
      {rtpt[[1]], gymax}]};
   AppendTo[glist, ctline];
   AppendTo[glist, twophasreg]; i++];
  p3 = Graphics[glist]; Show[p1, p2, p3]]
```

1

1: Here is the result, *CommonTangentConstruction* , which collects the previous three examples together and returns a single graphical object. *CommonTangentConstruction* takes two arguments for the molar free-energy function, `Gm`, and temperature `T`, and an optional third argument for the precision to calculate the hull. The optional argument is indicated by the `:100` and will default to 100 if not passed to the function.

The first argument must be the name of a defined function of composition and temperature.

3.016 Home

Full Screen

Close

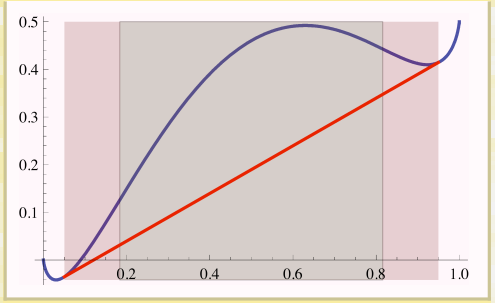
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

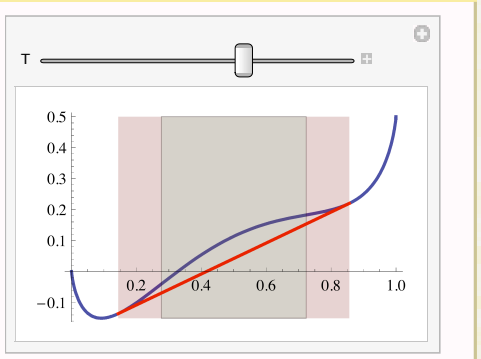
Worked Example (part E): Visualizing the Spinodal and Common Tangent Construction

Examples of visualizing with *CommonTangentConstruction* are presented here.

```
CommonTangentConstruction[Gmolar, 1.5]
```
1




```
Manipulate[CommonTangentConstruction[
  Gmolar, T, 300], {{T, 2}, 0, 3}]
```
2



1: This is the construction at $T = 1.5$.

2: Here we use the construction as an argument to `Manipulate` so that we can observe the effect of temperature on the spinodal and common tangent construction.

3.016 Home
Full Screen
Close
Quit

Lecture 6: Linear Algebra I

Reading:
Kreyszig Sections: 7.5, 7.6, 7.7, 7.8, 7.9 (pages302–305, 306–307, 308–314, 315–323, 323–329)

Vectors

Vectors as a list of associated information

$$\vec{x} = \begin{pmatrix} \text{number of steps to the east} \\ \text{number of steps to the north} \\ \text{number steps up vertical ladder} \end{pmatrix} \tag{6-1}$$

$$\vec{x} = \begin{pmatrix} 3 \\ 2.4 \\ 1.5 \end{pmatrix} \qquad \text{determines position} \qquad \begin{pmatrix} x_{\text{east}} \\ x_{\text{north}} \\ x_{\text{up}} \end{pmatrix} \tag{6-2}$$

The vector above is just one example of a position vector. We could also use coordinate systems that differ from the Cartesian (x, y, z) to represent the location. For example, the location in a *cylindrical coordinate system* could be written as

$$\vec{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} r\cos\theta \\ r\sin\theta \\ z \end{pmatrix} \tag{6-3}$$

i.e., as a *Cartesian vector* in terms of the cylindrical coordinates (r, θ, z).

The position could also be written as a cylindrical, or *polar* vector

$$\vec{x} = \begin{pmatrix} r \\ \theta \\ z \end{pmatrix} = \begin{pmatrix} \sqrt{x^2 + y^2} \\ \tan^{-1} \frac{y}{x} \\ z \end{pmatrix} \tag{6-4}$$

where the last term is the polar vector in terms of the Cartesian coordinates. Similar rules would apply for other coordinate systems like spherical, elliptic, etc.

However, vectors need not represent position at all, for example:

$$\vec{n} = \begin{pmatrix} \text{number of Hydrogen atoms} \\ \text{number of Helium atoms} \\ \text{number of Lithium atoms} \\ \vdots \\ \text{number of Plutonium atoms} \\ \vdots \end{pmatrix} \tag{6-5}$$

Scalar multiplication

$$\frac{1}{\text{N}_{\text{avag.}}} \vec{n} \equiv \begin{pmatrix} \frac{\text{number of H}}{\text{N}_{\text{avag.}}} \\ \frac{\text{number of He}}{\text{N}_{\text{avag.}}} \\ \frac{\text{number of Li}}{\text{N}_{\text{avag.}}} \\ \vdots \\ \frac{\text{number of Pu}}{\text{N}_{\text{avag.}}} \\ \vdots \end{pmatrix} = \begin{pmatrix} \text{moles of H} \\ \text{moles of He} \\ \text{moles of Li} \\ \vdots \\ \text{moles of Pu} \\ \vdots \end{pmatrix} = \vec{m} \tag{6-6}$$

Vector norms

$$\|\vec{x}\| \equiv (x_1^2 + x_2^2 + \ldots x_k^2)^{1/2} = \text{euclidean separation (also called l2-norm)} \tag{6-7}$$

$$\|\vec{n}\| \equiv n_{\mathrm{H}} + n_{\mathrm{He}} + \ldots n_{132?} = \text{total number of atoms (related to the Manhatten norm)} \tag{6-8}$$

Unit vectors

unit direction vector

$$\hat{x} = \frac{\vec{x}}{\|\vec{x}\|} \tag{6-9}$$

mole fraction composition

$$\hat{m} = \frac{\vec{m}}{\|\vec{m}\|} \tag{6-10}$$

Extra Information and Notes

Potentially interesting but currently unnecessary

If $\Re$ stands for the set of all real numbers (i.e., 0, -1.6, $\pi/2$, etc.), then we can use a shorthand to specify the position vector, $\vec{x} \in \Re^N$ (e.g., each of the N entries in the vector of length N must be a real number, or must be in the set of real numbers, $\|\vec{x}\| \in \Re$.)

For the unit (direction) vector: $\hat{x} = \{\vec{x} \in \Re^3 \mid \|\vec{x}\| = 1\}$ (i.e, the unit direction vector is the set of all position vectors such that their length is unity—or, the unit direction vector is the subset of all position vectors that lie on the unit sphere. $\vec{x}$ and $\hat{x}$ have the same number of entries, but compared to $\vec{x}$, the number of independent entries in $\hat{x}$ is smaller by one.

For the case of the composition vector, it is unphysical to have a negative number of atoms, therefore the mole fraction vector $\vec{n} \in (\Re^+)^{elements}$ ($\Re^+$ is the real non-negative numbers) and $\hat{m} \in (\Re^+)^{(elements-1)}$.

Matrices and Matrix Operations

Consider methane (CH_4), propane (C_3H_8), and butane (C_4H_{10}).

$$\underline{M_{HC}} = \begin{pmatrix} \begin{matrix}\text{H-column}\\ \frac{\text{number of H}}{\text{methane molecule}}\end{matrix} & \begin{matrix}\text{C-column}\\ \frac{\text{number of C}}{\text{methane molecule}}\end{matrix} \\ \frac{\text{number of H}}{\text{propane molecule}} & \frac{\text{number of C}}{\text{propane molecule}} \\ \frac{\text{number of H}}{\text{butane molecule}} & \frac{\text{number of C}}{\text{butane molecule}} \end{pmatrix} \begin{matrix}\text{methane row}\\ \text{propane row}\\ \text{butane row}\end{matrix} \tag{6-11}$$

$$\underline{M_{HC}} = \begin{pmatrix} 4 & 1 \\ 8 & 3 \\ 10 & 4 \end{pmatrix} = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \\ M_{31} & M_{32} \end{pmatrix} \tag{6-12}$$

Matrices as a linear transformation of a vector

$$\vec{N_{HC}} = (\text{number of methanes}, \text{number of propanes}, \text{number of butanes}) \tag{6-13}$$

$$= (N_{HC\,m}, N_{HC\,p}, N_{HC\,b}) \tag{6-14}$$

$$= (N_{HC\,1}, N_{HC\,2}, N_{HC\,3}) \tag{6-15}$$

(6-16)

$$\vec{N_{HC}}\underline{M_{HC}} \equiv \sum_{i=1}^{3} N_{HC\,i} M_{HC\,ij} = \vec{N} \tag{6-17}$$

The "summation" convention is often used, where a repeated index is summed over all its possible values:

$$\sum_{i=1}^{p} N_{HC\,i} M_{HC\,ij} \equiv N_{HC\,i} M_{HC\,ij} = N_j \tag{6-18}$$

For example, suppose

$$\vec{N_{HC}} = (1.2 \times 10^{12} \text{ molecules methane}, 2.3 \times 10^{13} \text{ molecules propane}, 3.4 \times 10^{14} \text{ molecules butane}) \tag{6-19}$$

$$\vec{N_{HC}}\underline{M_{HC}} =$$

$$(1.2 \times 10^{14} \text{ methanes}, 2.3 \times 10^{13} \text{ propanes}, 3.4 \times 10^{12} \text{ butanes}) \begin{pmatrix} \frac{4 \text{ atoms H}}{\text{methane}} & \frac{1 \text{ atoms C}}{\text{methane}} \\ \frac{8 \text{ atoms H}}{\text{propane}} & \frac{3 \text{ atoms C}}{\text{propane}} \\ \frac{10 \text{ atoms H}}{\text{butane}} & \frac{4 \text{ atoms C}}{\text{butane}} \end{pmatrix} \tag{6-20}$$

$$=(7.0 \times 10^{14} \text{ atoms H}, 2.0 \times 10^{14} \text{ atoms C})$$

Matrix transpose operations

Above, the lists (or vectors) of atoms were stored as rows, but often it is convenient to store them as columns. The operation to take a row to a column (and vice-versa) is called a "transpose".

$$\underline{M_{HC}}^T = \begin{array}{c} \begin{matrix} \text{methane-column} & \text{propane-column} & \text{butane-column} \end{matrix} \\ \begin{pmatrix} \frac{\text{number of H}}{\text{methane molecule}} & \frac{\text{number of H}}{\text{propane molecule}} & \frac{\text{number of H}}{\text{butane molecule}} \\ \frac{\text{number of C}}{\text{methane molecule}} & \frac{\text{number of C}}{\text{propane molecule}} & \frac{\text{number of C}}{\text{butane molecule}} \end{pmatrix} \end{array} \begin{matrix} \text{hydrogen row} \\ \text{carbon row} \end{matrix} \tag{6-21}$$

$$\vec{N_{HC}}^T = \begin{pmatrix} \text{number of methanes} \\ \text{number of propanes} \\ \text{number of butanes} \end{pmatrix} = \begin{pmatrix} N_{HC\,m} \\ N_{HC\,p} \\ N_{HC\,b} \end{pmatrix} \tag{6-22}$$

$$\underline{M_{HC}}^T \vec{N_{HC}}^T = \vec{N}^T \begin{pmatrix} 4 & 8 & 10 \\ 1 & 3 & 4 \end{pmatrix} \begin{pmatrix} \text{number of methanes} \\ \text{number of propanes} \\ \text{number of butanes} \end{pmatrix} = \begin{pmatrix} \text{number of H-atoms} \\ \text{number of C-atoms} \end{pmatrix} \tag{6-23}$$

The next example supposes that some process produces hydrocarbons and can be modeled with the pressure P and temperature T. Suppose (this is an artificial example) that the number of hydrocarbons produced in one millisecond can be related linearly to the pressure and temperature:

$$\begin{aligned} \text{number of methanes} &= \alpha P + \beta T \\ \text{number of propanes} &= \gamma P + \delta T \\ \text{number of butanes} &= \epsilon P + \phi T \end{aligned} \tag{6-24}$$

or

$$\vec{N_{HC}}^{T} = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \\ \epsilon & \phi \end{pmatrix} \begin{pmatrix} P \\ T \end{pmatrix} \tag{6-25}$$

Matrices

Here is an example operation that takes us from the processing vector $(P,T)^T$ to the number of hydrogens and carbons.

M_{HC} is our matrix that maps the three hydrocarbons (methane CH_4, propane C_3H_8, butane C_4H_{10}, to number of hydrogens and carbons

```
MHC = {
   {4, 1},
   {8, 3},
   {10, 4}
  }

MHC // MatrixForm
```

1

```
Transpose[MHC] // MatrixForm
```

2

PTmatrix is our matrix of kinetic data that gives rates of change of a particular atomic species (C or H) as a function of pressure and temperature (see lecture notes corresponding to this Mathematica notebook).

```
PTmatrix = {
     {α, β},
     {γ, δ},
     {ϵ, ϕ}
   };
PTmatrix // MatrixForm
```

3

```
MPT = MHC. PTmatrix
```

4

The matrix multiplication does not work because the sizes are inconsistent.

```
Clear[MPT]
```

5

```
MPT = Transpose[MHC]. PTmatrix;
MPT // MatrixForm
```

6

1: The matrix (Eq. 6-12) is entered as a list of sublists. The sub-lists are the rows of the matrix. The first elements of each row-sublist form the first column; the second elements are the second column and so on.

The `Length` of a matrix-object gives the number of rows, and the second member of the result of `Dimensions` gives the number of columns.

All sublists of a matrix must have the same dimensions.

It is good practice to enter a matrix and then display it separately using `MatrixForm`. Otherwise, there is a risk of defining a symbol as a `MatrixForm`-object *and not as a matrix which was probably the intent.*

2: The `Transpose` function exchanges the rows and columns. If `Dimensions[Mat]` returns `{r,c}`, then `Dimensions[Transpose[Mat]]` returns `{c,r}`.

3: `Dimensions[PTmatrix]` is `{3,2}`.

4: This command will generate an error.

Matrix multiplication in MATHEMATICA® is produced by the "dot" (.) operator—and not the "multiplication" (*) operator. For matrix multiplication, $\underline{A}.\underline{B}$, the number of columns of $\underline{A}$ must be equal to the number of rows of $\underline{B}$.

6: The `Transpose` "flips" a matrix by producing a new matrix which has the original's i^{th} row as the new matrix's i^{th} column (or, equivalently the j^{th} column as the new j^{th} row). In this example, a 3×2-matrix (`PTmatrix`) is being left-multiplied by a a 2×3-matrix.

The resulting matrix would map a vector with values P and T to a vector for the rate of production of C and H.

3.016 Home

Full Screen

Close

Quit

Matrix multiplication is defined by:

$$\underline{AB} = \sum_i A_{ki} B_{ij} \tag{6-26}$$

The indices of the matrix defined by the multiplication $\underline{AB} = \underline{C}$ are C_{kj}.

Matrix Inversion

Sometimes what we wish to know is: "What vector is it ($\vec{x}$), when transformed by some matrix ($\underline{A}$), that gives us a particular result ($\vec{b} = \underline{A}\vec{x}$)?"

$$\begin{aligned} \underline{A}\vec{x} &= \vec{b} \\ \underline{A}^{-1}\underline{A}\vec{x} &= \underline{A}^{-1}\vec{b} \\ \vec{x} &= \underline{A}^{-1}\vec{b} \end{aligned} \tag{6-27}$$

The inverse of a matrix is defined as: something, that when multiplied with the matrix, leaves a product that has no effect on any vector. This special product matrix is called the *identity matrix*.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Inverting Matrices

Our last example produced a linear operation that answered the question, "given a particular P and T, at what rate will C and H be produced?"

To answer the converse question, "If I want a particular rate of production for C and H, at what P and T should the process be carried out?"

To invert the question on linear processes, the matrix is inverted.

```
MPT = Transpose[M_HC]. PTmatrix;
MPTinverse = Factor[Inverse[MPT]];
MPTinverse // MatrixForm
```
1

$$\begin{pmatrix} -\frac{\beta+3\,\delta+4\,\phi}{2\,(2\,\beta\,\gamma-2\,\alpha\,\delta+3\,\beta\,\epsilon+\delta\,\epsilon-3\,\alpha\,\phi-\gamma\,\phi)} & \frac{2\,\beta+4\,\delta+5\,\phi}{2\,\beta\,\gamma-2\,\alpha\,\delta+3\,\beta\,\epsilon+\delta\,\epsilon-3\,\alpha\,\phi-\gamma\,\phi} \\ -\frac{\alpha+3\,\gamma+4\,\epsilon}{2\,(-2\,\beta\,\gamma+2\,\alpha\,\delta-3\,\beta\,\epsilon-\delta\,\epsilon+3\,\alpha\,\phi+\gamma\,\phi)} & \frac{2\,\alpha+4\,\gamma+5\,\epsilon}{-2\,\beta\,\gamma+2\,\alpha\,\delta-3\,\beta\,\epsilon-\delta\,\epsilon+3\,\alpha\,\phi+\gamma\,\phi} \end{pmatrix}$$

The denominators are related to the determinant---if the determinant vanishes, then the inverse matrix is not defined.

```
Det[MPT]
```
2

Checking to see if the the inverse multiplied by the original matrix is the identity matrix:

```
MPT.MPTinverse
```
3

It is not obvious unless simplified...

```
Simplify[MPT.MPTinverse] // MatrixForm
```
4

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

1: Inverting a matrix by hand is tedious and prone to error, `Inverse` does this in MATHEMATICA® . In this example, `Factor` is called on the result of `Inverse`. `Factor` is an example of a *threadable function*—it recursively operates on all members of any argument that is a list-object. Thus, each of the entries in the inverted matrix is factored individually.

2: The *determinant* of a matrix is fundamentally linked to the existence of its inverse. In this example, it is observed that if the `Det` of a matrix vanishes, then the entries of its inverse are undetermined.

3: The multiplication of a matrix by its inverse should produce the identity matrix (i.e., a matrix with 1 at each diagonal entry, and zero otherwise). That this multiplication gives the identity matrix is not obvious. Unless, . . .

4: `Simplify` is called on each of the entries.

3.016 Home

Full Screen

Close

Quit

$$\underline{A}\vec{x} = \vec{b}$$

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \tag{6-28}$$

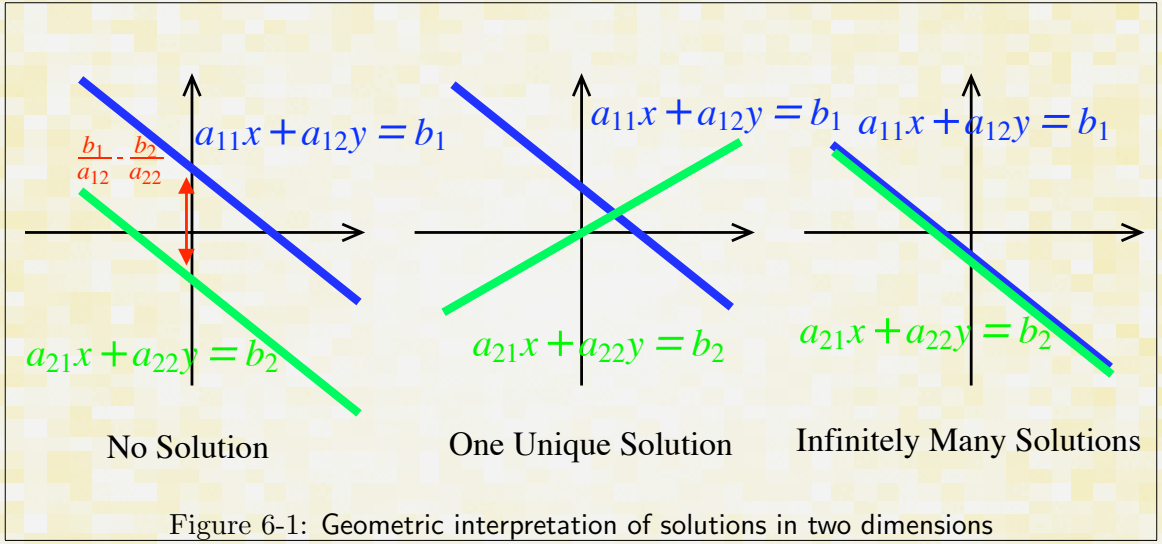


Figure 6-1: Geometric interpretation of solutions in two dimensions

Eliminating redundant equations or variables

Consider liquid water near the freezing point—dipole interactions will tend to make water molecules form clusters such as H_2O and H_4O_2. This example looks at such a case where the columns are not linearly independent.

Same example for water and water complexes: use the matrix watmat to store molecular formulas for each type of molecule in the system

```
watmat = {{2, 4}, {1, 2}};
watmat // MatrixForm
```
1

The vector molvec is used to store the number of each kind of molecule

```
molvec = {h20, h402}
```
2

The vector atomvec is used to store the number of each atomic species that is present

```
atomvec = {h, o}
```
3

```
atomvec // MatrixForm
```
4

The vector eq is now defined and its two elements are equations that give the number of hydrogen atoms and the number of oxygen atoms:

```
eq[1] = (watmat.molvec)[[1]] == atomvec[[1]]
```
5

```
eq[2] = (watmat.molvec)[[2]] == atomvec[[2]]
```
6

```
Solve[{eq[1], eq[2]}, molvec]
```
7

```
? Eliminate
```
8

```
Eliminate[eqns, vars] eliminates
    variables between a set of simultaneou
```

```
Eliminate[{eq[1], eq[2]}, molvec]
```
9

```
2 o == h
```

```
MatrixRank[watmat]
```
10

```
NullSpace[watmat]
Length[NullSpace[watmat]]
```
11

```
{{-2, 1}}
```

1: The mapping from molecules to the number of atoms becomes:

$$\begin{pmatrix} 2 & 4 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} N_{H_2O} \\ N_{H_4O_2} \end{pmatrix} = \begin{pmatrix} N_H \\ N_O \end{pmatrix} \tag{6-29}$$

The matrix `watmat` encodes the coefficients in these linear equations.

2–5: The vectors, `atomvec` and `molvec`, represent the numbers of each type of atom and each type of molecule.

5–6: These equations are the same as the rows of $\underline{A}\vec{x}$ being set to the corresponding entry of $\vec{b}$ for $\underline{A}\vec{x} = \vec{b}$. These are the linear equations given above.

7: This is an attempt (using `Solve` on the linear equations) to find the number of H_2O- and H_4O_2-molecules, given the number of H- and O-atoms. Of course, it has to fail.

8–9: `Eliminate` produces a logical equality for each redundancy in a set of equations. In this case, the result expresses the fact that 2 × (second row) is the same as the (first row).

10: The rank of a matrix, obtained with `MatrixRank`, gives the number of linearly independent rows.

11: The null space of a matrix, $\underline{A}$, is a linearly independent set of vectors $\vec{x}$, such that $\underline{A}\vec{x}$ is the zero-vector; this list can be obtained with `NullSpace`. The result is equivalent to that obtained with `Eliminate` in item 9. The *nullity* is the number of vectors in a matrix's null space. The rank and the nullity must add up to the number of columns of $\underline{A}$

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Lecture 7: Linear Algebra

Reading:
Kreyszig Sections: 13.1, 13.2, 13.5, 13.6 (pages602–606, 607–611, 623–626, 626–629)

Uniqueness and Existence of Linear System Solutions

It would be useful to use the Mathematica Help Browser and open the link to Matrices & Linear Algebra in the Mathematics & Algorithms section. Look through the tutorials at the bottom on the linked page.

A linear system of m equations in n variables $(x_1, x_2, \ldots, x_n)$ takes the form

$$\begin{aligned} A_{11}x_1 + A_{12}x_2 + A_{13}x_3 + \ldots + A_{1n}x_n &= b_1 \\ A_{21}x_1 + A_{22}x_2 + A_{23}x_3 + \ldots + A_{2n}x_n &= b_2 \\ \vdots &= \vdots \\ A_{k1}x_1 + A_{k2}x_2 + A_{k3}x_3 + \ldots + A_{kn}x_n &= b_k \\ \vdots &= \vdots \\ A_{m1}x_1 + A_{m2}x_2 + A_{m3}x_3 + \ldots + A_{mn}x_n &= b_m \end{aligned} \tag{7-1}$$

and is fundamental to models of many systems.

The coefficients, A_{ij}, form a matrix and such equations are often written in an "index" short-hand known as the Einstein summation convention:

$$A_{ji}x_i = b_j \quad \text{(Einstein summation convention)} \tag{7-2}$$

where *if an index* (here i) *is repeated in any set of multiplied terms,* (here $A_{ji}x_i$) *then a summation over all values of that index is implied.* Note that, by multiplying and summing in Eq. 7-2, the repeated index i disappears from the right-hand-side. It can be said that the repeated index in "contracted" out of the equation and this idea is emphasized by writing Eq. 7-2 as

$$x_i A_{ji} = b_j \qquad \text{(Einstein summation convention)} \tag{7-3}$$

so that the "touching" index, i, is contracted out leaving a matching j-index on each side. In each case, Eqs. 7-2 and 7-3 represent m equations ($j = 1, 2, \ldots, m$) in the n variables ($i = 1, 2, \ldots, n$) that are contracted out in *each equation.* The summation convention is especially useful when the dimensions of the coefficient matrix is larger than two; for a linear elastic material, the elastic energy density can be written as:

$$E_{\text{elast}} = \frac{1}{2}\epsilon_{ij}C_{ijkl}\epsilon_{kl} = \frac{1}{2}\sigma_{pq}S_{pqrs}\sigma_{rs} \tag{7-4}$$

where C_{ijkl} and ϵ_{ij} are the fourth-rank *stiffness* tensor and second-rank elastic *strain* tensor; where S_{ijkl} and σ_{ij} are the fourth-rank compliance tensor and second-rank stress tensor;

In physical problems, the goal is typically to find the n x_i for a given m b_j in Eqs. 7-2, 7-3, or 7-1. This goal is conveniently represented in matrix-vector notation:

$$\underline{A}\vec{x} = \vec{b} \tag{7-5}$$

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Solving Linear Sets of Equations

Demonstrations of several different ways to solve a set of linear equations for several variables. Here we will solve equations that we construct from matrices; in following examples, we will operate on the matrices directly.

Consider the set of equations

$$x + 2y + z + t = a$$
$$-x + 4y - 2z = b$$
$$x + 3y + 4z + 5t = c$$
$$x + z + t = d$$

We illustrate how to use a matrix representation to write these out and solve them...

Start with the matrix of coefficients of the variables, ***mymatrix****:*

```
mymatrix = {
   {1, 2, 1, 1},
   {-1, 4, -2, 0},
   {1, 3, 4, 5},
   {1, 0, 1, 1}};
mymatrix // MatrixForm
```
1

The system of equations will only have a unique solution if the determinant of ***mymatrix*** *is nonzero.*

```
Det[mymatrix]
```
2

Now define vectors for $\vec{x}$ *and* $\vec{b}$ *in* $\underline{A}\vec{x} = \vec{b}$

```
myx = {x, y, z, t};
```
3

```
myb = {a, b, c, d};
```
4

The left-hand side of the first equation will be

```
(mymatrix.myx)[[1]]
```
5

and the left-hand side of all four equations will be

```
lhs = mymatrix.myx;
lhs // MatrixForm
```
6

Now define an indexed variable ***linsys*** *with four entries, each being one of the equations in the system of interest:*

```
linsys[i_Integer] := lhs[[i]] == myb[[i]]
```
7

```
linsys[2]
```
8

Solving the set of equations for the unknowns $\vec{x}$

```
linsol = Solve[{linsys[1],
   linsys[2], linsys[3], linsys[4]}, myx]
```
9

1: This example is kind of backwards. We will take a matrix

$$\underline{A} = \begin{pmatrix} 1 & 2 & 1 & 1 \\ -1 & 4 & -2 & 0 \\ 1 & 2 & 4 & 5 \\ 1 & 0 & 1 & 1 \end{pmatrix} \text{ unknown vector } \vec{x} = \begin{pmatrix} x \\ y \\ z \\ t \end{pmatrix} \text{ and known vector } \vec{b} = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}$$

and extract four equations for input to `Solve` to obtain the solution to $\vec{x}$. Here, the coefficient matrix is a list of row-lists.

2: A unique solution will exist if the determinant, computed with `Det`, is non-zero.

3–4: These will be the left-hand- and right-hand-side vectors.

5: Matrix multiplication is indicated by the period (.). This will be the first of the equations.

6: `lhs` is a column-vector with four entries, and each entry is one of the lhs equations.

7–8: This function creates *logical equalities* for each corresponding entry on the left- and right-hand-sides. unknowns.

9: The function `Solve` is used on a system of equations ({`linsys[i]`} and variables.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Inverting Matrices or Just Solving for the Unknown Vector

Continuing the last example, it is much more compact to invert a matrix symbolically or numerically. If a matrix inverse is going to be used over and over again, it is probably best to compute and store the inverse once. However, if a one-time only solution for $\vec{x}$ in $\underline{A}\vec{x} = \vec{b}$ is needed, then computing the inverse is computationally less efficient than using an algorithm designed to solve for $\vec{x}$ directly. Here is an example of both methods.

Doing the same thing a different way, using Mathematica's ***LinearSolve*** *function:*

```
? LinearSolve
```

1

```
LinearSolve[m, b] finds an x which solves th
LinearSolve[m] generates a LinearSolveFun
    which can be applied repeatedly to diff
```

```
LinearSolve[mymatrix, myb]
```

2

And yet another way, based on $\vec{x} = \underline{A}^{-1}\,\underline{A}\,\vec{x} = \underline{A}^{-1}\,\vec{b}$

```
Inverse[mymatrix].myb // MatrixForm
```

3

$$\begin{pmatrix} \frac{a}{7} + \frac{b}{7} - \frac{2c}{7} + \frac{9d}{7} \\ \frac{a}{2} - \frac{d}{2} \\ \frac{13a}{14} - \frac{4b}{7} + \frac{c}{7} - \frac{23d}{14} \\ -\frac{15a}{14} + \frac{3b}{7} + \frac{c}{7} + \frac{19d}{14} \end{pmatrix}$$

And yet even another way, a very efficient LinearSolveFunction can be produced by LinearSolve. This function will operate on any rhs vector of the appropriate length. This would be an efficient way to find the numerical solution to a known matrix, but for many different rhs $\vec{b}$*.*

```
mymatrixsol = LinearSolve[mymatrix];
```

4

The result can be applied as a function calling a vector :

```
mymatrixsol[myb]
Simplify[mymatrixsol[myb]]
```

5

$$\left\{\frac{1}{7}(a + b - 2c + 9d),\ \frac{a - d}{2},\ \frac{1}{14}(13a - 8b + 2c - 23d),\ \frac{1}{14}(-15a + 6b + 2c + 19d)\right\}$$

1–2: `LinearSolve` can take two arguments, $\underline{A}$ and $\vec{b}$, and returns $\vec{x}$ that solves $\underline{A}\vec{x} = \vec{b}$. It will be noticibly faster than the following inversion method, especially for large matrices.

3: The matrix inverse is obtained with `Inverse` and a subsequent multiplication by the right-hand-side gives the solution.

4–5: Calling `LinearSolve` on a matrix alone, *returns an efficient function* that takes the unknown vector as an argument. Here we show the equivalence to item 3.

3.016

3.016 Home

Full Screen

Close

Quit

Uniqueness of solutions to the nonhomogeneous system

$$\underline{A}\vec{x} = \vec{b} \tag{7-6}$$

Uniqueness of solutions to the homogeneous system

$$\underline{A}\vec{x_o} = \vec{0} \tag{7-7}$$

Adding solutions from the nonhomogeneous and homogenous systems

You can add any solution to the homogeneous equation (if they exist, there are infinitely many of them) to any solution to the nonhomogeneous equation, and the result is still a solution to the nonhomogeneous equation.

$$\underline{A}(\vec{x} + \vec{x_o}) = \vec{b} \tag{7-8}$$

Determinants

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Determinants, Rank, and Nullity

Several examples of determinant calculations are provided to illustrate the properties of determinants. When a determinant vanishes (i.e., $\det \underline{A} = 0$), there is no solution to the inhomogeneous equation $\det \underline{A} = \vec{b}$, but there will be an infinity of solutions to $\det \underline{A} = 0$; the infinity of solutions can be characterized by solving for a number *rank* of the entries of $\vec{x}$ in terms of the *nullity* of other entries of $\vec{x}$

Create a matrix with one row as a linear combination of the others

```
myzeromatrix =
  {mymatrix[[1]], mymatrix[[2]],
   p * mymatrix[[1]] +
    q * mymatrix[[2]] + r * mymatrix[[4]],
   mymatrix[[4]]};
myzeromatrix // MatrixForm
```

1

$$\begin{pmatrix} 1 & 2 & 1 & 1 \\ -1 & 4 & -2 & 0 \\ p-q+r & 2p+4q & p-2q+r & p+r \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

```
Det[myzeromatrix]
```

2

```
LinearSolve[myzeromatrix, myb]
```

3

This was not expected to have a solution

```
MatrixRank[mymatrix]
MatrixRank[myzeromatrix]
```

4

```
NullSpace[mymatrix]
NullSpace[myzeromatrix]
```

5

*Try solving this inhomogeneous system of equations using **Solve**:*

```
zerolhs = myzeromatrix.myx
```

6

```
zerolinsys[i_Integer] :=
 zerolhs[[i]] == myb[[i]]
```

7

```
zerolinsolhet =
 Solve[Table[zerolinsys[i], {i, 4}], myx]
```

8

No solution, as expected, Let's solve the homogeneous problem:

```
zerolinsolhom = Solve[Table[zerolinsys[i] /.
    {a → 0, b → 0, c → 0, d → 0}, {i, 4}], myx]
```

9

```
{{y → 0, x → -2 t, z → t}}
```

1: A matrix is created where the third row is the sum of $p \times$ first row, $q \times$ second row, and $r \times$ fourth row. In other words, one row is a linear combination of the others.

2: The determinant is computed with `Det`, and its value should reflect that the rows are not linearly independent.

3: An attempt to solve the linear inhomogeneous equation (here, using `LinearSolve`) should fail.

4: When the determinant is zero, there may still be some linearly-independent rows or columns. The rank gives the number of linearly-independent rows or columns and is computed with `MatrixRank`. Here, we compare the rank of the original matrix and the linearly-dependent one we created.

5: The *null space* of a matrix, $\underline{A}$, is a set of linearly-independent vectors that, if left-multiplied by $\underline{A}$, gives a zero vector. The nullity is how many linearly-independent vectors there are in the null space. Sometimes, vectors in the null space are called *killing vectors*. By comparing to the above, you will see examples of the *rank + nullity = dimension* rule for square matrices.

6–8: Here, an attempt to use `Solve` for the heterogeneous system with vanishing determinant is attempted, but of course it is bound to fail...

9: However, this is the solution to the singular homogeneous problem ($\underline{A}\vec{x} = \vec{0}$, where $\det \underline{A} = 0$. The solution is three (the rank) dimensional surface embedded in four dimensions (the rank plus the nullity). Notice that the solution is a multiple of the null space that we computed in item 5.

3.016 Home

Full Screen

Close

Quit

In example 07-1, it was stated (item 2) that a unique solution exists if the matrix's determinant was non-zero. The solution,

$$\vec{x} = \begin{pmatrix} \frac{2a+2b-4c+18d}{\det \underline{A}} \\ \frac{7a-7d}{\det \underline{A}} \\ \frac{13a-8b+2c-23d}{\det \underline{A}} \\ \frac{-15a+6b+2c+19d}{\det \underline{A}} \end{pmatrix} \tag{7-9}$$

indicates why this is the case and also illustrates the role that the determinant plays in the solution. Clearly, if the determinant vanishes, then the solution is undetermined unless $\vec{b}$ is a zero-vector $\vec{0} = (0, 0, 0, 0)$. Considering the *algebraic equation*, $ax = b$, the determinant plays the same role for matrices that the condition $a = 0$ plays for algebra: the inverse exists when $a \neq 0$ or $\det \underline{A} \neq 0$.

The determinant is only defined for square matrices; it derives from the elimination of the n unknown entries in $\vec{x}$ using all n equation (or rows) of

$$\underline{A}\vec{x} = 0 \tag{7-10}$$

For example, eliminating x and y from

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \text{gives the expression}$$
$$\det \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \equiv a_{11}a_{22} - a_{12}a_{21} = 0 \tag{7-11}$$

and eliminating x, y, and z from

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \tag{7-12}$$

gives the expression

$$\det \underline{A} \equiv a_{11}a_{22}a_{33} - a_{11}a_{32}a_{23} + a_{21}a_{32}a_{13} - a_{21}a12a_{33} + a_{31}a_{12}a_{23} - a_{31}a_{22}a_{13} = 0$$

The following general and true statements about determinants are plausible given the above expressions:

- Each term in the determinant's sum us products of N terms—a term comes from each column.
- Each term is one of all possible the products of an entry from each column.
- There is a plus or minus in front each term in the sum, $(-1)^p$, where p is the number of *neighbor exchanges required to put the rows in order* in each term written as an ordered product of their columns (as in Eqs. 7-11 and 7-12).

These, and the observation that it is impossible to eliminate $\vec{x}$ in Eqs. 7-11 and 7-12 if the information in the rows is redundant (i.e., there is not enough information—or independent equations—to solve for the $\vec{x}$), yield the general properties of determinants that are illustrated in the following example.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Properties of Determinants and Numerical Approximations to Zero

Rules, corresponding to how $\det A$ changes when the columns of $\underline{A}$ are permuted or multiplied by a constant, are demonstrated.

```
rv[i_] :=
 rv[i] = Table[RandomReal[{-1, 1}], {j, 6}]
```
1

Now use rv to make a 6 x 6 matrix, then find its determinant:

```
RandMat = Table[rv[i], {i, 6}]
```
2

```
Det[RandMat]
```
3

Switching two rows changes the sign but not the magnitude of the determinant:

```
Det[{rv[2], rv[1], rv[3], rv[4], rv[5], rv[6]}]
```
4

Multiply one row by a constant and calculate determinant:

```
Det[{a * rv[2], rv[1],
  rv[3], rv[4], rv[5], rv[6]}]
```
5

Multiply two rows by a constant and calculate determinant:

```
Det[{a * rv[2], a * rv[1],
  rv[3], rv[4], rv[5], rv[6]}]
```
6

Multiply all rows by a constant and calculate determinant:

```
Det[
 a {rv[2], rv[1], rv[3], rv[4], rv[5], rv[6]}]
```
7

```
Clear[a, b, c, d, e]
LinDepVec = a * rv[1] + b * rv[2] +
  c * rv[3] + d * rv[4] + e * rv[5]
```
8

Example of numerical precision: this determinant should evaluate to zero...

```
Det[{rv[1], rv[2],
  rv[3], rv[4], rv[5], LinDepVec}]
```
9

$$-4.85723 \times 10^{-17}\, a + 4.85723 \times 10^{-17}\, b + 4.16334 \times 10^{-17}\, c - 4.85723 \times 10^{-17}\, d - 1.38778 \times 10^{-17}\, e$$

However, numerical precision does

```
Chop[Det[{rv[1], rv[2],
   rv[3], rv[4], rv[5], LinDepVec}]]
```
10

1–2: A matrix, *RandMat* , is created from rows with random real entries between -1 and 1.

3–4: This will demonstrate that switching neighboring rows of a matrix changes the sign of the determinant.

5–6: Multiplying *one* column of a matrix by a constant a, multiplies the matrix's determinant by *one factor of* a; multiplying two rows by a gives a factor of a^2. Multiplying every entry in the matrix by a changes its determinant by a^n.

7: Because the matrix has one linearly-dependent column, its determinant should vanish. This example demonstrates what happens with limited numerical precision operations on real numbers. The determinant is not zero, but could be considered *effectively zero*.

8: We create a row which is an arbitrary linear combination of the first five rows of `RandMat`.

9: This determinant should be zero. However, because the entries are numerical, differences which are smaller than the precision with which a number is stored, may make it difficult to distinguish between something that is *numerically zero* and one that is *precisely zero*. This is sometimes known as *round-off error*.

10: Problems with numerical imprecision can usually be alleviated with `Chop` which sets small magnitude numbers to zero.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Determinants and the Order of Matrix Multiplication

Symbolic matrices are constructed to show examples of the rules $\det(\underline{AB}) = \det \underline{A} \det \underline{B}$ and $\underline{AB} \neq \underline{BA}$.

Creating a symbolic matrix

```
SymVec = {a, a, a, c, c, c};                                          1
```

```
Permuts = Permutations[SymVec]                                        2
Permuts // Dimensions
```

```
SymbMat = {                                                           3
      Permuts[[1]],
      Permuts[[12]],
      Permuts[[6]],
      Permuts[[18]],
      Permuts[[17]],
      Permuts[[9]]};
SymbMat // MatrixForm
```

```
DetSymbMat = Simplify[Det[SymbMat]]                                   4
```

Creating a matrix of random rational numbers

```
RandomMat =                                                           5
  Table[Table[RandomInteger[{-100, 100}] / RandomInteger[{1, 100}],
    {i, 6}], {j, 6}];
MatrixForm[RandomMat]
```

```
DetRandomMat = Det[RandomMat]                                         6
```

```
CheckA = Det[SymbMat.RandomMat] // Simplify                           7
```

```
DetRandomMat * DetSymbMat == CheckA                                   8
```

Does the determinant of a product depend on the order of multiplication?

```
CheckB = Det[RandomMat.SymbMat] // Simplify                           9
```

```
CheckA == CheckB                                                      10
```

However, the product of two matrices depends on which matrix is on the left and which is on the right

```
(RandomMat.SymbMat - SymbMat.RandomMat) //                            11
  Simplify // MatrixForm
```

1–3: Using `Permutations` to create all possible permutations of two sets of three identical objects for subsequent construction of a symbolic matrix, `SymbMat`, for demonstration purposes.

4: The symbolic matrix has a fairly simple determinant—it can only depend on two symbols and must be sixth-order.

5: A matrix with random rational numbers is created...

6: And, of course, its determinant is also a rational number.

7–10: This demonstrates that the determinant of a product is the product of determinants and is independent of the order of multiplication...

11: However, the result of multiplying two matrices *does* depend on the order of multiplication: $\underline{AB} \neq \underline{BA}$, in general.

Matrix multiplication is *non-commutative*: $\underline{AB} \neq \underline{BA}$ for most matrices. However, any two matrices for which the order of multiplication does not matter ($\underline{AB} = \underline{BA}$) are said to *commute*. Commutation is an important concept in quantum mechanics and crystallography.

Think about what commuting matrices means physically. If two linear transformations commute, then the order in which they are applied doesn't matter. In quantum mechanics, an operation is roughly equivalent to making an observation—commuting operators means that one measurement does not interfere with a commuting measurement. In crystallography, operations are associated with symmetry operations—if two symmetry operations commute, they are, in a sense, "orthogonal operations."

3.016 Home
Full Screen
Close
Quit

Vector Spaces

Consider the position vector

$$\vec{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \tag{7-13}$$

The vectors $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$ can be used to generate any general position by suitable scalar multiplication and vector addition:

$$\vec{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = x \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + y \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + z \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{7-14}$$

Thus, three dimensional real space is "spanned" by the three vectors: $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$. These three vectors are candidates as "basis vectors for $\Re^3$."

Consider the vectors $(a,-a,0)$, $(a,a,0)$, and $(0,a,a)$ for real $a \neq 0$.

$$\vec{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \frac{x+y}{2a} \begin{pmatrix} a \\ -a \\ 0 \end{pmatrix} + \frac{x-y}{2a} \begin{pmatrix} a \\ a \\ 0 \end{pmatrix} + \frac{x-y+2z}{2a} \begin{pmatrix} 0 \\ a \\ a \end{pmatrix} \tag{7-15}$$

So $(a,-a,0)$, $(a,a,0)$, and $(0,a,a)$ for real $a \neq 0$ also are basis vectors and can be used to span $\Re^3$.

The idea of basis vectors and vector spaces comes up frequently in the mathematics of materials science. They can represent abstract concepts as well as being shown by the following two dimensional basis set:

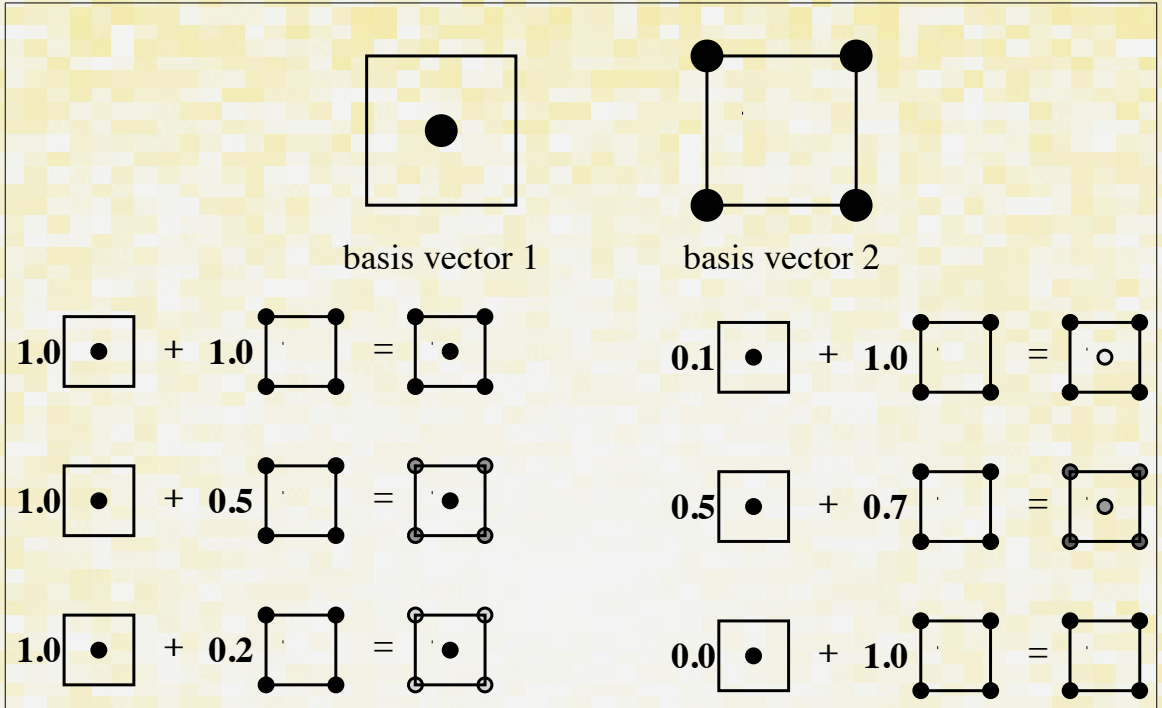


Figure 7-2: A vector space for two-dimensional CsCl structures. Any combination of center-site concentration and corner-site concentration can be represented by the sum of two basis vectors (or basis lattice). The set of all grey-grey patterns is a vector space of patterns.

Linear Transformations

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualization Example: Polyhedra

A simple octagon with different colored faces is transformed by operating on all of its vertices with a matrix. This example demonstrates how symmetry operations, like rotations reflections, can be represented as a matrix multiplication, and how to visualize the results of linear transformations generally.

We now demonstrate the use of matrix multiplication for manipulating an object, specifically an octohedron. The Octahedron is made up of eight polygons and the initial coordinates of the vertices were set to make a regular octahedron with its main diagonals parallel to axes x,y,z. The faces of the octahedron are colored so that rotations and other transformations can be easily tracked.

```
<< "PolyhedronOperations`"
Show[PolyhedronData["Octahedron"]]
```
1

Above, the color of the three dimensional object derives from the colors in the light sources. For example, note that there appears to be a blue light pointing down from the left. The lights stay fixed as we rotate the object. If Lighting → None, then the polyhedron's faces will appear to be black.

```
Show[PolyhedronData["Octahedron"],
 Lighting → None]
```
2

We can extract data from the Octahedron, and build our own with individually colored faces. We will need the individual colors to identify what happens to the polyhedron under linear transformaions.

```
PolyhedronData["Octahedron", "Faces"]
```
3

*The object **ColOct** is defined below to draw an octahedron and it invokes the **Polygon** function to draw the triangular faces by connecting three points at specific numerical coordinates that we obtain from the Octahedron data. Because we will turn off lighting, we will ask that each of the faces glow, using the **Glow** graphics directive*

```
octa = {p[1], p[2], p[3], p[4], p[5], p[6]} =
   PolyhedronData[
     "Octahedron", "Faces"][[1]];
colface[i_] := Glow[Hue[i / 8]];
ColOct =
  {{colface[0], Polygon[{p[4], p[5], p[6]}]},
   {colface[1], Polygon[{p[4], p[6], p[2]}]},
   {colface[2], Polygon[{p[4], p[2], p[1]}]},
   {colface[3], Polygon[{p[4], p[1], p[5]}]},
   {colface[4], Polygon[{p[5], p[1], p[3]}]},
   {colface[5], Polygon[{p[5], p[3], p[6]}]},
   {colface[6], Polygon[{p[3], p[1], p[2]}]},
   {colface[7], Polygon[{p[6], p[3], p[2]}]}};
```
4

```
Show[Graphics3D[ColOct], Lighting → None]
```
5

1: The package `PolyhedronOperations` contains `Graphics Objects` and other information such as vertex coordinates of many common polyhedra. This demonstrates how an `Octahedron` can be drawn on the screen. The color of the faces comes from the light sources. For example, there is a blue source behind your left shoulder; as you rotate the object the faces—oriented so that they reflect light from the blue source—will appear blue-ish. The color model and appearance is an advanced topic.

2: Setting `Lighting→None` removes the light sources and the octahedron will appear black. Our objective is to observe the effect of linear transformation on this object. To do this, will will want to identify each of the octahedron's faces by "painting" it.

3: We will build a custom octahedron from the Mm's version using `PolyhedronData`.

4: The data is extracted by grabbing the first part of `PolyhedronData` (i.e., `[[1]]`). We assign the name of the list *octa* , and name its elements `p[i]` in one step.
A function is defined and will be used to call `Glow` and `Hue` for each face. When the face glows and the lighting is off, the face will appear as the "glow color", independent of its orientation.
ColOct is a list of graphics-primitive lists: each element of the list uses the glow directive and then uses the points of the original octahedron to define `Polygons` in three dimensions.

5: We wrap *ColOct* inside `Graphics3D` and use `Show` with lighting off to visualize.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Linear Transformations: Matrix Operations on Polyhedra

A moderately sophisticated MATHEMATICA® function is defined to help visualize the effect of operating on each point of a polyhedron with a 3×3-matrix representing a symmetry operation.

```
transoct[tmat_, description_String] :=
 {ColOct /.
   {Polygon[{a_List, b_List, c_List}] →
     Polygon[{tmat.a, tmat.b, tmat.c}]},
  Text[Style[MatrixForm[tmat]], {0, 0, -.25}],
  Text[Style[description, Darker[Red]],
   {0, 0, .25}, Background → White]}
```

1

```
Show[Graphics3D[
  transoct[{{1, 0, 0}, {0, 1, 0}, {0, 0, -1}},
   "mirror-[001]"]], Lighting → None]
```

2

```
identity = IdentityMatrix[3];
rot90[001] = {{0, -1, 0}, {1, 0, 0}, {0, 0, 1}};
 ref[010] = {{1, 0, 0}, {0, -1, 0}, {0, 0, 1}};
o[1, 1] = transoct[identity, "original"];
o[1, 2] = transoct[rot90[001], "90-[001]"];
o[1, 3] = transoct[ref[010], "m-[010]"];
o[2, 1] = transoct[ref[010].rot90[001],
   "90-[100] then m-[010]"];
o[2, 2] = transoct[rot90[001].ref[010],
   "m-[010] then 90-[100]"];
```

3

```
RotationTransform[Pi, {1, 1, 0}]
```

4

```
o[2, 3] = transoct[{{0, 1, 0}, {1, 0, 0}, {0, 0, -1}}, "180-[110]"];
```

5

```
sc[θ_, ϕ_] :=
 3 {Cos[θ] Sin[ϕ], Sin[θ] Sin[ϕ], Cos[ϕ]}
Manipulate[GraphicsGrid[
  Table[Show[Graphics3D[o[i, j]],
    Lighting → None, ViewPoint → sc[θ, ϕ],
    ImageSize → {200, 200},
    PlotRange → {{-1, 1}, {-1, 1}, {-1, 1}}],
   {j, 3}, {i, 2}]], {{θ, 2.1}, 0, 2 π},
 {{ϕ, -1.4}, -π / 2, π / 2}]
```

6

1: This is a moderately sophisticated example of rule usage inside of a function (*transoct*) definition: the pattern matches triangles (`Polygons` with three points) in a graphics primitive; names the points; and then multiplies a matrix by each of the points. The first argument to *transoct* is the matrix which will operate on the points; the second argument is an identifyer that will be used with `Text` to annonate the graphics.

2: This demonstrates the use of *transoct* : we get a rotate-able 3D object with floating text identifying the name of the operation and the matrix that performs the operation.

3: We will build an example that will visualize several symmetry steps simultaneously (say that fast outloud). We define matrices for *identity* , *rot90[001]* , and *ref[010]* , respectively, which leave the polyhedra's points unchanged, rotate counter-clockwise by 90° around the [001]-axis, and reflect through the origin in the direction of the [010]-axis.

We use these matrices to create new octahedra corresponding to combinations of symmetry operations.

4–5: It is not always straightforward to write down the matrix corresponding to an arbitrary symmetry operation. MATHEMATICA® has functions to help find many of them; here, we use `RotationTransform` to find the matrix corresponding to rotation by 180° around the [110]-axis.

6: This will display six of the octahedra with their annotated symmetry operations. `Manipulate` is used to change the viewpoint to someplace on a sphere of radius 3 (by changing the latitude angle, ϕ, and the longitude θ). A function to return a cartesian representation of the spherical coordinates is defined first and is used as the `ViewPoint` for each `Graphics3D`-object. `Table` iterates over the `o[i,j]` and passes its result to `GraphicsGrid`.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualization Example: Invariant Symmetry Operations on Crystals

Each crystal's unit cell can be uniquely characterized by the symmetry operations (i.e., fixed rotation about an axis, reflection across a plane, and inversion through the origin) which leave the unit cell unchanged. The set of such symmetry operations define the *crystal point group*. There are only 32 point groups in three dimensions. In this example, we demonstrate invariant operations for an FCC cell.

```
corners = Flatten[Table[{i, j, k},
   {i, 0, 1}, {j, 0, 1}, {k, 0, 1}], 2]
faces = Join[Permutations[{0.5, 0.5, 0}],
  Permutations[{0.5, 0.5, 1}]]
fccsites = Join[faces, corners]
srad = √2 / 4;
FCC = Table[
  Sphere[fccsites[[i]], srad], {i, 1, 14}]
axes = { {RGBColor[1, 0, 0, .5],
    Cylinder[{{0, 0, 0}, {2, 0, 0}}, .05]},
   {RGBColor[0, 1, 0, .5],
    Cylinder[{{0, 0, 0}, {0, 2, 0}}, .05]},
   {RGBColor[0, 0, 1, .5],
    Cylinder[{{0, 0, 0}, {0, 0, 2}}, .05]}};
fccmodel = Translate[Join[FCC, axes],
  {-.5, -.5, -.5}]
Graphics3D[fccmodel]
```

1

```
bbox = 1.25 {{-1, 1}, {-1, 1}, {-1, 1}};
Manipulate[Grid[{{"original",
    "2π/3-[111]", "roto-inversion: 3̄"},
   {Graphics3D[fccmodel, PlotRange → bbox,
     ViewPoint → sc[θ, ϕ]],
    Graphics3D[Rotate[fccmodel, 2 π / 3,
      {1, 1, 1}], PlotRange → bbox,
     ViewPoint → sc[θ, ϕ]], Graphics3D[
     Rotate[GeometricTransformation[
       fccmodel, -IdentityMatrix[3]],
      2 π / 3, {1, 1, 1}], PlotRange → bbox,
     ViewPoint → sc[θ, ϕ]]}}],
 {{θ, 2.2}, 0, 2 π}, {{ϕ, -.6},
  -π / 2, π / 2}]
```

2

1: The first two commands define *faces* and *corners* which are the coordinates of the face-centered and corner lattice-sites. Note the use of `Flatten` in *corners* has the qualifier 2—it limits the scope of `Flatten` which would normally turn a list of lists into a (flat) single list. `Join` is used to collect the two coordinate lists together into *fccsites* . The atoms will be visualized with the `Sphere` graphics primitive and we use *srad* to set the radius of a close-packed FCC structure. *FCC* is a list of (a list of) graphics primitives for each of the fourteen spheres, and then three cylinders with `Opacity` and color are used to define the coordinate axes graphics: *axes* .
fccmodel is created by joining the spheres and the cylinders, and then using `Translate` on the resulting graphics primitives to put the center of the FCC cell at the origin.

2: `Translate` is an example of a function that operates directly on graphics primitives. We use related functions that also operate on graphics primitives, `Rotate` and `GeometricTransformation`, to illustrate how rotation by 120° about [111], and how inversion (multiplication by "minus the identity matrix") followed by the same rotation, are invariant symmetry operations for the FCC lattice.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Lecture 8: Complex Numbers and Euler's Formula

Reading:
Kreyszig Sections: 8.1, 8.2, 8.3 (pages334–338, 340–343, 345–348)

Complex Numbers and Operations in the Complex Plane

Consider, the number zero: it could be *operationally* defined as the number, which when multiplied by any other number always yields itself; and its other properities would follow.

Negative numbers could be defined operationally as something that gives rise to simple patterns. Multiplying by -1 gives rise to the pattern $1, -1, 1, -1, \ldots$ In the same vein, a number, $\imath$, can be created that doubles the period of the previous example: multiplying by $\imath$ gives the pattern: $1, \imath, -1, -\imath, 1, \imath, -1, -\imath, \ldots$ Combining the imaginary number, $\imath$, with the real numbers, arbitrarily long periods can be defined by multiplication; applications to periodic phenonena is probably where complex numbers have their greatest utility in science and engineering

With $\imath \equiv \sqrt{-1}$, the complex numbers can be defined as the space of numbers spanned by the vectors:

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ and } \begin{pmatrix} 0 \\ \imath \end{pmatrix} \tag{8-1}$$

so that any complex number can be written as

$$z = x \begin{pmatrix} 1 \\ 0 \end{pmatrix} + y \begin{pmatrix} 0 \\ i \end{pmatrix} \tag{8-2}$$

or just simply as

$$z = x + iy \tag{8-3}$$

where x and y are real numbers. $\mathrm{Re}z \equiv x$ and $\mathrm{Im}z \equiv y$.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Operations on complex numbers

Straightforward examples of addition, subtraction, multiplication, and division of complex numbers are demonstrated. An example that demonstrates that MATHEMATICA® doesn't make *a priori* assumptions about whether a symbol is real or complex. An example function that converts a complex number to its polar form is constructed.

```
imaginary = Sqrt[-1]                                                    1
(-imaginary)^2                                                          2
```

Complex numbers are composed of a real part + an imaginary part

```
z1 = a + i b;                                                           3
z2 = c + i d;
compadd = z1 + z2;                                                      4
compmult = z1 * z2;                                                     5
Simplify[compmult, a ∈ Reals &&                                         6
  b ∈ Reals && c ∈ Reals && d ∈ Reals ]
```

Mathematica does not assume that symbols are necessarily real...

```
Re[compadd]                                                             7
Im[compadd]
```

*However, the Mathematica function **ComplexExpand** does assume that the variables are real....*

```
ComplexExpand[Re[compadd]]                                              8
ComplexExpand[Im[compadd]]                                              9
ComplexExpand[Re[z1 / z2]]                                              10
ComplexExpand[compmult]                                                 11
ComplexExpand[Re[z1^3]]                                                 12
ComplexExpand[Im[z1^3]]
```

Function to convert to Polar Form

```
Pform[z_] := Abs[z] Exp[i Arg[z]]                                       13
```

*Note: the function **Arg[z]** returns an angle in the range $-\pi$ to π which measures the inclination of z with respect to the +Re axis in the complex plane.*

```
Pform[z1]                                                               14
Pform[z1 /. {a → 2, b → -π}]                                            15
ComplexExpand[Pform[z1]]                                                16
```

1–2: Just like `Pi` is a *mathematical constant*, the imaginary number is defined in MATHEMATICA® as something with the properties of i

3: Here, two numbers that are *potentially, but not necessarily* complex are defined.

4–5: Addition and multiplication are defined as for any symbol; here the results do not appear to be very interesting *because* the other symbols could themselves be complex...

6: And, `Simplify` doesn't help much even with assumptions.

7: The real and imaginary parts of a complex entity can be extracted with `Re` and `Im`. This demonstrates that MATHEMATICA® hasn't made assumptions about `a`, `b`, `c`, and `d`.

8-12: However, `ComplexExpand` does make assumptions that symbols are real and, here, demonstrate the rules for addition, multiplication, division, and exponentiation.

13–16: `Abs` calculates the magnitude (also known as modulus or absolute value) and `Arg` calculates the argument (or angle) of a complex number. Here, they are used to define a function (*Pform*) to convert and expression to an equivalent *polar form of a complex number*.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Because the complex basis can be written in terms of the vectors in Equation 8-1, it is natural to plot complex numbers in two dimensions—typically these two dimensions are the "complex plane" with $(0, \imath)$ associated with the y-axis and $(1, 0)$ associated with the x-axis.

The reflection of a complex number across the real axis is a useful operation. The image of a reflection across the real axis has some useful qualities and is given a special name—"the complex conjugate."

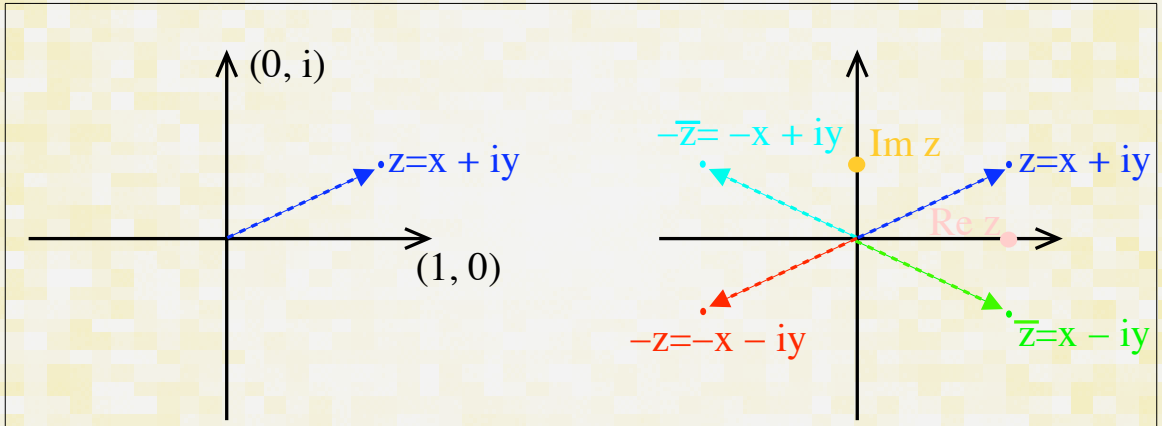


Figure 8-3: Plotting the complex number z in the complex plane: The complex conjugate $(\bar{z})$ is a *reflection* across the real axis; the minus $(-z)$ operation is an *inversion* through the origin; therefore $-(\bar{z}) = (\bar{-z})$ is equivalent to either a reflection across the imaginary axis or an inversion followed by a reflection across the real axis.
The real part of a complex number is the projection of the displacement in the real direction and also the average of the complex number and its conjugate: $\mathrm{Re}z = (z+\bar{z})/2$. The imaginary part is the displacement projected onto the imaginary axis, or the complex average of the complex number and its reflection across the imaginary axis: $\mathrm{Im}z = (z-\bar{z})/(2\imath)$.

Polar Form of Complex Numbers

There are physical situations in which a transformation from Cartesian (x, y) coordinates to polar (*or cylindrical*) coordinates (r, θ) simplifies the algebra that is used to describe the physical problem.

An equivalent coordinate transformation for complex numbers, $z = x + \imath y$, has an analogous simplifying effect for *multiplicative operations* on complex numbers. It has been demonstrated how the complex conjugate, $\bar{z}$, is related to a reflection—multiplication is related to a **counter-clockwise** rotation in the complex plane. Counter-clockwise rotation corresponds to increasing θ.

The transformations are:

$$\begin{aligned} (x, y) \rightarrow (r, \theta) &\begin{cases} x = r\cos\theta \\ y = r\sin\theta \end{cases} \\ (r, \theta) \rightarrow (x, y) &\begin{cases} r = \sqrt{x^2 + y^2} \\ \theta = \arctan\frac{y}{x} \end{cases} \end{aligned} \tag{8-4}$$

where $\arctan \in (-\pi, \pi]$.

Multiplication, Division, and Roots in Polar Form

One advantage of the polar complex form is the simplicity of multiplication operations:

DeMoivre's formula:

$$z^n = r^n(\cos n\theta + \imath \sin n\theta) \tag{8-5}$$

$$\sqrt[n]{z} = \sqrt[n]{z}(\cos\frac{\theta + 2k\pi}{n} + \imath \sin\frac{\theta + 2k\pi}{n}) \tag{8-6}$$

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Numerical Properties of Operations on Complex Numbers

Several examples demonstrate issues that arise when complex numbers are evaluated numerically.

```
ExactlyOne = Exp[2 π i]                                          1
NumericallyOne = Exp[N[2 π i]]                                   2
Chop[NumericallyOne]                                             3
Round[NumericallyOne]                                            4
ExactlyI = Exp[π i / 2]                                          5
NumericallyI = Exp[N[π i / 2]]                                   6
Round[NumericallyI]                                              7
Chop[NumericallyI]                                               8
ExactlyOnePlusI =
 ComplexExpand[√2 Exp[π i / 4]]                                  9
NumericallyOnePlusI =
 ComplexExpand[√2 Exp[N[π i / 4]]]                               10
Chop[NumericallyOnePlusI]                                        11
Round[NumericallyOnePlusI]                                       12
Round[1.5 - 3.5 Sqrt[-1]]                                        13
Re[NumericallyOnePlusI]                                          14
Im[NumericallyOnePlusI]                                          15
```

1: The relationship $e^{2\pi i} = 1$ is exact.

2: However, $e^{2.0\pi i}$ is *numerically* 1.

3: `Chop` removes small evalues that are presumed to be the result of numerical imprecision; it operates on complex numbers as well.

4: `Round` is useful for mapping a number to a simpler one in its neighborhood (such as the nearest integer).

5–8: Here, the difference between something that is exactly i and is numerically $1.0 \times i$ is demonstrated...

[: 9–15] And, this is similar demostration for $1 + i$ using its polar form as a starting point.

3.016 Home

Full Screen

Close

Quit

Exponentiation and Relations to Trignometric Functions

Exponentiation of a complex number is defined by:

$$e^z = e^{x+iy} = e^x(\cos y + \imath \sin y) \tag{8-7}$$

Exponentiation of a purely imaginary number advances the angle by rotation:

$$e^{\imath y} = \cos y + \imath \sin y \tag{8-8}$$

combining Eq. 8-8 with Eq. 8-7 gives the particularly useful form:

$$z = x + \imath y = re^{\imath\theta} \tag{8-9}$$

and the useful relations (obtained simply by considering the complex plane's geometry)

$$e^{2\pi\imath} = 1 \quad e^{\pi\imath} = -1 \quad e^{-\pi\imath} = -1 \quad e^{\frac{\pi}{2}\imath} = \imath \quad e^{-\frac{\pi}{2}\imath} = -\imath \tag{8-10}$$

Subtraction of powers in Eq. 8-8 and generalization gives known relations for trigonometric functions:

$$\begin{aligned} \cos z &= \frac{e^{\imath z} + e^{-\imath z}}{2} & \sin z &= \frac{e^{\imath z} - e^{-iz}}{2\imath} \\ \cosh z &= \frac{e^{z} + e^{-z}}{2} & \sinh z &= \frac{e^{z} - e^{-z}}{2} \\ \cos z &= \cosh \imath z & \imath \sin z &= \sinh \imath z \\ \cos \imath z &= \cosh z & \sin \imath z &= \imath \sinh z \end{aligned} \tag{8-11}$$

Complex Numbers in Roots to Polynomial Equations

Complex numbers frequently arise when solving for the roots of a polynomial equation. There are many cases in which a model of system's physical behavior depends on whether the roots of a polynomial are real or imaginary, and if the real part is positive. While evaluating the nature of the roots is straightforward conceptually, this often creates difficulties computationally. Frequently, ordered lists of solutions are maintained and the behavior each solution is followed.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Complex Roots of Polynomial Equations

Here we construct an artificial example of a model that depends on a single parameter in a quadratic polynomial and illustrate methods to analyze and visualize its roots. Methods to "peek" at the form of long expressions are also demonstrated.

```
sols = Solve[(x^4 - x^3 + x + 1) == 0, x]                                    1
x /. sols                                                                    2
Im[x /. sols]                                                                3
ComplexExpand[Im[x /. sols]]                                                 4
ComplexExpand[Im[x /. sols]] // N                                            5
ComplexExpand[Re[x /. sols]] // N                                            6
```

Generalize the above to a family of solutions.

```
bsols = Solve[(x^4 - x^3 + b*x + 1) == 0, x]                                 7
Dimensions[bsols]                                                            8
Short[bsols, 4]                                                              8
SolsbImag = ComplexExpand[Im[x /. bsols]];                                   9
Dimensions[SolsbImag]                                                        10
Short[SolsbImag[[1]]]                                                        10
SolsbReal = ComplexExpand[Re[x /. bsols]];                                   11
Plot[Evaluate[SolsbImag], {b, -10, 10}]                                      12
Plot[Evaluate[SolsbImag], {b, -10, 10},                                      13
 PlotStyle → Table[{Hue[1 - a / 6]}, {a, 1, 4}]]
Plot[Evaluate[SolsbReal], {b, -10, 10},                                      14
 PlotStyle → Table[{Hue[1 - a / 6]}, {a, 1, 4}]]
Plot[Evaluate[SolsbReal], {b, -10, 10},                                      15
 PlotStyle → Table[{Hue[1 - a / 6],
    Thickness[0.05 - .01 * a]}, {a, 1, 4}]]
Plot[Evaluate[x /. bsols],                                                   16
 {b, -10, 10}, PlotStyle → Thick]
```

1–6: Using a prototype fourth order equation, a list of solutions are obtained; the real and imaginary parts are computed.

7: The above is generalized to a single parameter b in the quartic equation; the conditions that the roots are real will be visualized. `bsols`, the list of solution rule-lists is long and complicated.

8: First, one must consider the structure of `bsols`. `Dimensions` indicates it is a list of four lists, each of length 1. `Dimensions` and `Short` used together, provides a practical method to observe the structure of a complicated expression without filling up the screen display.

9–11: Here, the real and complex parts of *each* of the solutions is obtained with `Re` and `Im` where the parameter b is assumed to be real via the use of `ComplexExpand`. **These may take a long time to evaluate on some computers.**

12–13: Which of the solutions (i.e., 1,2,3, or 4) is identified by a different color (if `Evaluate` is used inside the `Plot` function). In the first case, MATHEMATICA® 's default indexed colors are used, and in the second case they are set explicitely using `Hue` in `PlotStyle`.

14: Similarly, the real parts appear to converge to a single value when the imaginary parts (from above) appear...

15: But, the actual behavior is best illustrated by using `Thickness` to distinguish superimposed values. The behavior of real parts of this solution have what is called a *pitchfork structure*.

16: As of MATHEMATICA® 6, it is not necessary that the plotted function evaluate to a real value at each point. Now, only those points that evaluate to a real number will be graphed.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Lecture 9: Eigensystems of Matrix Equations

Reading:
Kreyszig Sections: 8.1, 8.2, 8.3 (pages334–338, 340–343, 345–348)

Eigenvalues and Eigenvectors of a Matrix

The conditions for which general linear equation

$$\underline{A}\vec{x} = \vec{b} \tag{9-1}$$

has solutions for a given matrix $\underline{A}$, fixed vector $\vec{b}$, and unknown vector $\vec{x}$ have been determined.

The operation of a matrix on a vector—whether as a physical process, or as a geometric transformation, or just a general linear equation—has also been discussed.

Eigenvalues and eigenvectors are among the most important mathematical concepts with a very large number of applications in physics and engineering.

An eigenvalue problem (associated with a matrix $\underline{A}$) relates the operation of a matrix multiplication on a particular vector $\vec{x}$ to its multiplication by a particular scalar λ.

$$\underline{A}\vec{x} = \lambda\vec{x} \tag{9-2}$$

This equation bespeaks that the matrix operation can be replaced—or is equivalent to—a stretching or contraction of the vector: "$\underline{A}$ has some vector $\vec{x}$ for which its multiplication is simply a scalar multiplication operation by λ." $\vec{x}$ is an *eigenvector* of $\underline{A}$ and λ is $\vec{x}$'s associated *eigenvalue*.

The condition that Eq. 9-2 has solutions is that its associated homogeneous equation:

$$(\underline{A} - \lambda \underline{I})\vec{x} = \vec{0} \tag{9-3}$$

has a zero determinant:

$$\det(\underline{A} - \lambda \underline{I}) = 0 \tag{9-4}$$

Eq. 9-4 is a polynomial equation in λ (the power of the polynomial is the same as the size of the square matrix).

The eigenvalue-eigenvector system in Eq. 9-2 is solved by the following process:

1. Solve the characteristic equation (Eq. 9-4) for each of its roots λ_i.
2. Each root λ_i is used as an eigenvalue in Eq. 9-2 which is solved for its associated eigenvector $\vec{x_i}$

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Calculating Matrix Eigenvalues and Eigenvectors

The symbolic computation of eigenvalues and eigenvectors is demonstrated for simple 2×2 matrices. This example is illustrative—more interesting uses would be for larger matrices. In this example, a "cheat" is employed so that a matrix with "interesting" eigenvalues and eigenvectors is used as computation fodder.

```
mymatrix = {{2 + Pi, -2 + Pi}, {-2 + Pi, 2 + Pi}};
mymatrix // MatrixForm
```
1

$$\begin{pmatrix} 2+\pi & -2+\pi \\ -2+\pi & 2+\pi \end{pmatrix}$$

Solve the characteristic equation for the two eigenvalues:

```
Solve[
 Det[mymatrix - λ IdentityMatrix[2]] == 0, λ]
```
2

Compute the eigenvectors:

```
Eigenvectors[mymatrix]
```
3

```
{evec1, evec2} = Eigenvectors[mymatrix]
```
4

Eigensystem *will solve for eigenvalues and corresponding eigenvectors in one step:*

```
Eigensystem[mymatrix]
```
5

```
{{2 π, 4}, {{1, 1}, {-1, 1}}}
```

Note the output format above: the first item in the list is a list of the two eigenvalues; the second item in the list is a list of the two corresponding eigenvectors. Thus, the eigenvector corresponding 2π is (1,1).

1: A "typical" 2×2 matrix `mymatrix` is defined for the calculations that follow. We will calculate its eigenvalues directly and with a built-in function.

2: Its eigenvalues can be obtained by by using `Solve` for the characteristic equation Eq. 9-4 in terms of λ.

3: And, its eigenvectors could be obtained by putting each eigenvalue back into Eq. 9-2 and then solving $\vec{x}$ for each unique λ. However, this tedious procedure can also be performed with `Eigenvectors`

4: Here, a matrix of eigenvectors is defined with named rows `evec1` and `evec2`.

5: `Eigensystem` generates the same results as `Eigenvectors` and `Eigenvalues` in one step.

3.016 Home
Full Screen
Close
Quit

The matrix operation on a vector that returns a vector that is in the same direction is an eigensystem. A physical system that is associated can be interpreted in many different ways:

geometrically The vectors $\vec{x}$ in Eq. 9-2 are the ones that are unchanged by the linear transformation on the vector.

iteratively The vector $\vec{x}$ that is processed (either forward in time or iteratively) by $\underline{A}$ increases (or decreases if $\lambda < 1$) along its direction.

In fact, the eigensystem can be (and will be many times) generalized to other interpretations and generalized beyond linear matrix systems.

Here are some examples where eigenvalues arise. These examples generalize beyond matrix eigenvalues.

- As an analogy that will become real later, consider the "harmonic oscillator" equation for a mass, m, vibrating with a spring-force, k, this is simply Newton's equation:

$$m\frac{d^2x}{dt^2} = kx \tag{9-5}$$

  If we treat the second derivative as some linear operator, $\mathcal{L}_{\text{spring}}$ on the position x, then this looks like an eigenvalue equation:

$$\mathcal{L}_{\text{spring}}x = \frac{k}{m}x \tag{9-6}$$

- Letting the positions x_i form a vector $\vec{x}$ of a bunch of atoms of mass m_i, the harmonic oscillator can be generalized to a bunch of atoms that are interacting as if they were attached to each other by springs:

$$m_i\frac{d^2x_i}{dt^2} = \sum_{i\text{'s near neighbors } j} k_{ij}(x_i - x_j) \tag{9-7}$$

3.016 Home
Full Screen
Close
Quit

For each position i, the j-terms can be added to each side, leaving and operator that looks like:

$$\mathcal{L}_{\text{lattice}} = \begin{pmatrix} m_1\frac{d^2}{dt^2} & -k_{12} & 0 & -k_{14} & \dots & & 0 \\ -k_{21} & m_2\frac{d^2}{dt^2} & -k_{23} & 0 & & \dots & 0 \\ \vdots & & \ddots & & & & \vdots \\ \vdots & & & m_i\frac{d^2}{dt^2} & & & \vdots \\ & & & & \ddots & & \\ & & & & & m_{N-1}\frac{d^2}{dt^2} & -k_{N-1\,N} \\ 0 & 0 & \dots & & & -k_{N\,N-1} & m_N\frac{d^2}{dt^2} \end{pmatrix} \tag{9-8}$$

The operator $\mathcal{L}_{\text{lattice}}$ has diagonal entries that have the spring (second-derivative) operator and one off-diagonal entry for each other atom that interacts with the atom associated with row i. The system of atoms can be written as:

$$\underline{k}^{-1}\mathcal{L}_{\text{lattice}}\vec{x} = \vec{x} \tag{9-9}$$

which is another eigenvalue equation and solutions are constrained to have unit eigenvalues—these are the 'normal modes.'

- To make the above example more concrete, consider a system of three masses connected by springs.

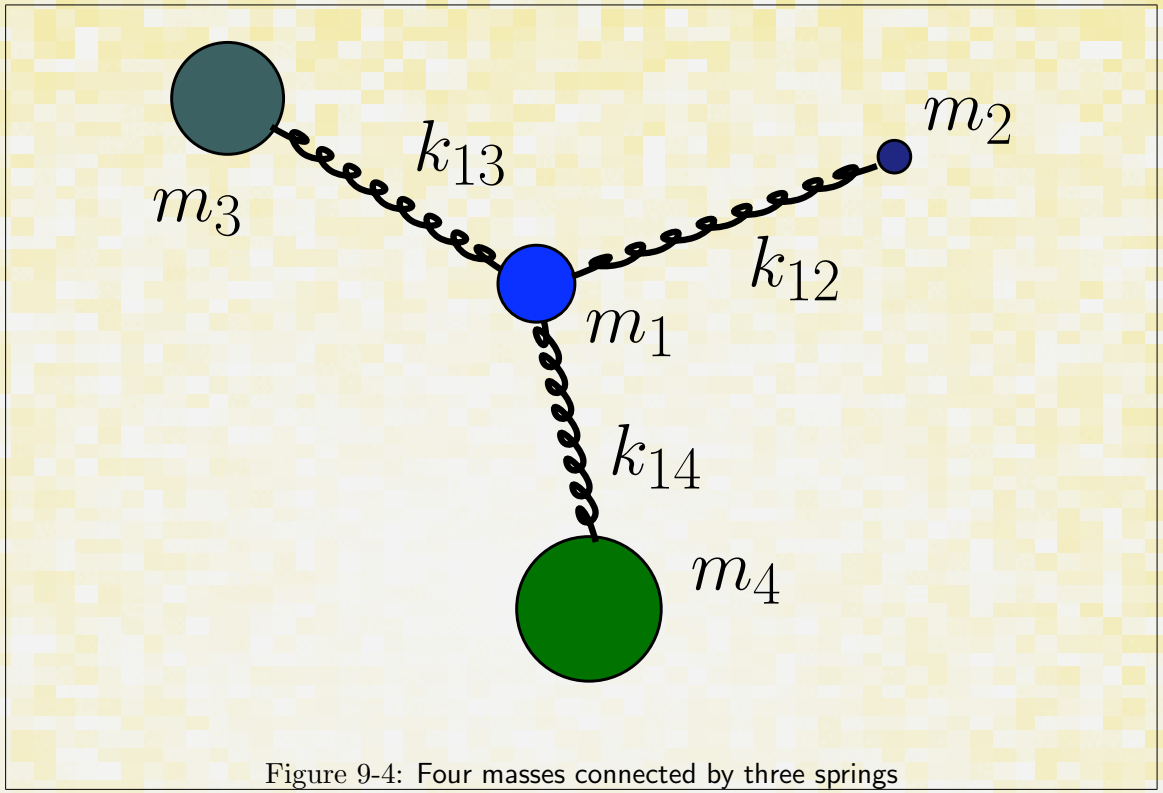


Figure 9-4: Four masses connected by three springs

The equations of motion become:

$$\begin{pmatrix} m_1\frac{d^2}{dt^2} & -k_{12} & -k_{13} & -k_{14} \\ -k_{12} & m_2\frac{d^2}{dt^2} & 0 & 0 \\ -k_{13} & 0 & m_2\frac{d^2}{dt^2} & 0 \\ -k_{14} & 0 & 0 & m_2\frac{d^2}{dt^2} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} k_{12}+k_{13}+k_{14} & 0 & 0 & 0 \\ 0 & k_{12} & 0 & 0 \\ 0 & 0 & k_{13} & 0 \\ 0 & 0 & 0 & k_{14} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \tag{9-10}$$

which can be written as

$$\mathcal{L}_{4\times 4}\vec{x} = \underline{k}\vec{x} \tag{9-11}$$

or

$$\underline{k}^{-1}\mathcal{L}_{4\times4}\vec{x} = \vec{x} \tag{9-12}$$

As will be discussed later, this system of equations can be "diagonalized" so that it becomes four independent equations. Diagonalization depends on finding the eigensystem for the operator.

- The one-dimensional Shrödinger wave equation is:

$$-\frac{\hbar}{2m}\frac{d^2\psi(x)}{dx^2} + U(x)\psi(x) = E\psi(x) \tag{9-13}$$

where the second derivative represents the kinetic energy and $U(x)$ is the spatial-dependent potential energy. The "Hamiltonian Operator" $\mathcal{H} = -\frac{\hbar}{2m}\frac{d^2}{dx^2} + U(x)$, operates on the wave-function $\psi(x)$ and returns the wave-function's total energy multiplied by the wave-vector;

$$\mathcal{H}\psi(x) = E\psi(x) \tag{9-14}$$

This is another important eigenvalue equation (and concept!)

Symmetric, Skew-Symmetric, Orthogonal Matrices

Three types of matrices occur repeatedly in physical models and applications. They can be placed into three categories according to the conditions that are associated with their eigenvalues:

All real eigenvalues Symmetric matrices—those that have a "mirror-plane" along the northwest–southeast diagonal ($\underline{A} = \underline{A}^T$)—must have all real eigenvalues.

Hermitian matrices—the complex analogs of symmetric matrices—in which the reflection across the diagonal is combined with a complex conjugate operation ($a_{ij} = \bar{a_{ji}}$), must also have all real eigenvalues.

All imaginary eigenvalues Skew-symmetric (diagonal mirror symmetry combined with a minus) matrices ($-\underline{A} = \underline{A}^T$) must have all complex eigenvalues.

Skew-Hermitian matrices—-the complex analogs of skew-symmetric matrices ($a_{ij} = -\bar{a_{ji}}$)—have all imaginary eigenvalues.

Unitary Matrices: unit determinant Real matrices that satisfy $\underline{A}^T = \underline{A}^{-1}$ have the property that product of all the eigenvalues is ± 1. These are called *orthogonal* matrices and they have *orthonormal* rows. Their determinants are also ± 1.

This is generalized by complex matrices that satisfy $\bar{\underline{A}}^T = \underline{A}^{-1}$. These are called *unitary* matrices and their (complex) determinants have magnitude 1. Orthogonal matrices, $\underline{A}$, have the important physical property that they preserve the inner product: $\vec{x} \cdot \vec{y} = (\underline{A}\vec{x}) \cdot (\underline{A}\vec{y})$. When the orthogonal matrix is a rotation, the interpretation is that the vectors maintain their relationship to each other if they are both rotated.

3.016 Home

Full Screen

Close

Quit

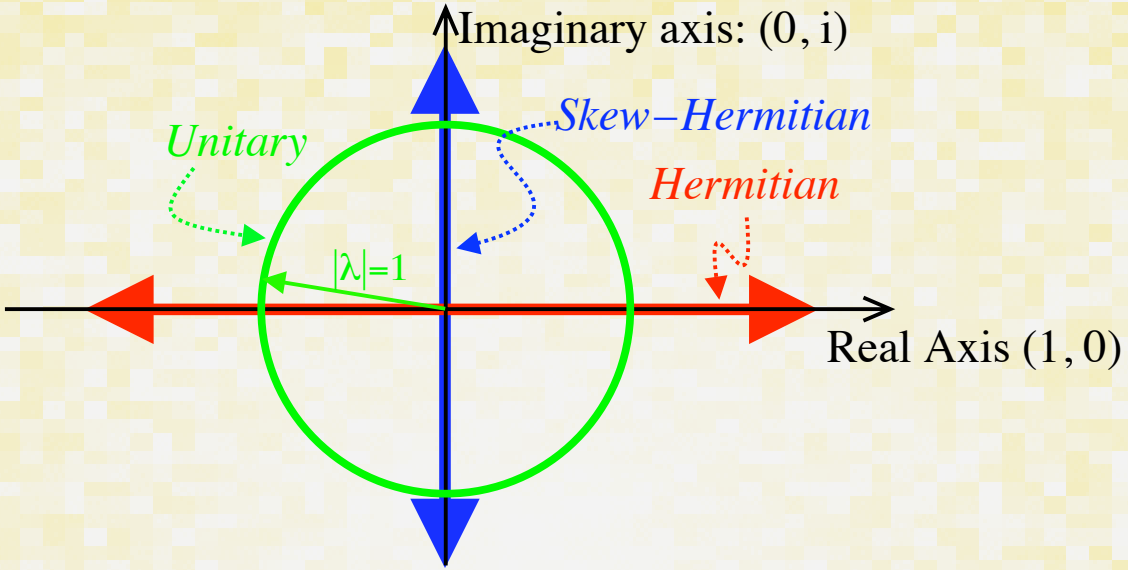


Figure 9-5: The Symmetric (complex Hermetic), Skew-Symmetric (complex Skew-Hermitian), Orthogonal, and Unitary Matrix sets characterized by the position of their eigenvalues in the complex plane. *(Hermits live alone on the real axis; Skew-Hermits live alone on the imaginary axis)*

Orthogonal Transformations

Multiplication of a vector by an orthogonal matrix is equivalent to an orthogonal geometric transformation on that vector.

For orthogonal transformation, the inner product between any two vectors is *invariant*. That is, the inner product of two vectors is always the same as the inner product of their images under an orthogonal transformation. Geometrically, the

projection (or the angular relationship) is unchanged. This is characteristic of a rotation, or a reflection, or an inversion.

Rotations, reflections, and inversions are orthogonal transformations. The product of orthogonal matrices is also an orthogonal matrix.

Coordinate Transformations to The Eigenbasis

Here we demonstrate that a matrix, composed of columns of constructed eigenvectors of a matrix, can be used to *diagonalize a matrix*, and the resulting diagonal entries are the matrix eigenvalues.

```
simtrans = {evec2, evec1} // Transpose;
simtrans // MatrixForm
```
1

```
Inverse[simtrans].mymatrix.simtrans //
  Simplify // MatrixForm
```
2

▼ *Shows that the transformation to the diagonal basis is a rotation of $\pi/4$*

▼ *Which makes sense considering in initialization steps that mymatrix was created with a rotation on $\pi/4$ of a diagonal matrix*

The next command produces an orthonormal basis of the eigenspace (i.e., the eigenvectors are of unit magnitude):

```
Orthogonalize[Eigenvectors[mymatrix],
  Method → "GramSchmidt"] // MatrixForm
```
3

The command RotationTransform computes a matrix that will rotate vectors ccw about the origin in two dimensions, by a specified angle:

```
RotationTransform[π/4][{{1, 0}, {0, 1}}] //
 MatrixForm
```
4

This last result shows that the transformation to the eigenvector space involves rotation by $\pi/4$--and that the matrix corresponding to the eigenvectors produces this same transformation

Here is a demonstration of the general result $A\,\vec{x}_i = \lambda_i\,\vec{x}_i$, where $\vec{x}$ is an eigenvector and λ its corresponding eigenvalue:

```
evec1
evec2
```
5

```
mymatrix.evec1
```
6

```
mymatrix.evec2
```
7

MatrixPower multiplies a matrix by itself n times...

```
MatrixPower[mymatrix, 12].evec2 // Simplify
```
8

1: The matrix *simtrans* is constructed by assigning rows defined by the eigenvectors from the previous example and then transposing (`Transpose`) so that the eigenvectors are the columns.

2: The original matrix is left-multiplied by the *inverse* of *simtrans* and right-multiplied by *simtrans* ; the result will be a diagonal matrix with the original matrix's eigenvalues as diagonal entries.

3: The eigenvectors are already orthogonal. There is a process called *Gram-Schmidt* orthogonalization used to define a set of vectors that are normal to each other. These orthogonalized vectors form a convenient *basis* Linear combinations of the basis vectors can produce *any other vector* in same vector space; for the orthogonalized basis, the basis vectors are as independent as possible. Here, `GramSchmidt` produces vectors that are also *normalized to unit vectors*. This, and other useful vector functions such as `Normalize` are available for common vector operations.

4: The geometrical interpretation of this operation can be found by comparing a matching `MatrixTransform` to the matric composed of eigenvector columns. Here, we see that eigenvector-matrix is equivalent to the $\pi/4$ rotation matrix.

6–7: These demonstrate that Eq. 9-2 is true.

8: This demonstrates that $\underline{A}^n\vec{x} = \lambda^n\vec{x}$.

3.016 Home

Full Screen

Close

Quit

Lecture 10: Real Eigenvalue Systems; Transformations to Eigenbasis

Reading:
Kreyszig Sections: 8.4, 8.5 (pages349–354, 356–361)

Similarity Transformations

A matrix has been discussed as a linear operation on vectors. The matrix itself is defined in terms of the coordinate system of the vectors that it operates on—and that of the vectors it returns.

In many applications, the coordinate system (or laboratory) frame of the vector that gets operated on is the same as the vector gets returned. This is the case for almost all physical properties. For example:

- In an electronical conductor, local current density, $\vec{J}$, is linearly related to the local electric field $\vec{E}$:

$$\underline{\rho}\vec{J} = \vec{E} \tag{10-1}$$

- In a thermal conductor, local heat current density is linearly related to the gradient in temperature:

$$\underline{k}\nabla T = \vec{j_Q} \tag{10-2}$$

- In diamagnetic and paramagnetic materials, the local magnetization, $\vec{B}$ is related to the applied field, $\vec{H}$:

$$\underline{\mu}\vec{H} = \vec{B} \tag{10-3}$$

- In dielectric materials, the local total polarization, $\vec{D}$, is related to the applied electric field:

$$\underline{\kappa}\vec{E} = \vec{D} = \kappa_o\vec{E} + \vec{P} \tag{10-4}$$

When $\vec{x}$ and $\vec{y}$ are vectors representing a physical quantity in Cartesian space (such as force $\vec{F}$, electric field $\vec{E}$, orientation of a plane $\hat{n}$, current $\vec{j}$, etc.) they represent something *physical*. They don't change if we decide to use a different space in which to represent them (such as, exchanging x for y, y for z, z for x; or, if we decide to represent length in nanometers instead of inches, or if we simply decide to rotate the system that describes the vectors. The representation of the vectors themselves may change, but they stand for the same thing.

One interpretation of the operation $\underline{A}\vec{x}$ has been described as geometric transformation on the vector $\vec{x}$. For the case of orthogonal matrices $\underline{A_{orth}}$, geometrical transformations take the forms of rotation, reflection, and/or inversion.

Suppose we have some *physical* relation between two physical vectors in some coordinate system, for instance, the general form of Ohm's law is:

$$\vec{J} = \underline{\sigma}\vec{E}$$

$$\begin{pmatrix} J_x \\ J_y \\ J_z \end{pmatrix} = \begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{pmatrix} \begin{pmatrix} E_x \\ E_y \\ E_z \end{pmatrix} \tag{10-5}$$

The matrix (actually it is better to call it a *rank-2 tensor*) $\underline{\sigma}$ is a physical quantity relating the amount of current that flows (in a direction) proportional to the applied electric field (perhaps in a different direction). $\underline{\sigma}$ is the "conductivity tensor" for a particular material.

The physical law in Eq. 10-5 can be expressed as an inverse relationship:

$$\vec{E} = \underline{\rho}\vec{j}$$

$$\begin{pmatrix} E_x \\ E_y \\ E_z \end{pmatrix} = \begin{pmatrix} \rho_{xx} & \rho_{xy} & \rho_{xz} \\ \rho_{xy} & \rho_{yy} & \rho_{yz} \\ \rho_{xz} & \rho_{yz} & \rho_{zz} \end{pmatrix} \begin{pmatrix} j_x \\ j_y \\ j_z \end{pmatrix} \tag{10-6}$$

where the resistivity tensor $\underline{\rho} = \underline{\sigma}^{-1}$.

3.016 Home
Full Screen
Close
Quit

What happens if we decide to use a new coordinate system (i.e., one that is rotated, reflected, or inverted) to describe the relationship expressed by Ohm's law?

The two vectors must transform from the "old" to the "new" coordinates by:

$$\begin{aligned} \underline{A_{orth}^{old\rightarrow new}}\vec{E^{old}} &= \vec{E^{new}} \qquad \underline{A_{orth}^{old\rightarrow new}}\vec{j^{old}} = \vec{j^{new}} \\ \underline{A_{orth}^{new\rightarrow old}}\vec{E^{new}} &= \vec{E^{old}} \qquad \underline{A_{orth}^{new\rightarrow old}}\vec{j^{new}} = \vec{j^{old}} \end{aligned} \tag{10-7}$$

Where is simple proof will show that:

$$\begin{aligned} \underline{A_{orth}^{old\rightarrow new}} &= \underline{A_{orth}^{new\rightarrow old}}^{-1} \\ \underline{A_{orth}^{new\rightarrow old}} &= \underline{A_{orth}^{old\rightarrow new}}^{-1} \\ \underline{A_{orth}^{new\rightarrow old}} &= \underline{A_{orth}^{old\rightarrow new}}^{T} \\ \underline{A_{orth}^{new\rightarrow old}} &= \underline{A_{orth}^{old\rightarrow new}}^{T} \end{aligned} \tag{10-8}$$

where the last two relations follow from the special properties of orthogonal matrices.

How does the physical law expressed by Eq. 10-5 change in a new coordinate system?

$$\begin{aligned} \text{in old coordinate system: } \vec{j^{old}} &= \underline{\chi^{old}}\vec{E^{old}} \\ \text{in new coordinate system: } \vec{j^{new}} &= \underline{\chi^{new}}\vec{E^{new}} \end{aligned} \tag{10-9}$$

To find the relationship between $\underline{\chi^{old}}$ and $\underline{\chi^{new}}$: For the first equation in 10-9, using the transformations in Eqs. 10-7:

$$\underline{A_{orth}^{new\rightarrow old}}\vec{j^{new}} = \underline{\chi^{old}}\,\underline{A_{orth}^{new\rightarrow old}}\vec{E^{new}} \tag{10-10}$$

and for the second equation in 10-9:

$$\underline{A_{orth}^{old\rightarrow new}}\vec{j^{old}} = \underline{\chi^{new}}\,\underline{A_{orth}^{old\rightarrow new}}\vec{E^{old}} \tag{10-11}$$

Left-multiplying by the inverse orthogonal transformations:

$$\begin{aligned} \underline{A_{orth}^{old\rightarrow new}\, A_{orth}^{new\rightarrow old}}\, \vec{j^{new}} &= \underline{A_{orth}^{old\rightarrow new}\, \chi^{old}\, A_{orth}^{new\rightarrow old}}\, \vec{E^{new}} \\ \underline{A_{orth}^{new\rightarrow old}\, A_{orth}^{old\rightarrow new}}\, \vec{j^{old}} &= \underline{A_{orth}^{new\rightarrow old}\, \chi^{new}\, A_{orth}^{old\rightarrow new}}\, \vec{E^{old}} \end{aligned} \tag{10-12}$$

Because the transformation matrices are inverses, the following relationship between *similar* matrices in the old and new coordinate systems is:

$$\begin{aligned} \underline{\chi^{old}} &= \underline{A_{orth}^{old\rightarrow new}\, \chi^{new}\, A_{orth}^{new\rightarrow old}} \\ \underline{\chi^{new}} &= \underline{A_{orth}^{new\rightarrow old}\, \chi^{old}\, A_{orth}^{old\rightarrow new}} \end{aligned} \tag{10-13}$$

The $\underline{\chi^{old}}$ is said to be *similar* to $\underline{\chi^{new}}$ and the double multiplication operation in Eq. 10-13 is called *a similarity transformation.*

Stresses and Strains

Stresses and strains are rank-2 tensors that characterize the mechanical state of a material.

A spring is an example of a one-dimensional material—it resists or exerts force in one direction only. A volume of material can exert forces in all three directions simultaneously—and the forces need not be the same in all directions. A volume of material can also be "squeezed" in many different ways: it can be squeezed along any one of the axis or it can be subjected to squeezing (*or smeared*) around any of the axes[4]

All the ways that a force can be applied to small element of material are now described. A force divided by an area is a *stress*—think of it the area density of force.

$$\sigma_{ij} = \frac{F_i}{A_j} \qquad (\text{i.e., } \sigma_{xz} = \frac{F_x}{A_z} = \sigma_{xz} = \frac{\vec{F}\cdot\hat{i}}{\vec{A}\cdot\hat{k}}) \tag{10-14}$$

A_j is a plane with its normal in the $\hat{j}$-direction (or the projection of the area of a plane $\vec{A}$ in the direction parallel to $\hat{j}$)

[4] Consider a blob of modeling clay—you can deform it by placing between your thumbs and one opposed finger; you can deform it by simultaneously squeezing with two sets of opposable digits; you can "smear" it by pushing and pulling in opposite directions. These are examples of uniaxial, biaxial, and shear stress.

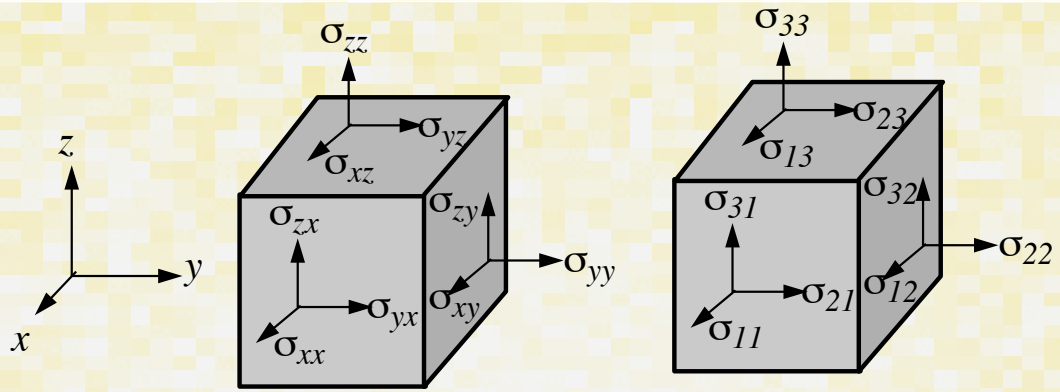


Figure 10-6: Illustration of stress on an oriented volume element.

$$\sigma_{ij} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix} \tag{10-15}$$

There is one special and very simple case of elastic stress, and that is called the hydrostatic stress. It is the case of pure pressure and there are no shear (off-diagonal) stresses (i.e., all $\sigma_{ij} = 0$ for $i \neq j$, and $\sigma_{11} = \sigma_{22} = \sigma_{33}$). An equilibrium system composed of a body in a fluid environment is always in hydrostatic stress:

$$\sigma_{ij} = \begin{bmatrix} -P & 0 & 0 \\ 0 & -P & 0 \\ 0 & 0 & -P \end{bmatrix} \tag{10-16}$$

where the pure hydrostatic pressure is given by P.

Strain is also a rank-2 tensor and it is a physical measure of a how much a material changes its shape.[5]

[5]It is unfortunate that the words of these two related physical quantities, stress and strain, sound so similar. Strain measures the change in

Why should strain be a rank-2 tensor?

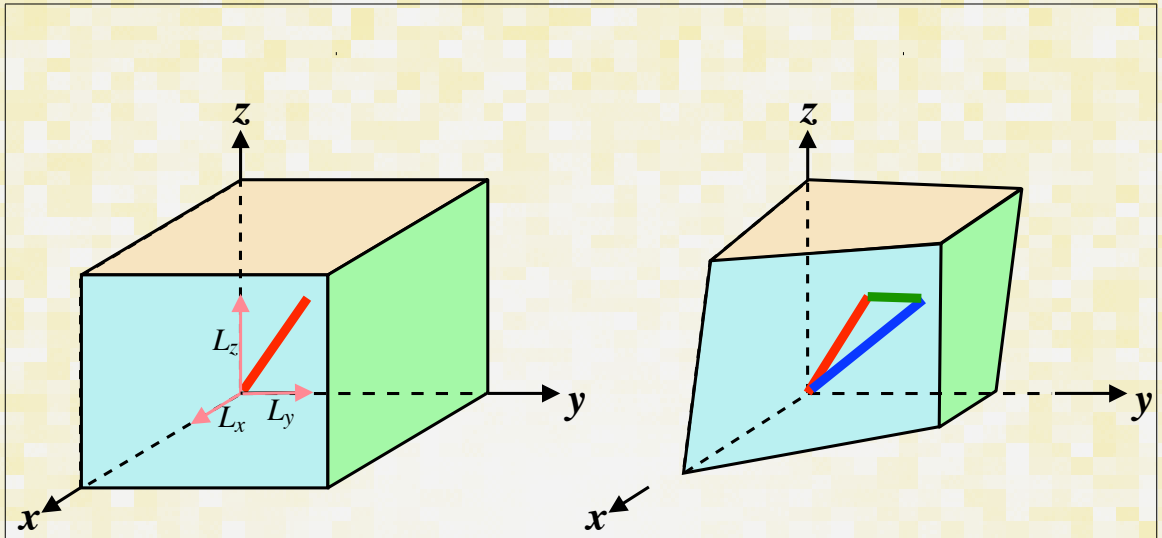


Figure 10-7: Illustration of how strain is defined: imagine a small line-segment that is aligned with a particular direction (one set of indices for the direction of the line-segment); after deformation the end-points of the line segment define a new line-segment in the deformed state. The difference in these two vectors is a vector representing how the line segment has changed from the initial state into the deformed state. The difference vector can be oriented in any direction (the second set of indices)—the strain is a representation of "a difference vectors for all the oriented line-segments" divided by the length of the original line.

Or, using the same idea as that for stress:

$$\epsilon_{ij} = \frac{\Delta L_i}{L_j} \qquad (\text{i.e., } \epsilon_{xz} = \frac{\Delta L_x}{L_z} = \epsilon_{xz} = \frac{\Delta \vec{L} \cdot \hat{i}}{\vec{L} \cdot \hat{k}}) \tag{10-17}$$

geometry of a body and stress measures the forces that squeeze or pull on a body. Stress is the press; Strain is the gain.

If a body that is being stressed hydro-statically is isotropic, then its response is pure dilation (in other words, it expands or shrinks uniformly and without shear):

$$\epsilon_{ij} = \begin{bmatrix} \Delta/3 & 0 & 0 \\ 0 & \Delta/3 & 0 \\ 0 & 0 & \Delta/3 \end{bmatrix} \tag{10-18}$$

$$\Delta = \frac{dV}{V} \tag{10-19}$$

So, for the case of hydrostatic stress, the work term has a particularly simple form:

$$V \sum_{i=1}^{3} \sum_{j=1}^{3} \sigma_{ij} d\epsilon_{ij} = -P dV \tag{10-20}$$

$$V \sigma_{ij} d\epsilon_{ij} = -P dV \qquad \text{(summation convention)}$$

This expression is the same as the rate of work performed on a compressible fluid, such as an ideal gas.

EigenStrains and EigenStresses

For any strain matrix, there is a choice of an coordinate system where line-segments that lie along the coordinate axes always deform parallel to themselves (i.e., they only stretch or shrink, they do not twist).

For any stress matrix, there is a choice of an coordinate system where all shear stresses (the off-diagonal terms) vanish and the matrix is diagonal.

These coordinate systems define the eigenstrain and eigenstress. The matrix transformation that takes a coordinate system into its eigenstate is of great interest because it simplifies the mathematical representation of the physical system.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Representations of Stress (or Strain) in Rotated Coordinate Systems

A demonstration of rotating a quasi-two dimensional stress state is given. Convenient forms for the stress in any coordinate system are derived. The stress invariants are demonstrated.

This is a general state, we will rotate about the z-axis and compare the result to a general two-dimensional stress state.

```
σtensordiag = {{σprinc_xx, 0, 0}, {0, σprinc_yy, 0}, {0, 0, σprinc_zz}};   1
σtensordiag // MatrixForm
```

```
rotmat[θ_] := {{Cos[θ], -Sin[θ], 0}, {Sin[θ], Cos[θ], 0}, {0, 0, 1}};   2
rotmat[θ] // MatrixForm
```

Transformation to general two-dimensional stress state coordinate system by rotating the principal system by θ around z-axis

```
σrot = Simplify[Transpose[rotmat[θ]].
    σtensordiag.rotmat[θ]];   3
σrot // MatrixForm
```

Writing the same equation in a slightly different way...

```
σrotalt = Collect[
   σrot // TrigReduce, {Cos[2 θ], Sin[2 θ]}];   4
σrotalt // MatrixForm
```

Naming the coefficients of the rotated two-dimensional state:

```
σlabMat = {{σlab_xx, σlab_xy, σlab_xz}, {σlab_xy, σlab_yy, σlab_yz}, {σlab_xz, σlab_yz, σlab_zz}} = σrotalt;   5
σlabMat // MatrixForm
```

Looking at the x-y components of stress (i.e, the upper-left 2x2 submatrix), notice that there are two invariants of the generalized two-dimensional stress state: The trace and the determinant:

```
Simplify[σlab_xx + σlab_yy]   6
```

```
Simplify[σlab_xx σlab_yy - (σlab_xy)^2]   7
```

Do not depend on θ; thus illustrating the invariance of these quantities under rotation of coordinate rotations.

1: The problem is done in reverse by finding the backwards rotation of a diagonal matrix. This is the stress in the principle coordinate system (it is diagonalized with eigenvalue entries) Any rotation similarity transformation on this matrix is the equivalent stress in the rotated frame.

2: This is the rotation operator by counter-clockwise angle θ about the z-axis.

3: Therefore σ`rot`, obtained by the similarity transformation, is the stress in any rotation about the z-axis. It is a quasi-two dimensional state defined by $\sigma_{zx} = \sigma yz = 0$.

4: The rotation matrix factors well using the double angle formulas; here we use `TrigReduce` to convert powers of trigonometric functions into functions of multiples of their angle, and the `Collect` the double-angle terms.

5: This can be compared with the general form of any quasi-two-dimensional (x–y plane) stress that has the same principle stresses identified above. In the rotated (i.e., laboratory) frame, the stresses are geometrically related to the circle plotted in 10-8, which will be also visualized in the following examples.

6: Here, we use double assignment for the laboratory-frame matrix and, simultaneously, its elements.

7: This will show that the trace of the stress tensor is independent of rotation—this is a general property for any unitary transformation.

8: Like the trace, the determinant is also an *matrix invariant.*

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Principal Axes: Mohr's Circle of Two-Dimensional Stress

By diagonalizing a quasi-two-dimensional stress tensor, the equations for Mohr's circle of stress (Fig. 10-8) are derived.

1. σlab_{xx} in laboratory system rotated by θ from principal axis system

```
σlab_xx                                                                    1
```

2. σlab_{yy} in laboratory system rotated by θ from principal axis system

```
σlab_yy                                                                    2
```

3. σlab_{xy} in laboratory system rotated by θ from principal axis system

```
σlab_xy                                                                    3
```

```
uniaxial10 = {σprinc_xx -> 10, σprinc_yy -> 0}                             4
```

```
ParametricPlot[{σlab_xx, σlab_xy} /. uniaxial10
 , {θ, 0, π}, AxesLabel →
  {"normal stress", "shear stress"},
 AspectRatio → 1, PlotLabel → " \t \t Mohr
    Circle for 10 MPa Uniaxial Tension",
 PlotStyle → {Thickness[0.01], Hue[1]}]                                    5
```

```
uniaxialother = {σprinc_xx -> 30, σprinc_yy -> 10}                         6
```

```
ParametricPlot[
 {σlab_xx, σlab_xy} /. uniaxialother
 , {θ, 0, π}, AxesLabel → {"normal stress",
   "shear stress"}, AspectRatio → 1,
 PlotRange -> {{0, 40}, {-20, 20}},
 PlotLabel → " \t \t Mohr Circle
    for σprinc_xx= 30   σprinc_yy=10",
 PlotStyle → {Thickness[0.01], Hue[1]}]                                    7
```

1–3: These are the forms of the three two-dimensional stress components are simple expressions in terms of 2θ. We will see how these equations produce a circle.

4: This is a rule that defines *a particular* stress state in terms of the principal stresses (here given by 10 and 0).

5: 5–7 Using `ParametricPlot` for $\sigma_{xx}(\theta)$ and $\sigma_{xy}(\theta)$, an example of Mohr's circle is plotted for examples of principle stress for (0,10) and (30,10). The interpretation of this figure is given in Fig. 10-8.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

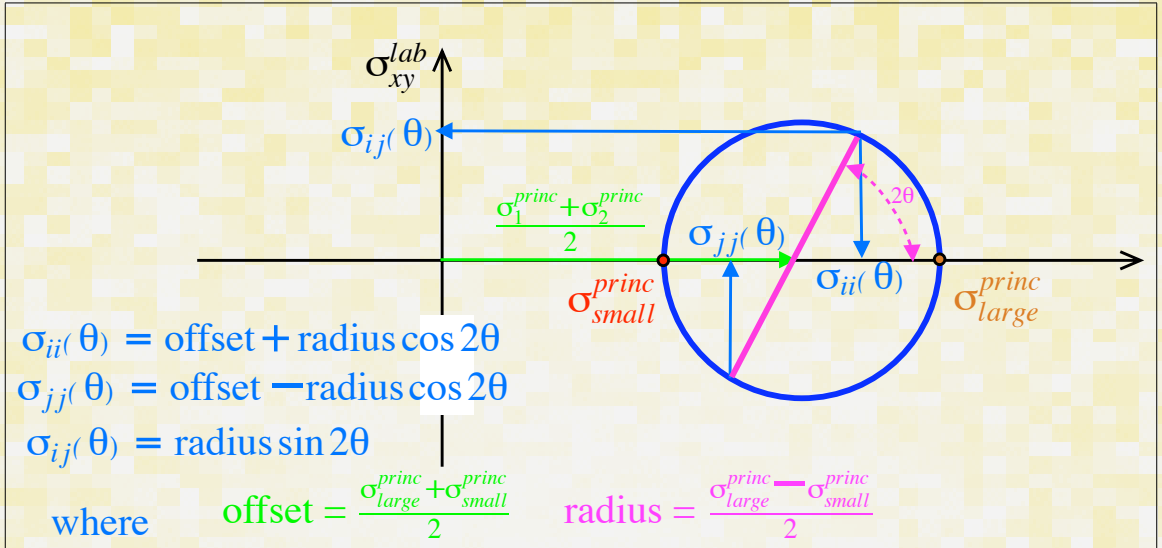


Figure 10-8: Mohr's circle of stress is a way of graphically representing the two-dimensional stresses of identical stress states, but in rotated laboratory frames.
The center of the circle is displaced from the origin by a distance equal to the average of the principal stresses (or average of the eigenvalues of the stress tensor).
The maximum and minimum stresses are the eigenvalues—and they define the diameter in the principal $\theta = 0$ frame.
Any other point on the circle gives the stress tensor in a frame rotated by 2θ from the principal axis using the construction illustrated by the blue lines (and equations).

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualization Example: Graphics for Mohr's Circle

Our goal is to produce a manipulatable and interactive graphical representation for Mohr's circle of stress. We break the problem up by creating the individual graphical elements that appear in Fig. 10-8. These functions will be utilized in the following example.

```
mohrs[off_, rad_] :=
 {Red, Thick, Circle[{offset, 0}, radius]}
```
1

```
s12graph[s11_, s12_] := {Darker[Orange],
  Arrow[{{s11, s12}, {0, s12}}],
  Text[s12, {0, s12},
   {-1.5, -1.5}, Background → White]}
```
2

```
s22graph[s22_, s12_] :=
 {Blue, Arrow[{{s22, -s12}, {s22, 0}}],
  Text[s22, {s22, 0}, {0, -1},
   Background → White]}
```
3

```
s11graph[s11_, s12_] := {Darker[Green],
  Dynamic[Arrow[{{s11, s12}, {s11, 0}}]],
  Dynamic[Text[s11, {s11, 0},
    {0, 1}, Background → White]]}
```
4

```
diamgraph[s11_, s12_, s22_] :=
 {Line[{{s22, -s12}, {s11, s12}}]}
```
5

```
anglegraph[twotheta_] :=
 {Purple, Dashed, Circle[{offset, 0},
   1.2 * radius, {0, twotheta}], Text[
   Style["2θ=" <> ToString[twotheta 180 / Pi],
    Medium], {offset, 0} + 1.2 * radius *
     {Cos[twotheta / 2], Sin[twotheta / 2]},
   Background → White]}
```
6

```
titlegraph[s11_, s12_, s22_] :=
 Text[MatrixForm[
   {{Text[Style[s11, Darker[Green], Large]],
     Text[Style[s12, Darker[Orange], Large]]},
    {Text[Style[s12, Darker[Orange], Large]],
     Text[Style[s22, Blue, Large]]}}]]
```
7

1: The function *mohrs* will produce *graphical primitives* at an arbitrary `offset`$=(\sigma_{xx}+\sigma_{yy})/2$ and `radius`$==(\sigma_{xx}-\sigma_{yy})/2$.

2: This will be an annotated dark orange horizontal arrow for the shear stress value σ_{xy}

3: This will be an annotated blue vertical arrow for the stress σ_{yy}

4: This will be an annotated dark green vertical arrow for the stress σ_{xx}

5: This will illustrate the diameter of the circle in the current θ-orientation.

6: This will annotate the current angle 2θ.

7: This creates a title for the graph with entries colored corresponding to the arrows on the graph.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

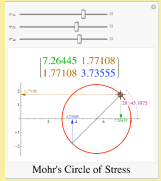
notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Interactive Graphics Demonstration for Mohr's Circle

We create graphics in which we can use the mouse to drag elements that dynamically update the visualization. In this case, we create a handle that will allow visualization of two-dimensional stress in an arbitrary rotation. The dynamic graphics are enclosed within `Manipulate` so that the original stress state can be set by the user.

```
Manipulate[
 offset = (σ11 + σ22) / 2; radius = (σ11 - σ22) / 2;
 ThetaInit = ArcSin[σ12 / radius] / 2;
 DynamicModule[{twotheta = 2 ThetaInit,
   s11 = 3, s12 = 4, s22}, LocatorPane[
   Dynamic[({radius Cos[twotheta] + offset,
      radius Sin[twotheta]}),
    (twotheta = Mod[Apply[ArcTan,
         (# - {offset, 0})], 2 Pi]) &],
   s11 = Dynamic[offset + radius
       Cos[twotheta]];
   s12 = Dynamic[ radius Sin[twotheta]];
   s22 =
    Dynamic[offset - radius Cos[twotheta]];
   Graphics[{mohrs[offset, radius],
     Dynamic[anglegraph[twotheta]],
     Dynamic[diamgraph[s11, s12, s22]],
     Dynamic[s12graph[s11, s12]] ,
     Dynamic[s22graph[s22, s12]] ,
     Dynamic[s11graph[s11, s12]] },
    Axes → True, AxesOrigin → {0, 0},
    PlotLabel →
     Dynamic[titlegraph[s11, s12, s22]]]
 ]
], {{σ11 , 8}, -1, 11},
{{σ22 , 3.0}, -5, 7}, {{σ12 , 2}, -5, 5},
FrameLabel → Text[
  Style["Mohr's Circle of Stress", Large]]]
```

1



1: `Manipulate` will get sliders for each of σ_{11}, σ_{22}, and σ_{33} (see note below about these values).
The circle offset, radius, and the value of 2θ are calculated first from the Mohr's equation formula.
Because we will be changing θ, its current value and the stress state for that rotation will change.
We make these variables local (as in `Module`), but further specify that they will be dynamically updated by using `DynamicModule`.
We use `LocatorPane` to place a "handle" at a specific spot which can be updated with the mouse.
As the value of θ is changed, we must inform the symbols (here `twotheta`, `s11`, `s12`, and `s22`) are to be updated by embedding them in the `Dynamic` function.
The graphics are drawn using the functions from the previous example.
In this example, no precautions are made to ensure that the stress components will be real in every orientation.

3.016 Home
Full Screen
Close
Quit

The example above, where a matrix (rank-2 tensor) represents a material property, can be understood with a useful geometrical interpretation.

For the case of the conductivity tensor $\underline{\sigma}$, the dot product $\vec{E} \cdot \vec{j}$ is a scalar related to the local energy dissipation:

$$e = \vec{E}^T \underline{\sigma} \vec{E} \tag{10-21}$$

The term on the right-hand-side is called a quadratic form, as it can be written as:

$$\begin{aligned} e = & \sigma_{11} x_1^2 + \sigma_{12} x_1 x_2 + \sigma_{13} x_1 x_3 + \\ & \sigma_{21} x_1 x_2 + \sigma_{22} x_2^2 + \sigma_{23} x_2 x_3 + \\ & \sigma_{31} x_1 x_3 + \sigma_{32} x_2 x_3 + \sigma_{33} x_3^2 \end{aligned} \tag{10-22}$$

or, because $\underline{\sigma}$ is symmetric:

$$\begin{aligned} e = & \sigma_{11} x_1^2 + 2\sigma_{12} x_1 x_2 + 2\sigma_{13} x_1 x_3 + \\ & \sigma_{22} x_2^2 + 2\sigma_{23} x_2 x_3 + \\ & \sigma_{33} x_3^2 \end{aligned} \tag{10-23}$$

It is not unusual for such quadratic forms to represent energy quantities. For the case of paramagnetic and diamagnetic materials with magnetic permeability tensor $\underline{\mu}$, the energy per unit volume due to an applied magnetic field $\vec{H}$ is:

$$\frac{E}{V} = \frac{1}{2} \vec{H}^T \underline{\mu} \vec{H} \tag{10-24}$$

for a dielectric (i.e., polarizable) material with electric electric permittivity tensor $\underline{\kappa}$ with an applied electric field $\vec{E}$:

$$\frac{E}{V} = \frac{1}{2} \vec{E}^T \underline{\kappa} \vec{E} \tag{10-25}$$

The geometric interpretation of the quadratic forms is obtained by turning the above equations around and asking—what are the general vectors $\vec{x}$ for which the quadratic form (usually an energy or power density) has a particular value? Picking that particular value as unity, the question becomes what are the directions and magnitudes of $\vec{x}$ for which

$$1 = \vec{x}^T \underline{A} \vec{x} \tag{10-26}$$

This equation expresses a quadratic relationship between one component of $\vec{x}$ and the others. This is a surface—known as the *quadric surface* or *representation quadric*—which is an ellipsoid or hyperboloid sheet on which the quadratic form takes on the particular value 1.

In the principal axes (or, equivalently, the eigenbasis) the quadratic form takes the quadratic form takes the simple form:

$$e = \vec{x_{\text{eb}}}^T \underline{A_{\text{eb}}} \vec{x_{\text{eb}}} = A_{11}x_1^2 + A_{22}x_2^2 + A_{33}x_3^2 \tag{10-27}$$

and the representation quadric

$$A_{11}x_1^2 + A_{22}x_2^2 + A_{33}x_3^2 = 1 \tag{10-28}$$

which is easily characterized by the signs of the coefficients.

In other words, in the principal axis system (or the eigenbasis) the quadratic form has a particularly simple, in fact the most simple, form.

Eigenvector Basis

Among all similar matrices (defined by the similarity transformation defined by Eq. 10-13), the simplest matrix is the diagonal one. In the coordinate system where the similar matrix is diagonal, its diagonal entries are the eigenvalues. The question remains, "what is the coordinate transformation that takes the matrix into its diagonal form?"

The coordinate system is called the eigenbasis or principal axis system, and the transformation that takes it there is particularly simple.

The matrix that transforms from a general (old) coordinate system to a diagonalized matrix (in the new coordinate system) is the matrix of columns of the eigenvectors. The first column corresponds to the first eigenvalue on the diagonal matrix, and the n^{th} column is the eigenvector corresponding the n^{th} eigenvalue.

3.016 Home
Full Screen
Close
Quit

$$\begin{pmatrix} \text{The} \\ \text{Diagonalized} \\ \text{Matrix} \end{pmatrix} = \begin{pmatrix} \text{Eigenvector} \\ \text{Column} \\ \text{Matrix} \end{pmatrix}^{-1} \begin{pmatrix} \text{The} \\ \text{General} \\ \text{Matrix} \end{pmatrix} \begin{pmatrix} \text{Eigenvector} \\ \text{Column} \\ \text{Matrix} \end{pmatrix} \tag{10-29}$$

This method provides a method for finding the simplest quadratic form.

Lecture 11: Geometry and Calculus of Vectors

Reading:
Kreyszig Sections: 9.1, 9.2, 9.3, 9.4 (pages364–369, 371–374, 377–383, 384–388)

Vector Products

The concept of vectors as abstract objects representing a collection of data has already been presented. Every student at this point has already encountered vectors as representation of points, forces, and accelerations in two and three dimensions.

Review: The Inner (dot) product of two vectors and relation to projection

An inner- (or dot-) product is the multiplication of two vectors that produces a scalar.

$$\begin{aligned}\vec{a}\cdot\vec{b} \equiv & \\ \equiv & a_i b_i \\ \equiv & a_i b_j \delta_{ij} \text{ where } \delta_{ij} \equiv \begin{cases} 1 \text{ if } i=j \\ 0 \text{ otherwise} \end{cases} \\ \equiv & (a_1, a_2, \ldots a_N) \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_N \end{pmatrix} \\ \equiv & (b_1, b_2, \ldots b_N) \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{pmatrix}\end{aligned} \tag{11-1}$$

The inner product is:

linear, distributive $(k_1\vec{a}+k_2\vec{b})\cdot\vec{c} = k_1\vec{a}\cdot\vec{c}+k_2\vec{b}\cdot\vec{c}$

symmetric $\vec{a}\cdot\vec{b} = \vec{b}\cdot\vec{a}$

satisfies Schwarz inequality $\|\vec{a}\cdot\vec{b}\| \leq \|\vec{b}\|\|\vec{a}\|$

ratifies triangle inequality $\|\vec{a}+\vec{b}\| \leq \|\vec{b}\| + \|\vec{a}\|$

If the vector components are in a Cartesian (i.e., cubic lattice) space, then there is a useful equation for the angle between two vectors:

$$\cos\alpha = \frac{\vec{a}\cdot\vec{b}}{\|\vec{a}\|\|\vec{b}\|} = \hat{n_a}\cdot\hat{n_b} \tag{11-2}$$

where $\hat{n}_i$ is the unit vector that shares a direction with i. *Caution: when working with vectors in non-cubic crystal lattices (e.g, tetragonal, hexagonal, etc.) the angle relationship above does not hold. One must convert to a cubic system first to calculate the angles.*

The projection of a vector onto a direction $\hat{n}_b$ is a scalar:

$$p = \vec{a} \cdot \hat{n}_b \tag{11-3}$$

Review: Vector (or cross-) products

The vector product (or cross $\times$) differs from the dot (or inner) product in that multiplication produces a vector from two vectors. One might have quite a few choices about how to define such a product, but the following idea proves to be useful (and standard).

normal Which way should the product vector point? Because two vectors (usually) define a plane, the product vector might as well point away from it.

The exception is when the two vectors are linearly-dependent; in this case the product vector will have zero magnitude.

The product vector is normal to the plane defined by the two vectors that make up the product. A plane has two normals, but which normal should be picked? By convention, the "right-hand-rule" defines which of the two normals should be picked.

magnitude Given that the product vector points away from the two vectors that make up the product, what should be its magnitude? We already have a rule that gives us the cosine of the angle between two vectors, so a rule that gives the sine of the angle between the two vectors would be useful. Therefore, the cross product is defined so that its magnitude for two unit vectors is the sine of the angle between them.

This has the extra utility that the cross product is zero when two vectors are linearly-dependent (i.e., they do not define a plane).

This also has the utility, discussed below, that the triple product will be a scalar quantity equal to the volume of the parallelepiped defined by three vectors.

The triple product,

$$\begin{aligned} \vec{a} \cdot (\vec{b} \times \vec{c}) = (\vec{a} \times \vec{b}) \cdot \vec{c} = \\ \|\vec{a}\| \|\vec{b}\| \|\vec{c}\| \sin \gamma_{b-c} \cos \gamma_{a-bc} = \\ \|\vec{a}\| \|\vec{b}\| \|\vec{c}\| \sin \gamma_{a-b} \cos \gamma_{ab-c} \end{aligned} \tag{11-4}$$

where γ_{i-j} is the angle between two vectors i and j, and γ_{ij-k} is the angle between the vector k and the plane spanned by i and j, is equal to the parallelepiped that has $\vec{a}$, $\vec{b}$, and $\vec{c}$ emanating from its bottom-back corner.

If the triple product is zero, the volume between three vectors is zero and therefore they must be linearly dependent.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Cross Product Example

This is a simple demonstration of the vector product of two spatial vectors.

Here is the built-in cross product between two vectors

```
crossab = Cross[{a1, a2, a3} , {b1, b2, b3}]
```
1

And, here is the standard visual way to do it by hand with the determinant.

```
detab = Det[( i  j  k )
            ( a1 a2 a3 )
            ( b1 b2 b3 )]
```
2

Pick out each of the compenents to create the vector

```
testcrossab = {Coefficient[detab, i],
  Coefficient[detab, j],
  Coefficient[detab, k]}
```
3

Check for equality between the old-fashioned way and Mathematica's built-in function

```
testcrossab == crossab
```
4

1: `Cross` produces the vector product of two symbolic vectors $\vec{a}$ and $\vec{b}$ of length 3.

2: `Det` produces the same result using the memorization device:

$$\vec{a} \times \vec{b} = \det \begin{pmatrix} \hat{i} & \hat{j} & \hat{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{pmatrix}$$

3–4: `Coefficient` is used to extract each vector component and create a vector result, and then equality test the two vectors to show the equivalence.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualizing Space-Curves as Time-Dependent Vectors

Examples of $\vec{x}(t)$ and $d\vec{x}/dt$ are illustrated as curves and as animations.

Create a trajectory of a point or a particle

```
XVector[t_] :=
 {Cos[6 t], Sin[4 t], Sin[t] + Cos[t]}
```
1

ParametricPlot3D allows us to visualize the entire trajectory at once.

```
ParametricPlot3D[XVector[t],
 {t, 0, 2 π}, PlotStyle → {Thick, Blue},
 AxesLabel → {"x", "y", "z"}]
```
2

Here is a function to create a graphic with a variable end - point. We will have the function remember when it has already computed a graphic, trading memory for a possible speed-up.

```
paraplot[time_] := paraplot[time] =
  ParametricPlot3D[XVector[t],
   {t, 0, time}, PlotStyle → {Thick, Blue},
   AxesLabel → {"x", "y", "z"}]
```
3

Use manipulate on the graphics function to visualize how the curve develops with its parameter

```
Manipulate[paraplot[time],
 {{time, 0.05}, 0.01, 2 π}]
```
4

However, we need to fix the length scale between frames, so we use the last graphic to infer what PlotRange should be.

```
Manipulate[Show[paraplot[time],
   PlotRange → {{-1, 1}, {-1, 1}, {-1.5, 1.5}}],
 {{time, 0.05}, 0.01, 2 π}]
```
5

Next, we add a graphic element to show the vector, drawn from the origin, for each end-point.

```
Manipulate[
 Show[{paraplot[time], Graphics3D[{Cylinder[
      {{0, 0, 0}, XVector[time]}, 0.03]}]},
  PlotRange → {{-1, 1}, {-1, 1}, {-1.5, 1.5}}],
 {{time, 0.05}, 0.01, 2 π}]
```
6

1: A list of three time-dependent components for (x, y, z) is constructed as the function $XVector$.

2: `ParametricPlot3D` takes a three-component vector as an argument and then will plot the evolution of the vector as a function of a parameter.

3: To visualize the evolution of the curve, it is useful to plot the resulting trajectory. We create a function that constructs a plot up to a variable end-point, `time`, which appears as the upper-bound to `ParametricPlot3D`.

4: `Manipulate` provides an interactive animation of the curve's development.

5: However, it is better if the length scale is fixed. We set `PlotRange` from visual inspection of the last frame of the previous example.

6: To improve the visualization, we add a `Cylinder` graphics primitive to illustrate the vector drawn from the origin.

3.016 Home

Full Screen

Close

Quit

Derivatives of Vectors

Consider a vector, $\vec{p}$, as a point in space. If that vector is a function of a real continuous parameter, for instance, t, then $\vec{p}(t)$ represents the loci as a function of a parameter.

If $\vec{p}(t)$ is continuous, then it sweeps out a continuous curve as t changes continuously. It is very natural to think of t as time and $\vec{p}(t)$ as the trajectory of a particle—such a trajectory would be continuous if the particle does not disappear at one instant, t, and then reappear an instant later, $t + dt$, some finite distance distance away from $\vec{p}(t)$.

If $\vec{p}(t)$ is continuous, then the limit is:

$$\frac{d\vec{p}(t)}{dt} = \lim_{\Delta t \to 0} \frac{\vec{p}(t + \Delta t) - \vec{p}(t)}{\Delta t} \tag{11-5}$$

Notice that the numerator inside the limit is a vector and the denominator is a scalar; so, the derivative is also a vector. Think about the equation geometrically—it should be apparent that the vector represented by the derivative is locally tangent to the curve that is traced out by the points $\vec{p}(t - dt)$, $\vec{p}(t)$ $\vec{p}(t + dt)$, etc.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualizing Time-Dependent Vectors and their Derivatives

Examples of $\vec{x}(t)$ and $d\vec{x}/dt$ are illustrated as curves and as animations.

The local derivative of the vector that we visualized above: $\vec{v} = \frac{d\vec{x}}{dt}$

```
Simplify[D[XVector[t], t]]
```
1

Write out a function for the derivative:

```
dxdt[s_] :=
 {-6 Sin[6 s], 4 Cos[4 s], Cos[s] - Sin[s]}
```
2

```
dxdtplot[time_] :=
 ParametricPlot3D[dxdt[t], {t, 0, time},
  PlotStyle → {Thick, Darker[Red]},
  AxesLabel → {"x", "y", "z"}]
dxdtplot[2 π]
```
3

```
Manipulate[
 Show[paraplot[time], dxdtplot[time],
  Graphics3D[{
    {Lighter[Blue], Cylinder[
      {{0, 0, 0}, XVector[time]}, 0.1]},
    {Lighter[Red], Cylinder[
      {{0, 0, 0}, dxdt[time]}, 0.1]}
   }], PlotRange → {{-6, 6}, {-4, 4},
   {-1.5, 1.5}}], {{time, 0.05}, 0.01, 2 π}]
```
4

To visualize the "tangency property," we translate the derivative-vector to the end of the space curve

```
Manipulate[
 Show[paraplot[time], dxdtplot[time],
  Graphics3D[{
    {Lighter[Blue], Cylinder[
      {{0, 0, 0}, XVector[time]}, 0.1]},
    {Lighter[Red], Translate[
      Cylinder[{{0, 0, 0}, dxdt[time]}, 0.1],
      XVector[time]]}
   }], PlotRange → {{-6, 6}, {-6, 6},
   {-4, 4}}], {{time, 0.05}, 0.01, 2 π}]
```
5

1: The derivative operator `D` is a *threadable function* so it will operate on each component of its vector argument; thus, we can obtain the vector derivative by operating on the entire vector.

2: The derivative-vector $d\vec{x}/dt$ is encoded as a function.

3: The derivative of the space-curve (from the above example) is visualized with a function that calls `ParameticPlot3D`. Because the space curve is differentiable and periodic, its derivative should be periodic as well, but it *appears to not be periodic.*

4: Using `Manipulate` reveals that the function is periodic.

5: Here, we repeat the interactive example, but use `Translate` to move the visualized derivative-vector to the end of the curve. This will give a visual demonstration of the tangent behavior of the derivative.

3.016 Home

Full Screen

Close

Quit

Review: Partial and total derivatives

One might also consider that a time- and space-dependent vector field, for instance $\vec{E}(x,y,z,t) = \vec{E}(\vec{x},t)$ could be the force on a unit charge located at $\vec{x}$ and at time t.

Here, there are many different things which might be varied and which give rise to a derivative. Such questions might be:

1. How does the force on a unit charge differ for two nearby unit-charge particles, say at (x,y,z) and at $(x,y+\Delta y,z)$?
2. How does the force on a unit charge located at (x,y,z) vary with time?
3. How does the force on a particle change as the particle traverses some path $(x(t),y(t),z(t))$ in space?

Each question has the "flavor" of a derivative, but each is asking a different question. So a different kind of derivative should exist for each type of question.

The first two questions are of the nature, "How does a quantity change if only one of its variables changes and the others are held fixed?" The kind of derivative that applies is the partial derivative.

The last question is of the nature "How does a quantity change when all of its variables depend on a single variable?" The kind of derivative that applies is the total derivative. The answers are:

1.
$$\frac{\partial \vec{E}(x,y,z,t)}{\partial y} = \left(\frac{\partial \vec{E}}{\partial y}\right)_{\text{constant}x,z,t} \tag{11-6}$$

2.
$$\frac{\partial \vec{E}(x,y,z,t)}{\partial t} = \left(\frac{\partial \vec{E}}{\partial t}\right)_{\text{constant}x,y,z} \tag{11-7}$$

3.

$$\frac{d\vec{E}(x(t),y(t),z(t),t)}{dt} = \frac{\partial \vec{E}}{\partial x}\frac{dx}{dt} + \frac{\partial \vec{E}}{\partial y}\frac{dy}{dt} + \frac{\partial \vec{E}}{\partial z}\frac{dz}{dt} + \frac{\partial \vec{E}}{\partial t}\frac{dt}{dt} = \nabla \vec{E}(\vec{x}(t),t) \cdot \frac{d\vec{x}}{dt} + \frac{\partial \vec{E}}{\partial t} \tag{11-8}$$

Time-Dependent Scalar and Vector Fields

A physical quantity that is spatially variable is often called a *spatial field*. It is a particular case of a field quantity.

Such fields can be simple scalars, such as the altitude as a function of east and west in a topographical map. Vectors can also be field quantities, such as the direction uphill and steepness on a topographical map— this is an example of how each scalar field is naturally associated with its *gradient field*. Higher dimensional objects, such as stress and strain, can also be field quantities.

Fields that evolve in time are *time-dependent fields* and appear frequently in physical models. Because time-dependent 3D spatial fields are four-dimensional objects, animation is frequently used to visualize them.

For a working example, consider the time-evolution of "ink concentration" $c(x,y,t)$ of a very small spot of ink spilled on absorbent paper at $x = y = 0$ and at time $t = 0$. This example could be modeled with Fick's first law:

$$\vec{J} = -D\nabla c(x,y,t) = -D\left(\frac{\partial c}{\partial x} + \frac{\partial c}{\partial y}\right) \tag{11-9}$$

where D is the diffusivity that determines "how fast" the ink moves for a given gradient ∇c, and $\vec{J}$ is a time-dependent vector that represents "rate of ink flow past a unit-length line segment oriented perpendicular to $\vec{J}$. This leads to the two-dimensional diffusion equation

$$\frac{\partial c}{\partial t} = D\left(\frac{\partial^2 c}{\partial x^2} + \frac{\partial^2 c}{\partial y^2}\right) \tag{11-10}$$

For this example, the solution, $c(x,y,t)$ is given by

$$c(x,y,t) = \frac{c_o}{4\pi D t} e^{-\frac{x^2+y^2}{4Dt}} \tag{11-11}$$

where c_o is the initial concentration of ink.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualizing a Solution to the Diffusion Equation

The solution to a 2D diffusion equation in the infinite plane with rectangular initial conditions $c(-a/2 < x < a/2, -b/2 < y < b/2, t = 0) = 1$ and $c(|x| > a/2, |y| > b/2, t = 0) = 0$ is visualized and will serve as an example of the flux (or time-dependent gradient) field in the following example.

```
concentration =
 Integrate[Exp[-((xsource-x)^2 + (ysource-y)^2)/(4 Diffusivity t)]/(4 Pi Diffusivity t),
  {xsource, -a / 2, a / 2},
  {ysource, -b / 2, b / 2},
  Assumptions → Diffusivity > 0 && t > 0 &&
    a > 0 && b > 0 && x ∈ Reals && y ∈ Reals]
```
1

```
aspectRatio = 3;
b = aspectRatio a;
length = a;
time = length^2 / Diffusivity;
ScaleRules =
  {t -> τ time, x -> ξ length, y -> η length};
```
2

```
scaledconc =
 Simplify[concentration /. ScaleRules,
  Assumptions → a > 0]
```
3

```
Plot3D[scaledconc /. τ → 0.003,
 {ξ, -3, 3}, {η, -3, 3}, PlotRange → {0, 1},
 MeshFunctions → {#3 &}, PlotPoints → 30,
 Mesh → 5, MeshStyle → {Thick}]
```
4

```
Manipulate[Plot3D[scaledconc /. τ → timevar,
  {ξ, -3, 3}, {η, -3, 3},
  PlotRange → {0, 1}, MaxRecursion → 4],
 {{timevar, 0.05}, 0.001, 0.1}]
```
5

```
cplots =
  Table[ContourPlot[scaledconc /. τ → timevar,
    {ξ, -3, 3}, {η, -3, 3},
    PlotRange → {0, 1}, ColorFunction →
     ColorData["TemperatureMap"]],
   {timevar, .001, .2, .005}];
```
6

```
ListAnimate[cplots]
```
7

1: This is the solution to the stated problem. It is obtained by integrating the appropriate *Green's function* which will be discussed in a later lecture. We use physical `Assumptions` in the call to `Integrate`.

2: We set a model parameter (`aspectRatio`) for the shape of the initial rectangle, and then define a characteristic length and time in terms of quantities that appear in the model. A set of rules are defined, *ScaleRules* , that can be applied to the solution to create a *non-dimensional model* the problem.

3: The re-scaled solution, *scaledconc* , depends only on non-dimensional quantities, ξ, η, and τ.

4: Here is an example of the solution at $\tau = 0.003$. We use the `MeshFunctions` option to `Plot3D` to draw the five isoconcentration lines on the surface.

5: This will produce an interactive animation of the solution. However, because the evaluation of each animation-frame is likely to be slow, this visualization will be sluggish on many computers.

6: A simpler graphical representation is obtained with `ContourPlot` by plotting contours of constant concentration. We pre-compute a table of plots and store the result.

7: The resulting animation is created from two-dimensional objects using `ListAnimate` on the pre-computed frames.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualizing the Diffusion Flux: The Time-Dependent Gradient Field

The time-dependent gradient field for the 2D diffusion equation treated above is visualized.

```
flux = {-D[scaledconc, ξ], -D[scaledconc, η]}                    1
```

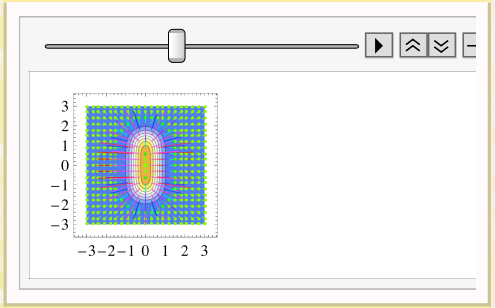
```
Needs["VectorFieldPlots`"];                                      2
```

```
VectorFieldPlot[flux /. τ → .08,
 {ξ, -3, 3}, {η, -3, 3}, PlotPoints → 21,                        3
 ColorFunction → ColorData["DarkRainbow"]]
```

```
vplots =
  Table[VectorFieldPlot[flux /. τ → time,
    {ξ, -3, 3}, {η, -3, 3}, PlotPoints → 21,
    PlotRange → {{-4, 4}, {-4, 4}},
    Frame → True, MaxArrowLength → 3,                            4
    ScaleFactor → None,
    ScaleFunction → (2 # &),
    ColorFunction → (Hue[0.25 + 0.75 #] &)],
   {time, .001, .2, .005}];
```

```
ListAnimate[vplots]                                              5
```

```
ListAnimate[
 Table[Show[cplots[[i]], vplots[[i]],
   PlotRange → 3.5 {{-1, 1}, {-1, 1}},                           6
   ImageSize → {72, 72}],
  {i, 1, Length[cplots]}]]
```



1: The flux, $\vec{J}$, is the rate of matter-flow through an oriented plane per unit area. Fick's first law related the flux to the gradient of the concentration field, $\vec{J} = -D\nabla c$. Here, we compute a non-dimensionalized form of flux.

2: To plot the flux which is a vector field, the package `VectorFieldPlots` is loaded so that we have access to its `PlotVectorField` function. For example, here is an example of (minus) the gradient field at $\tau = 0.08$. The vectors are plotted on a square mesh and we color them according to their magnitude with `ColorFunction` and `ColorData`.

5–6: We precompute a series of frames and animate them. To ensure that the vector length scales are consistent between frames, the option `ScaleFactor` is used.

7: Here, we visualize time-dependent vector flux and the concentration together by using `Show` on pre-computed graphics.

3.016 Home
Full Screen
Close
Quit

All vectors are not spatial

It is useful to think of vectors as spatial objects when learning about them—but one shouldn't get stuck with the idea that all vectors are points in two- or three-dimensional space. The spatial vectors serve as a good analogy to generalize an idea.

For example, consider the following chemical reaction:

Reaction:	H_2	$\frac{1}{2}O_2$	$\rightleftharpoons$	H_2O
Initial:	1	1	$\rightleftharpoons$	0
During Rx.:	$1-\xi$	$1-\frac{1}{2}\xi$	$\rightleftharpoons$	ξ

The composition could be written as a vector:

$$\vec{N} = \begin{pmatrix} \text{moles } H_2 \\ \text{moles } O_2 \\ \text{moles } H_2O \end{pmatrix} = \begin{pmatrix} 1-\xi \\ 1-\frac{1}{2}\xi \\ \xi \end{pmatrix} \tag{11-12}$$

and the variable ξ plays the role of the "extent" of the reaction—so the composition variable $\vec{N}$ lives in a reaction-extent (ξ) space of chemical species.

Lecture 12: Multivariable Calculus

Reading:
Kreyszig Sections: 9.5, 9.6, 9.7 (pages389–398, 400–403, 403–409)

The Calculus of Curves

In the last lecture, the derivatives of a vector that varied continuously with a parameter, $\vec{r}(t)$, were considered. It is natural to think of $\vec{r}(t)$ as a curve in whatever space the vector $\vec{r}$ is defined. The most familiar example is a curve in the plane: the two values $(x(t), y(t))$ are mapped onto the plane through values as t sweeps through its range $t_{\text{initial}} \leq t \leq t_{\text{final}}$. A curve in three-dimensional Cartesian space is the mapping of three values $(x(t), y(t), z(t))$; and in cylindrical coordinates it is: $(r(t), \theta(t), z(t))$. In general, a curve is represented by N coordinates, as a *single parameter* (i.e., t) takes on a range of numbers—the N coordinates form the embedding space.

Objects that have more dimensions than curves need more parameters. The number of parameters is the dimensionality of the object and the number of coordinates is the dimensionality of the embedding space. What we naturally call a *surface* is a two dimensional object embedded in a three-dimensional space—for example, in Cartesian coordinates $(x(u, v), y(u, v), z(u, v))$ is a surface.

The two-dimensional surface $(x(u, v), y(u, v), z(u, v))$ can itself become an embedding space for lower dimensional objects; for example, the curve $(u(t), v(t))$ is embedded in the surface (u, v) which itself is embedded in (x, y, z). In other words, the curve $(x(u(t), v(t)), y(u(t), v(t)), z(u(t), v(t)))$ can be considered to be embedded in (u, v), or embedded in (x, y, z) and constrained to the surface $(x(u, v), y(u, v), z(u, v))$.

In higher dimensions, there are many more possibilities and we can make a few introductory remarks about the language that is used to describe them. For application to physical problems, these considerations indicate the number of degrees-of-freedom

that are available and the conditions that a system is over-constrained. An N-dimensional surface (sometimes called a *hyper-surface*) embedded in an M-dimensional space is said to have *codimension* $M - N$. Some objects cannot be embedded in a higher dimensional space; these are called *non-embeddable*, and examples include the Klein bottle which cannot be embedded in our three-dimensional space.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Embedding Curves in Surfaces in Three Dimensions

An example is constructed that visualizes a two-dimensional surface in three dimensions and then visualizes a one-dimensional curve constrained to that surface.

Create an example function that returns a position {x, y, z} as a function of two parameters

```
FlowerPot[u_, v_] := {(3 + Cos[v]) Cos[u],
  Sin[u] + (3 + Cos[v]) Sin[u],
  (3 / 2 + Cos[u + v]) Sin[v]}
```
1

Visualize it.

```
Flowerplot = ParametricPlot3D[
  FlowerPot[u, v], {u, 0, 2 Pi},
  {v, 0, 2 Pi}, PlotPoints -> {120, 40},
  PlotStyle → Directive[Brown, Opacity[0.6],
    Specularity[White, 40]], Mesh → None]
```
2

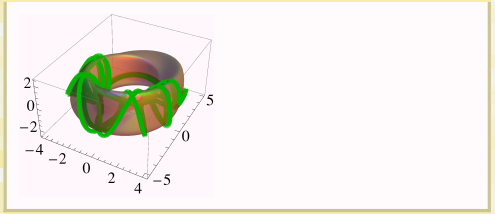
A Curve on a parameritized surface

*Now, we call the function again, but make the two parameters {u,v}, depend on a single parameter t (*note when visualizing this curve, it has been scaled in and out a little so it will be visible in subsequent visualizations*)*

```
Vines[t_] := FlowerPot[t Cos[t], -t^2 Sin[ t]]
vineplot = ParametricPlot3D[
  {1.05 * Vines[t], 0.95 * Vines[t]},
  {t, 0, 2 Pi}, PlotStyle →
   {{Thickness[0.025], Darker[Green]},
    {Thickness[0.025], Darker[Green]}},
  PlotRange → All]
```
3

This is the paramertized surface with a curve embedded in the surface.

```
Show[vineplot, Flowerplot]
```
4



1: *FlowerPot* takes two arguments and returns a vector. As the arguments sweep through domains, the vector will trace out a surface.

2: Using the `ParametricPlot3D`, the surface is visualized. Here we use `Directive` to make the surface brown with `Opacity` to make the surface about 40% transparent. `Specularity` is used to make the surface text look a bit shiny here. The resulting graphics are assigned to the symbol `Flowerplot`.

3: *Vines* takes a single argument and then calls *FlowerPot* with two arguments that are functions of that single argument—the result must be a curve embedded in the surface. In this case, the function is repeated and is scaled in and out a little, so the curves will be visible later.

4: Here, both the embedded curve and the surface are shown together.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Because the derivative of a curve with respect to its parameter is a tangent vector, the unit tangent can be defined immediately:

$$\hat{u} = \frac{\frac{d\vec{r}}{dt}}{\|\frac{d\vec{r}}{dt}\|} \tag{12-1}$$

It is convenient to find a new parameter, $s(t)$, that would make the denominator in Eq. 12-1 equal to one. This parameter, $s(t)$, is the arc-length:

$$\begin{aligned} s(t) &= \int_{t_o}^{t} ds \\ &= \int_{t_o}^{t} \sqrt{dx^2 + dy^2 + dz^2} \\ &= \int_{t_o}^{t} \sqrt{(\frac{dx}{dt})^2 + (\frac{dy}{dt})^2 + (\frac{dz}{dt})^2}dt \\ &= \int_{t_o}^{t} \sqrt{(\frac{d\vec{r}}{dt}) \cdot (\frac{d\vec{r}}{dt})}dt \end{aligned} \tag{12-2}$$

and with s instead of t,

$$\hat{u}(s) = \frac{d\vec{r}}{ds} \tag{12-3}$$

This is natural because $\|\vec{r}\|$ and s have the same units (i.e., meters and meters, foots and feet, etc) instead of, for instance, time, t, that makes $d\vec{r}/dt$ a velocity and involves two different kinds of units (e.g., furlongs and hours).

With the arc-length s, the magnitude of the curvature is particularly simple,

$$\kappa(s) = \|\frac{d\hat{u}}{ds}\| = \|\frac{d^2\vec{r}}{ds^2}\| \tag{12-4}$$

as is its interpretation—the curvature is a measure of how rapidly the unit tangent is changing direction.

Furthermore, the rate at which the unit tangent changes direction is a vector that must be normal to the tangent (because $d(\hat{u} \cdot \hat{u} = 1) = 0$) and therefore the unit normal is defined by:

$$\hat{p}(s) = \frac{1}{\kappa(s)}\frac{d\hat{u}}{ds} \tag{12-5}$$

There are two unit vectors that are locally normal to the unit tangent vector $\hat{u}'(s)$ and the curve unit normal $\hat{p}(s) \times \hat{u}$ and $\hat{u}(s) \times \hat{p}$. This last choice is called the unit binormal, $\hat{b} \equiv \hat{u}(s) \times \hat{p}$ and the three vectors together form a nice little moving orthogonal axis pinned to the curve. Furthermore, there are convenient relations between the vectors and differential geometric quantities called the Frenet equations.

Using Arc-Length as a Curve's Parameter

However, it should be pointed out that—although re-parameterizing a curve in terms of its arc-length makes for simple analysis of a curve—achieving this re-parameterization is not necessarily simple.

Consider the steps required to represent a curve $\vec{r}(t)$ in terms of its arc-length:

integration The integral in Eq. 12-2 may or may not have a closed form for $s(t)$.

If it does, then we can perform the following operation:

inversion $s(t)$ is typically a complicated function that is not easy to invert, i.e., solve for t in terms of s to get a $t(s)$ that can be substituted into $\vec{r}(t(s)) = \vec{r}(s)$.

These difficulties usually result in treating the problem symbolically and resorting to numerical methods of achieving the integration and inversion steps.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Calculating arclength

Examples of computing a curve's arc-length s from the relation $ds = |d\vec{x}| = (dx^2 + dy^2 + dz^2)$ are presented.

Make up two functions that will illustrate the difference between a curve's parameter and its arclength

```
PrettyFlower[t_] := (1/4 + 3/4 Cos[3 t])
  {Cos[t]^3, Sin[t]^3, Sin[t] Cos[t]^2}
```
1

```
Bendy[t_] := {Cos[t], Sin[t], Sin[t] Cos[t]}
```
2

Here is a general way to take a function of a general parameter, t, and compute the arc length traversed as t varies from one value to another:

```
dFlowerDt = Simplify[D[PrettyFlower[t], t]]
```
3

This is the arclength up to the parameter t, the integral does not have a closed-form

```
sFlower = Integrate[
  Sqrt[Simplify[dFlowerDt.dFlowerDt]], t]
```
4

In other words,
$ds^2 = dx^2 + dy^2 + dz^2$ so integrating the square root of this is the arclength

Applying this to the function Bendy defined above:

```
dBendyDt = D[Bendy[t], t]
```
5

This is the arclength up to the parameter t, the integral does have a closed-form, but is not easily invertible.

The arc length in this case is given by a tabulated function called an elliptic integral and after checking its behavior at t = 0 we can plot it over the range {t,0,2π}:

```
sBendy /. t → 0
```
6

However, the inverse exits, we can find a t(s) (the curve parameter t for any arclength s)

```
Plot[sBendy, {t, 0, 2 Pi}]
```
7

Alternatively, we can evaluate the expression for arc length numerically using the following:

```
Plot[Evaluate[
  NIntegrate[Sqrt[dFlowerDt.dFlowerDt],
   {t, 0, uplim}]], {uplim, 0, 6.4}]
```
8

1–2: Two example functions of a single parameter t that return a vector $\vec{x}$ are defined for this example.

3: Here, the tangent-vector for the function, *PrettyFlower* defined above, is computed.

4: This is an attempt to find a closed-form solution for arclength $s(\tau) - s(0) = \int_0^\tau \sqrt{\left(\frac{dc}{dt}\right)^2} dt$. A closed-form solution doesn't exist.

5: However, a closed-form solution does exist for the arclength of the *Bendy* function defined earlier. If the closed-form $s(t)$ could be inverted (i.e., $t(s)$) then the curve $c(t)$ could be expressed in terms of its natural variable $c(s) = c(t(s))$.

6–7: The plot, $s(t)$, is monotonically increasing, and therefore the function could always be inverted numerically.

8: Even for the arc-length that could not be evaluated in closed-form (i.e., *PrettyFlower*), a numerical integration could be used to perform the inversion.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Scalar Functions with Vector Argument

In Materials Science and Engineering, the concept of a spatially varying function arises frequently:

For example:

Concentration $c_i(x,y,z) = c_i(\vec{x})$ is the number (or moles) of chemical species of type i *per unit volume* located at the point $\vec{x}$.

Density $\rho(x,y,z) = \rho(\vec{x})$ mass *per unit volume* located at the point $\vec{x}$ is $\rho(x,y,z) = \rho(\vec{x})$.

Energy Density $u(x,y,z) = u(\vec{x})$ energy *per unit volume* located at the point $\vec{x}$.

The examples above are spatially-dependent densities of "extensive quantities."

There are also spatially variable intensive quantities:

Temperature $T(x,y,z) = T(\vec{x})$ is the temperature which would be measured at the point $\vec{x}$.

Pressure $P(x,y,z) = P(\vec{x})$ is the pressure which would be measured at the point $\vec{x}$.

Chemical Potential $\mu_i(x,y,z) = \mu_i(\vec{x})$ is the chemical potential of the species i which would be measured at the point $\vec{x}$.

Each example is a scalar function of space—that is, the function associates a scalar with each point in space.

A topographical map is a familiar example of a graphical illustration of a scalar function (altitude) as a function of location (latitude and longitude).

However, there are many other types of scalar functions of several arguments, such as the state function: $U = U(S, V, N_i)$ or $P = P(V, T, N_i)$. It is sometimes useful to think of these types of functions a scalar functions of a "point" in a thermodynamics space.

However, this is often a source of confusion: notice that the internal energy is used in two different contexts above. One context is the value of the energy, say 128.2 Joules. The other context is the function $U(S, V, N_i)$. While the two symbols are identical, their meanings are quite different.

The root of the confusion lurks in the question, "What are the variables of U?" Suppose that there is only one (independent) chemical species, then $U(\cdot)$ has three variables, such as $U(S, V, N)$. "But if $S(T, P, \mu)$, $V(T, P, \mu)$, and $N(T, P, \mu)$ are known functions, then what are the variables of U?" The answer is that they are any three independent variables, one could write $U(T, P, \mu) = U(S(T, P, \mu), V(T, P, \mu), N(T, P, \mu))$, and there are six other natural choices.

The trouble arises when variations of a function like U are queried—then the variables that are varying must be specified.

For this reason, it is either a good idea to keep the functional form explicit in thermodynamics, i.e.,

$$\begin{aligned} dU(S,V,N) &= \frac{\partial U(S,V,N)}{\partial S}dS + \frac{\partial U(S,V,N)}{\partial V}dV + \frac{\partial U(S,V,N)}{\partial N}dN \\ dU(T,P,\mu) &= \frac{\partial U(T,P,\mu)}{\partial T}dT + \frac{\partial U(T,P,\mu)}{\partial V}dV + \frac{\partial U(T,P,\mu)}{\partial \mu}d\mu \end{aligned} \tag{12-6}$$

or use, the common thermodynamic notation,

$$\begin{aligned} dU &= \left(\frac{\partial U}{\partial S}\right)_{V,N} dS + \left(\frac{\partial U}{\partial V}\right)_{S,N} dV + \left(\frac{\partial U}{\partial N}\right)_{S,V} dN \\ dU &= \left(\frac{\partial U}{\partial T}\right)_{P,\mu} dT + \left(\frac{\partial U}{\partial P}\right)_{T,\mu} dP + \left(\frac{\partial U}{\partial \mu}\right)_{T,P} d\mu \end{aligned} \tag{12-7}$$

Total and Partial Derivatives, Chain Rule

There is no doubt that a great deal confusion arises from the following question, "What are the variables of my function?"

For example, suppose we have a three-dimensional space (x, y, z), in which there is an embedded surface $(x(w, v), y(w, v), z(w, v))$ $\vec{x}(w, v) = \vec{x}(\vec{u})$ where $\vec{u} = (v, w)$ is a vector that lies in the surface, and an embedded curve $(x(s), y(s), z(s)) = \vec{x}(s)$. Furthermore, suppose there is a curve that lies within the surface $(w(t), v(t)) = \vec{u}(t)$.

Suppose that $\mathcal{E} = f(x, y, z)$ is a scalar function of (x, y, z).

Here are some questions that arise in different applications:

1. How does $\mathcal{E}$ vary as a function of position?
2. How does $\mathcal{E}$ vary along the surface?
3. How does $\mathcal{E}$ vary along the curve?
4. How does $\mathcal{E}$ vary along the curve embedded in the surface?

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Total Derivatives and Partial Derivatives: A Mathematica Review

Demonstrations of 1) the three spatial derivatives of $F(x, y, z)$; 2) the two independent derivatives on a two-dimensional surface embedded in x–y–z; 3) the complete derivative of $F(x, y, z)$ along a curve $(x(t), y(t), z(t))$.

AScalarFunction is defined everywhere in (x,y,z)

```
AScalarFunction[x_ , y_ , z_] :=
 SomeFunction[x, y, z]
```
1

```
AScalarFunction[x, y, z]
```
2

The following lines print and they define expressions.

```
dFuncX = D[AScalarFunction[x, y, z], x]
dFuncY = D[AScalarFunction[x, y, z], y]
dFuncZ = D[AScalarFunction[x, y, z], z]
```
3

x(w,v), y(w,v), z(w,v) is a restriction of all space to a surface parameterized by (w,v),
AScalarFunction is now defined on the surface as a function of (w,v)

```
AScalarFunction[x[w, v], y[w, v], z[w, v]]
```
4

Because it is now a function of w and v, the derivative with respect to x will vanish:

```
D[AScalarFunction[
  x[w, v], y[w, v], z[w, v]], x]
```
5

Two more flavors of derivatives, these are partial derivatives evaluated on the surface

```
dFuncW = D[AScalarFunction[
   x[w, v], y[w, v], z[w, v]], w]
```
6

```
dFuncV = D[AScalarFunction[
   x[w, v], y[w, v], z[w, v]], v]
```
7

On the surface x(w,v), y(w,v), z(w,v), we can prescribe a curve w(t), v(t), .now we have AScalarFunction defined on that curve

```
AScalarFunction[x[w[t], v[t]],
 y[w[t], v[t]], z[w[t], v[t]]]
```
8

The following is a derivative of the function along the curve parameterized by t

```
dFuncT = D[AScalarFunction[x[w[t], v[t]],
   y[w[t], v[t]], z[w[t], v[t]]], t]
```
9

```
dFuncT =
 D[AScalarFunction[x[t], y[t], z[t]], t]
```
10

1–2: *AScalarFunction* is a symbolic representation of a function—it will be a place-holder for examples of partial derivatives.

3: This will print Mathematica's representation of derivatives with respect to one of several arguments—e.g., $\partial F(x, y, z)/\partial y$ is written as $\mathtt{F}^{(0,1,0)}[\mathrm{x,y,z}]$.

4: `AScalarFunction` becomes a function of two variables when x, y, and z are restricted to a surface parameterized by (u, v): $(x(w, v), y(w, v), z(w, v))$

5–6: Caution: the distinction between the symbol `x` and the symbol `x[w,v]` is important; the following two examples show how the derivatives should appear.

7: This and the previous example show how the chain rule is computed, these two terms are the components of the gradient in the surface.

8: In this case, the previous `AScalarFunction` becomes a function of a single variable by specifying a curve in the surface with $(w(t), v(t))$.

9: Now, a total derivative of the curve embedded in the surface can be calculated with the chain rule. This is nearly equivalent to the following along a specified curve.

10: Here is the total derivative along a specific curve $(x(t), y(t), z(t)$, but here there is no constraint to the surface $\vec{x}(w, v)$

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

The behavior of a function near a point is something that arises frequently in physical models. When the function has lower-order continuous partial derivatives (generally, a "smooth" function near the point in question), the stock method to model local behavior is Taylor's series expansions around a fixed point.

Taylor's expansion for a scalar function of n variables, $f(x_1, x_2, \ldots, x_n)$, which has continuous first and second partial derivatives near the point $\vec{\xi} = (\xi_1, \xi_2, \ldots, \xi_n)$, is:

$$
\begin{aligned}
& f(\xi_1, \xi_2, \ldots, \xi_n) = f(x_1, x_2, \ldots, x_n) \\
& + \left.\frac{\partial f}{\partial x_1}\right|_{\vec{\xi}} (\xi_1 - x_1) + \left.\frac{\partial f}{\partial x_2}\right|_{\vec{\xi}} (\xi_2 - x_2) + \ldots + \left.\frac{\partial f}{\partial x_n}\right|_{\vec{\xi}} (\xi_n - x_n) \\
+ \frac{1}{2} [& \\
& \left.\frac{\partial^2 f}{\partial {x_1}^2}\right|_{\vec{\xi}} (\xi_1 - x_1)^2) + \left.\frac{\partial^2 f}{\partial x_1 \partial x_2}\right|_{\vec{\xi}} (\xi_1 - x_1)(\xi_2 - x_2) + \ldots + \left.\frac{\partial^2 f}{\partial x_1 \partial x_n}\right|_{\vec{\xi}} (\xi_1 - x_1)(\xi_n - x_n) \\
& + \left.\frac{\partial^2 f}{\partial x_2 \partial x_1}\right|_{\vec{\xi}} (\xi_2 - x_2)(\xi_1 - x_1) + \left.\frac{\partial^2 f}{\partial {x_2}^2}\right|_{\vec{\xi}} (\xi_2 - x_2)^2 + \ldots + \left.\frac{\partial^2 f}{\partial x_2 \partial x_n}\right|_{\vec{\xi}} (\xi_2 - x_2)(\xi_n - x_n) \\
& \vdots \qquad\qquad \ldots \qquad\qquad \vdots \\
& + \left.\frac{\partial^2 f}{\partial x_n \partial x_1}\right|_{\vec{\xi}} (\xi_n - x_n)(\xi_1 - x_1) + \left.\frac{\partial^2 f}{\partial x_n \partial x_2}\right|_{\vec{\xi}} (\xi_n - x_n)(\xi_2 - x_2) + \ldots + \left.\frac{\partial^2 f}{\partial {x_n}^2}\right|_{\vec{\xi}} (\xi_n - x_n)^2 \\
] & \\
& + \mathcal{O}\left[(\xi_1 - x_1)^3\right] + \mathcal{O}\left[(\xi_1 - x_1)^2 (\xi_2 - x_2)\right] + \mathcal{O}\left[(\xi_1 - x_1)(\xi_2 - x_2)^2\right] + \mathcal{O}\left[(\xi_2 - x_2)^3\right] \\
& + \ldots + \mathcal{O}\left[(\xi_1 - x_1)^2 (\xi_n - x_n)\right] + \mathcal{O}\left[(\xi_1 - x_1)(\xi_2 - x_2)(\xi_n - x_n)\right] + \ldots + \mathcal{O}\left[(\xi_n - x_n)^3\right]
\end{aligned}
\tag{12-8}
$$

or in a vector shorthand:

$$
f(\vec{x}) = f(\vec{\xi}) + \nabla_{\vec{x}} f|_{\vec{\xi}} \cdot (\vec{\xi} - \vec{x}) + (\vec{\xi} - \vec{x}) \cdot (\nabla_{\vec{x}} \nabla_{\vec{x}} f)|_{\vec{\xi}} \cdot (\vec{xi} - \vec{x}) + \mathcal{O}\left[\|\vec{\xi} - \vec{x}\|^3\right] \tag{12-9}
$$

In the following example, visualization of local approximations will be obtained for a scalar function of two variables, $f(x,y)$. This will be extended into an approximating function of four variables by expanding it about a point (ξ,η) to second order. The expansion is now a function of four variables—the first two variables are the point the function is expanded around (x and y), and the second two are the variable of the parabolic approximation at that point (ξ and η): $f_{\text{appx}}(\xi,\eta;x,y) = f(x,y) + \left.\frac{\partial f}{\partial x}\right|_{x,y}(\xi-x) + \left.\frac{\partial f}{\partial y}\right|_{x,y}(\eta-y) + \mathcal{Q}$ where $\mathcal{Q} \equiv \frac{1}{2}\left.\frac{\partial^2 f}{\partial x^2}\right|_{x,y}(\xi-x)(\xi-x) + \left.\frac{\partial^2 f}{\partial x \partial y}\right|_{x,y}(\xi-x)(\eta-y) + \frac{1}{2}\left.\frac{\partial^2 f}{\partial y^2}\right|_{x,y}(\eta-y)(\eta-y)$

or $f_{\text{appx}}(\xi,\eta,x,y) = f(x,y) + \nabla f \cdot \begin{pmatrix} \xi-x \\ \eta-y \end{pmatrix} + \frac{1}{2}\mathcal{Q}_{\text{form}}$ where $\mathcal{Q}_{\text{form}} \equiv (\xi-x, \eta-y) \begin{pmatrix} \left.\frac{\partial^2 f}{\partial x^2}\right|_{x,y} & \left.\frac{\partial^2 f}{\partial x \partial y}\right|_{x,y} \\ \left.\frac{\partial^2 f}{\partial y \partial x}\right|_{x,y} & \left.\frac{\partial^2 f}{\partial y^2}\right|_{x,y} \end{pmatrix} \begin{pmatrix} \xi-x \\ \eta-y \end{pmatrix}$

The function $f_{\text{appx}}(\xi,\eta,x,y)$ will be plotted as a function of ξ and η for $|\xi-x| < \delta$ and $|\eta-y| < \delta$ for a selected number of points (x,y).

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Taylor Expansions of a Scalar Function of $\vec{v}$ in the Neighborhood of Zero

This will demonstrate how to produce an expression-form for a first-order expansion to a function of three variables.

This is symbolic representation of a first order expansion (keeping all terms) of a function near x,y,z

```
SmallChangeSeries =
 Expand[Series[AScalarFunction[x + dx,
     y + dy, z + dz], {dx, 0, 1}, {dy, 0, 1},
    {dz, 0, 1}]] - AScalarFunction[x, y, z]
```

1

Using normal converts the approximation into an expression, but higher order terms are still present

```
dScalarFunction =
 Expand[Normal[SmallChangeSeries]]
```

2

The next step eliminates second- and third-order terms... (remember, dx, dy, and dz are small)

```
dScalarFunction = dScalarFunction /.
  {dx dy → 0, dy dz → 0, dx dz → 0}
```

3

```
dz SomeFunction^(0,0,1)[x, y, z] +
 dy SomeFunction^(0,1,0)[x, y, z] +
 dx SomeFunction^(1,0,0)[x, y, z]
```

The above form is like the thermodynamic expression:

$$dF = \frac{\partial F}{\partial x}\,dx + \frac{\partial F}{\partial y}\,dy + \frac{\partial F}{\partial z}\,dz$$

1: Here, we get a symbolic representation of the approximation of $F(x+dx, y+dy, z+dz)$ near the point $(x,y,z) = (0,0,0)$ to first order. The `Series` function first expands about the last iterator-argument. This produces a result that multiplies the expansion parameters `dx`, `dy`, and `dz` by the *order of approximation* functions `O[dx^2]`.

2: Using `Normal` and `Expand` on the series-result above yields an expression for the approximation, but it is not first-order as we might have intended. The `dx dy`-terms are a result of expanding the result of the three sequential expansions in the first command.

3: Here, a rule is applied to remove the second-order terms. The result has the same form as the thermodynamic expression:

$$\begin{aligned} dG &= \left(\frac{\partial G}{\partial P}\right)_{T,n} dP + \left(\frac{\partial G}{\partial T}\right)_{n,P} + \left(\frac{\partial G}{\partial n}\right)_{P,T} \\ &= V(P,T,n)dP - S(P,T,n)dT + \mu(P,T,n)dn \end{aligned}$$

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Approximating Surfaces at Points

Visualization of quadratic approximations to a surface at points on that surface

```
CrazyFun[x_, y_] :=
 Sin[5 π x] Sin[5 π y] / (x y) + Sin[5 π (x - 1)]
   Sin[5 π (y - 1)] / ((x - 1) (y - 1))
```
1

```
theplot = Plot3D[CrazyFun[x, y],
  {x, 0.1, .9}, {y, 0.1, .9}, PlotRange → All,
  Mesh → False, PlotStyle → {Opacity[0.5]}]
```
2

```
Approxfunction[x_, y_, xo_, yo_] :=
 Series[CrazyFun[x, y],
   {x, xo, 2}, {y, yo, 2}] // Normal
```
3

```
anapprox = Plot3D[
  Evaluate[Approxfunction[x, y, .7, .1]],
  {x, .7 - .1, .7 + .1}, {y, .1 - .1, .1 + .1}]
```
4

```
Show[ theplot, anapprox]
```
5

```
Table[{xo[i] = RandomReal[],
   yo[i] = RandomReal[]}, {i, 1, 100}];
```
6

```
ApproxPlot[i_] := Plot3D[Evaluate[
   Approxfunction[x, y, xo[i], yo[i]]],
  {x, xo[i] - .1, xo[i] + .1},
  {y, yo[i] - .1, yo[i] + .1}, PlotPoints → 6,
  Mesh → True, ColorFunction →
   (RGBColor[0.9 xo[i], 0.9 yo[i], #] &)]
```
7

```
GraphicsStack[1] = Show[ApproxPlot[1]];
```
8

```
GraphicsStack[i_] := GraphicsStack[i] =
  Show[GraphicsStack[i - 1], ApproxPlot[i]]
```
9

```
Manipulate[Show[theplot, GraphicsStack[i]],
 {{i, 3}, 1, 20, 1}]
```
10

1–2: *CrazyFun* is an example scalar function of two variables.

3–4: Using `Normal` to convert the Taylor Expansion obtained by `Series` at a point x_o, y_o produces a function *Approxfunction* of four variables (the point it is approximated (x_o, y_o) and the local expansion variables at that point (x, y)).

5: This illustrates how the local quadratic approximation fits the surface locally at a *particular* point $(0.7, 0.7)$ in a square of (half)-side-length 0.1

6: Generate a list of random points at which to visualize the local approximation.

7: *ApproxPlot* is an example that will plot the local approximation for any indexed random point. The surface is colored by using the value of the point as an indicator.

8-9: This is an example of producing a stack of graphics with a *recursive graphics function*. It iteratively adds a new approximating surface graphics object to the set of the previous ones.

10: This will produce a visualization of approximations at different parts of the original surface.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

Just a few of many examples of instances where Taylor's expansions are used are:

linearization Examining the behavior of a model near a point where the model is understood. Even if the model is wildly non-linear, a useful approximation is to make it linear by evaluating near a fixed point.

approximation If a model has a complicated representation in terms of unfamiliar functions, a Taylor expansion can be used to characterize the 'local' model in terms of a simple polynomial.

asymptotics Even when a system has singular behavior (e.g, the value of a function becomes infinite as some variable approaches a particular value), *how* the system becomes singular is very important. At singular points, the Taylor expansion will have leading order terms that are singular, for example near $x = 0$,

$$\frac{\sin(x)}{x^2} = \frac{1}{x} - \frac{x}{6} + \mathcal{O}(x^3) \tag{12-10}$$

The singularity can be subtracted off and it can be said that this function approaches ∞ "linearly" from below with slope -1/6. Comparing this to the behavior of another function that is singular near zero:

$$\frac{\exp(x)}{x} = \frac{1}{x} + 1 + \frac{x}{2} + \frac{x^2}{6} + \mathcal{O}(x^3) \tag{12-11}$$

shows that the e^x/x behavior is "more singular."

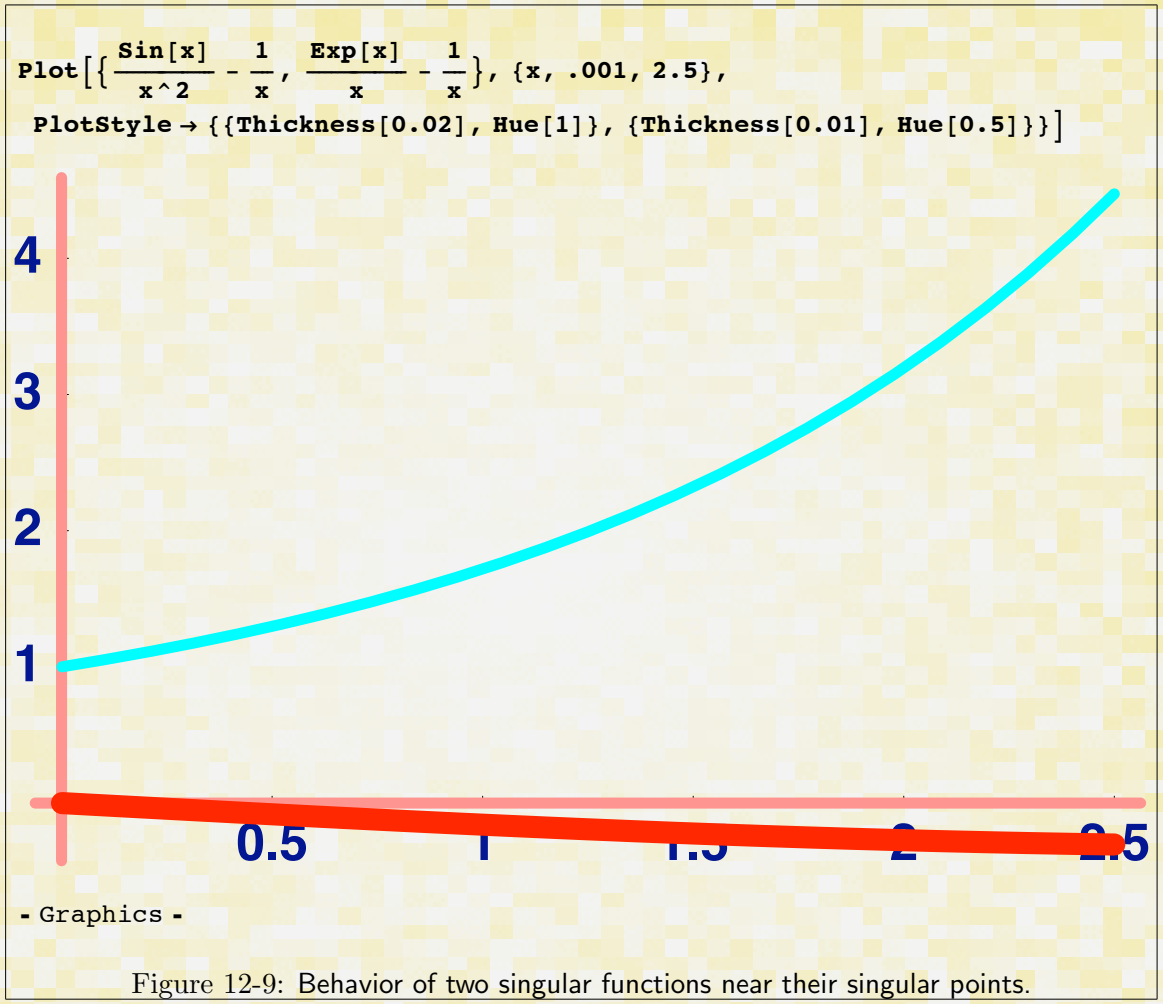


Figure 12-9: Behavior of two singular functions near their singular points.

stability If the expansion of energy about a point is obtained, then the various orders of expansion can be interpreted:

zero-order The zeroth-order term in a local expansion is the energy of the system at the point evaluated. Unless this term is to be compared to another point, it has no particular meaning (if it is not infinite), as energy is always

arbitrarily defined up to a constant.

first-order The first-order is related to the driving force to change the state of the system. Consider:

$$\Delta E = \nabla E \cdot \delta\vec{x} = -\vec{F} \cdot \delta\vec{x} \tag{12-12}$$

If force exists, the system can decrease its energy linearly by picking a particular change $\delta\vec{x}$ that is anti-parallel to the force.

For a system to be stable, it **is a necessary first condition that the forces (or first order expansion coefficients) vanish**.

second order If a system has no forces on it (therefore satisfying the necessary condition of stability), then where the system is stable or unstable depends on whether a small $\delta\vec{x}$ can be found that deceases the energy:

$$\begin{aligned}\Delta E &= \frac{1}{2}\delta\vec{x} \cdot \nabla -\vec{F} \cdot \delta\vec{x} \\ &= \frac{1}{2} \cdot \nabla\nabla E \cdot \delta\vec{x} \\ &= \frac{1}{2} \left.\frac{\partial^2 E}{\partial x_i \partial x_j}\right|_{x_1,x_2,\ldots x_n} \delta x_i \delta x_j\end{aligned} \tag{12-13}$$

where the summation convention is used above and the point $(x_1, x_2, \ldots, x_n)$ is one for which ∇E is zero.

numerics Derivatives are often obtained numerically in numerical simulations. The Taylor expansion provides a formula to approximate numerical derivatives—and provides an estimate of the numerical error as a function of quantities like numerical mesh size.

Gradients and Directional Derivatives

Scalar functions $F(x, y, z) = F(\vec{x})$ have a natural vector field associated with them—at each point $\vec{x}$ there is a direction $\hat{n}(\vec{x})$ pointing in the direction of the most rapid increase of F. Associating the *magnitude* of a vector in the direction of steepest increase with the *rate of increase* of F defines the gradient.

The gradient is therefore a vector function with a vector argument ($\vec{x}$ in this case) and it is commonly written as ∇F.

3.016 Home
Full Screen
Close
Quit

Here are some natural examples:

Meteorology The "high pressure regions" are commonly displayed with weather reports—as are the "isobars" or curves of constant barometric pressure. Thus displayed, pressure is a scalar function of latitude and longitude.

At any point on the map, there is a direction that points to a local high pressure center—this is the direction of the gradient. The rate at which the pressure is increasing gives the magnitude of the gradient.

The gradient of pressure should be a vector that is related to the direction and the speed of wind.

Mosquitoes It is known that hungry mosquitoes tend to fly towards sources (or local maxima) of carbon dioxide. Therefore, it can be hypothesized that mosquitoes are able to determine the gradient in the concentration of carbon dioxide.

Heat In an isolated system, heat flows from high-temperature ($T(\vec{x})$) regions to low-temperature regions.

The Fourier empirical law of heat flow states that the rate of heat flows is proportional to the local *decrease* in temperature.

Therefore, the local rate of heat flow should be proportional to the vector which is *minus* the gradient of $T(\vec{x})$: $-\nabla T$

Finding the Gradient

Potentials and Force Fields

Force is a vector. Force projected onto a displacement vector $d\vec{x}$ is the rate at which work, dW, is done *on* an object $dW = -\vec{F} \cdot d\vec{x}$.

If the work is being supplied by an external agent (e.g., a charged sphere, a black hole, a magnet, etc.), then it may be possible to ascribe a potential energy ($E(\vec{x})$, a scalar function with vector argument) to the agent associated with the position at which the force is being applied.[6] This $E(\vec{x})$ is the potential for the agent and the force field due to the agent is $\vec{F}(\vec{x}) = -\nabla E(\vec{x})$.

[6] As with any energy, there is always an arbitrary constant associated with the position (or state) at which the energy is taken to be zero. There is no such ambiguity with force. Forces are, in a sense, more fundamental than energies. Energy appears to be fundamental because all observations of the first law of thermodynamics demonstrate that there is a conserved quantity which is a state function and is called energy.

Sometimes the force (and therefore the energy) scales with the "size" of the object (i.e., the object's total charge in an electric potential due to a fixed set of charges, the mass of an object in the gravitational potential of a black hole, the magnetization of the object in a magnetic potential, etc.). In these cases, the potential field can be defined in terms of a unit size (per unit charge, per unit mass, etc.). One can determine whether such a scaling is applied by checking the units.

3.016 Home

Full Screen

Close

Quit

Lecture 13: Differential Operations on Vectors

Reading:
Kreyszig Sections: 9.8, 9.9 (pages410–413, 414–416)

Generalizing the Derivative

The number of different ideas, whether from physical science or other disciplines, that can be understood with reference to the "meaning" of a derivative from the calculus of scalar functions, is very very large. Our ideas about many topics, such as price elasticity, strain, stability, and optimization, are connected to our understanding of a derivative.

In vector calculus, there are generalizations to the derivative from basic calculus that act on a scalar and give another scalar back:

gradient (∇): A derivative on a scalar that gives a vector.

curl ($\nabla\times$): A derivative on a vector that gives another vector.

divergence ($\nabla\cdot$): A derivative on a vector that gives scalar.

Each of these have "meanings" that can be applied to a broad class of problems.

The gradient operation on $f(\vec{x}) = f(x,y,z) = f(x_1,x_2,x_3)$,

$$\text{grad} f = \nabla f \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right) = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right) f \tag{13-1}$$

has been discussed previously. The curl and divergence will be discussed below.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Scalar Potentials and their Gradient Fields

An example of a scalar potential, due three point charges in the plane, is visualized. Methods for computing a gradient are presented.

Simple 2 D 1/r potential

```
potential[x_, y_, xo_, yo_] :=
 -1 / Sqrt[(x - xo)^2 + (y - yo)^2]
```
1

A field source located a distance 1 south of the origin

```
HoleSouth[x_, y_] :=
 potential[x, y, Cos[3 Pi / 2], Sin[3 Pi / 2]]
```
2

```
HoleNorthWest[x_, y_] :=
 potential[x, y, Cos[Pi / 6], Sin[ Pi / 6]]
```
3

```
HoleNorthEast[x_, y_] :=
 potential[x, y, Cos[ 5 Pi / 6], Sin[5 Pi / 6]]
```
4

Function that returns the two dimensional (x,y) gradient field of any function declared a function of two arguments:

```
gradfield[scalarfunction_] :=
 {D[scalarfunction[x, y], x] // Simplify,
  D[scalarfunction[x, y], y] // Simplify}
```
5

Generalizing the function to any arguments:

```
gradfield[scalarfunction_, x_, y_] :=
 {D[scalarfunction[x, y], x] // Simplify,
  D[scalarfunction[x, y], y] // Simplify}
```
6

The sum of three potentials:

```
ThreeHolePotential[x_, y_] :=
 HoleSouth[x, y] +
  HoleNorthWest[x, y] + HoleNorthEast[x, y]
```
7

f(x,y) visualization of the scalar potential:

```
Plot3D[ThreeHolePotential[x, y],
 {x, -2, 2}, {y, -2, 2}]
```
8

Contour visualization of the three-hole potential

```
ContourPlot[ThreeHolePotential[x, y],
 {x, -2, 2}, {y, -2, 2}, PlotPoints → 40,
 ColorFunction → (Hue[1 - # * 0.66] &)]
```
9

1: This is the 2D $1/r$-potential; here *potential* takes four arguments: two for the location of the charge and two for the position where the "test" charge "feels" the potential.

2-4: These are three fixed charge potentials, arranged at the vertices of an equilateral triangle.

5: *gradfield* is an example of a function that takes a scalar function of x and y and returns a vector with component derivatives: the gradient vector of the scalar function of x and y.

6: However, the previous example only works for functions of x and y explicitly. This expands *gradfield* to other Cartesian coordinates other than x and y.

7: *ThreeHolePotential* is the superposition of the three potentials defined in 2–4.

8: `Plot3D` is used to visualize the superposition of the potentials due to the three charges.

9: `ContourPlot` is an alternative method to visualize this scalar field. The option `ColorFunction` points to an example of a *Pure Function*—a method of making functions that do not operate with the usual "square brackets." Pure functions are indicated with the & at the end; the # is a place-holder for the pure function's argument.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

Divergence and Its Interpretation

The divergence operates on a vector field that is a function of position, $\vec{v}(x,y,z) = \vec{v}(\vec{x}) = (v_1(\vec{x}), v_2(\vec{x}), v_3(\vec{x}))$, and returns a scalar that is a function of position. The scalar field is often called the divergence field of $\vec{v}$, or simply the divergence of $\vec{v}$.

$$\text{div } \vec{v}(\vec{x}) = \nabla \cdot \vec{v} = \frac{\partial v_1}{\partial x} + \frac{\partial v_2}{\partial y} + \frac{\partial v_3}{\partial z} = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right) \cdot (v_1, v_2, v_3) = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right) \cdot \vec{v} \tag{13-2}$$

Think about what the divergence means.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

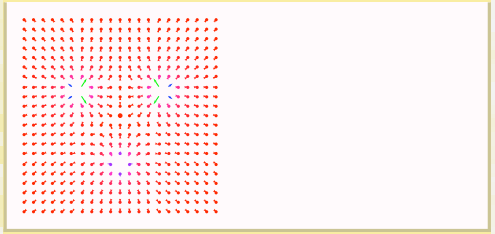
Visualizing the Gradient Field and its Divergence: The Laplacian

A visualization gradient field of the potential defined in the previous example is presented. The divergence of the gradient $\nabla \cdot \nabla\phi = \nabla^2\phi$ (i.e., the result of the Laplacian operator ∇^2) is computed and visualized.

Gradient field of three-hole potential

```
gradthreehole = gradfield[ThreeHolePotential]
```
1

```
Needs["VectorFieldPlots`"];
VectorFieldPlots`VectorFieldPlot[
 gradthreehole, {x, -2, 2}, {y, -2, 2},
 ScaleFactor → 0.2`, ColorFunction →
  (Hue[1 - #1 0.66`] &), PlotPoints → 21]
```
2



Function that takes a two-dimensional vector function of (x,y) as an argument and returns its divergence

```
divergence[{xcomp_ , ycomp_}] :=
 Simplify[D[xcomp, x] + D[ycomp, y]]
```
3

```
divgradthreehole = divergence[
   gradfield[ThreeHolePotential]] // Simplify
```
4

Plotting the divergence of the gradient

($\nabla \cdot (\nabla f)$ is the ``Laplacian'' $\nabla^2 f$, sometimes indicated with symbol Δf)

```
Plot3D[divgradthreehole, {x, -2, 2},
 {y, -2, 2}, PlotPoints -> 60]
```
5

1: We use our previously defined function *gradfield* to compute the gradient of *ThreeHolePotential* everywhere in the plane.

2: `PlotVectorField` is in the `VectorFieldPlots` package. Because a gradient produces a vector field from a scalar potential, arrows are used at discrete points to visualize it.

3: The divergence operates on a vector and produces a scalar. Here, we define a function, *divergence* , that operates on a 2D-vector field of x and y and returns the sum of the component derivatives. Therefore, taking the divergence of the gradient of a scalar field returns a scalar field that is naturally associated with the original—its physical interpretation is (minus) the rate at which gradient vectors "diverge" from a point.

4–5: We compute the divergence of the gradient of the scalar potential. This is used to visualize the Laplacian field of *ThreeHolePotential* .

3.016 Home
Full Screen
Close
Quit

The above definitions are for a Cartesian (x, y, z) system. Sometimes it is more convenient to work in other (spherical, cylindrical, etc) coordinate systems. In other coordinate systems, the derivative operations ∇, $\nabla\cdot$, and $\nabla\times$ have different forms. These other forms can be derived, or looked up in a mathematical handbook, or specified by using the MATHEMATICA® package "VectorAnalysis."

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Coordinate Transformations

Examples of *Coordinate Transformations* obtained from the `VectorAnalysis` package are presented.

It is no surprise that many of these differential operations already exist in Mathematica packages.

```
<< "VectorAnalysis`"
```
1

Converting between coordinate systems

The spherical coordinates expressed in terms of the cartesian x,y,z

```
CoordinatesFromCartesian[
 {x, y, z}, Spherical[r, theta, phi]]
```
2

$$\left\{\sqrt{x^2+y^2+z^2},\ \text{ArcCos}\left[\frac{z}{\sqrt{x^2+y^2+z^2}}\right],\ \text{ArcTan}[x, y]\right\}$$

The cartesian coordinates expressed in terms of the spherical $r\,\theta\,\phi$

```
CoordinatesToCartesian[
 {r, theta, phi}, Spherical[r, theta, phi]]
```
3

```
{r Cos[phi] Sin[theta],
 r Sin[phi] Sin[theta], r Cos[theta]}
```

The equation of a line through the origin in spherical coodinates

```
Simplify[
 CoordinatesFromCartesian[{a t, b t, c t},
  Spherical[r, theta, phi]], t > 0]
```
4

1–2: `CoordinatesFromCartesian` from the `VectorAnalysis` package transforms three Cartesian coordinates, named in the first argument-list, into one of many coordinate systems named by the second argument.

3: `CoordinatesToCartesian` transforms one of many different coordinate systems, named in the second argument, into the three Cartesian coordinates, named in the first argument (which is a list).

4: For example, this would be the equation of a line radiating from the origin in spherical coordinates.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Frivolous Example Using `Geodesy`, `VectorAnalysis`, and `CityData`.

We compute distances from Boston to Paris along different routes.

(The following will not work unless you have an active internet connection)

```
CityData["Boston", "Latitude"]
```
1

```
CityData["Marseille", "Latitude"]
```
2

```
CityData["Paris", "Longitude"]
```
3

```
SphericalCoordinatesofCity[
  cityname_String] := {
  6378.1 , CityData[cityname, "Latitude"]
   Degree,
  CityData[cityname, "Longitude"] Degree}
```
4

```
SphericalCoordinatesofCity["Boston"]
```
5

```
LatLong[city_String] :=
 {CityData[city, "Latitude"],
  CityData[city, "Longitude"]}
```
6

```
CartesianCoordinatesofCity[
  cityname_String] := CoordinatesToCartesian[
  SphericalCoordinatesofCity[cityname],
  Spherical[r, theta, phi]]
```
7

```
CartesianCoordinatesofCity["Paris"]
```
8

```
MinimumTunnel[city1_String, city2_String] :=
 Norm[CartesianCoordinatesofCity[city1] -
   CartesianCoordinatesofCity[city2]]
```
9

```
MinimumTunnel["Boston", "Paris"]
```
10

```
Needs["Geodesy`"]
```
11

```
SphericalDistance[
 LatLong["Paris"], LatLong["Boston"]]
```
12

```
SpheroidalDistance[
 LatLong["Paris"], LatLong["Boston"]]
```
13

1–3: `CityData` provides downloadable data. The data includes—among many other things—the latitude and longitude of many cities in the database. This show that Marseilles is north of Boston (which I found to be surprising).

4–5: *SphericalCoordinatesofCity* takes the string-argument of a city name and uses `CityData` to compute its spherical coordinates (i.e., $(r_{\text{earth}}, \theta, \phi)$ are same as (average earth radius = 6378.1 km, latitude, longitude)). We use `Degree` which is numerically $\pi/180$.

6: *LatLong* takes the string-argument of a city name and uses `CityData` to return a list-structure for its latitude and longitude. We will use this function below.

7–8: *CartesianCoordinatesofCity* uses a coordinate transform and *SphericalCoordinatesofCity*

9–10: If we imagine traveling *through* the earth instead of around it, we would use the `Norm` of the difference of the Cartesian coordinates of two cities.

11–12: Comparing the great circle route using `SphericalDistance` (from the `Geodesy` package) to the Euclidean distance, is a result that surprises me. It would save only about 55 kilometers to dig a tunnel to Paris—sigh.

13: `SpheroidalDistance` accounts for the earth's extra waistline for computing great-circle distances.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Gradient and Divergence Operations in Other Coordinate Systems

A $1/r^n$-potential is used to demonstrate how to obtain gradients and divergences in other coordinate systems.

```
SimplePot[x_, y_, z_, n_] :=
  1 / (x^2 + y^2 + z^2)^(n/2)
```
1

```
gradsp = Grad[
  SimplePot[x, y, z, 1], Cartesian[x, y, z]]
```
2

$$\left\{-\frac{x}{\left(x^2+y^2+z^2\right)^{3/2}},\ -\frac{y}{\left(x^2+y^2+z^2\right)^{3/2}},\ -\frac{z}{\left(x^2+y^2+z^2\right)^{3/2}}\right\}$$

The above is equal to $\vec{r}\,/\left(\|\vec{r}\|\right)^3$

```
SimplePot[r_, n_] := 1 / r^n
```
3

```
gradsphere =
 Grad[SimplePot[r, 1], Spherical[r, θ, φ]]
```
4

```
Grad[SimplePot[r, 1], Cylindrical[r, θ, z]]
```
5

```
Grad[SimplePot[r, 1],
 ProlateSpheroidal[r, θ, φ]]
```
6

```
GradSimplePot[x_, y_, z_, n_] :=
 Evaluate[Grad[SimplePot[x, y, z, n],
   Cartesian[x, y, z]]]
```
7

```
Div[GradSimplePot[x, y, z, n],
  Cartesian[x, y, z]] // Simplify
```
8

```
Div[GradSimplePot[x, y, z, 1],
  Cartesian[x, y, z]] // Simplify
```
9

```
0
```

1: *SimplePot* is the simple $1/r^n$ potential in Cartesian coordinates.

2: `Grad` is defined in the `VectorAnalysis`: in this form it takes a scalar function and returns its gradient in the coordinate system defined by the second argument.

3: An alternate form of *SimplePot* is defined in terms of a single coordinate; *if* r is the spherical coordinate $r^2 = x^2 + y^2 + z^2$ (referring back to a Cartesian (x, y, z)), then this is equivalent the function in **1**.

4: Here, the gradient of $1/r$ is obtained in spherical coordinates; it is equivalent to the gradient in **2**, but in spherical coordinates.

5: Here, the gradient of $1/r$ is obtained in cylindrical coordinates, but it is not equivalent to **2** nor **4**, because in cylindrical coordinates, (r, θ, z), $r^2 = x^2 + y^2$, even though the form appears to be the same.

6: Here, the gradient of $1/r$ is obtained in prolate spheroidal coordinates.

7: We define a function for the x–y–z gradient of the $1/r^n$ scalar potential. `Evaluate` is used in the function definition, so that `Grad` is not called each time the function is used.

8: The Laplacian $(\nabla^2(1/r^n))$ has a particularly simple form, $n(n-1)/r^{2+n}$

9: By inspection of $\nabla^2(1/r^n)$ or by direct calculation, it follows that $\nabla^2(1/r)$ vanishes identically.

3.016 Home
Full Screen
Close
Quit

Curl and Its Interpretation

The curl is the vector-valued derivative of a vector function. As illustrated below, its operation can be geometrically interpreted as the rotation of a field about a point.

For a vector-valued function of (x, y, z):

$$\vec{v}(x,y,z) = \vec{v}(\vec{x}) = (v_1(\vec{x}), v_2(\vec{x}), v_3(\vec{x})) = v_1(x,y,z)\hat{i} + v_2(x,y,z)\hat{j} + v_3(x,y,z)\hat{k} \tag{13-3}$$

the curl derivative operation is another vector defined by:

$$\text{curl } \vec{v} = \nabla \times \vec{v} = \left(\left(\frac{\partial v_3}{\partial y} - \frac{\partial v_2}{\partial z} \right), \left(\frac{\partial v_1}{\partial z} - \frac{\partial v_3}{\partial x} \right), \left(\frac{\partial v_2}{\partial x} - \frac{\partial v_1}{\partial y} \right) \right) \tag{13-4}$$

or with the memory-device:

$$\text{curl } \vec{v} = \nabla \times \vec{v} = \det \begin{pmatrix} \hat{i} & \hat{j} & \hat{k} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ v_1 & v_2 & v_3 \end{pmatrix} \tag{13-5}$$

For an example, consider the vector function that is often used in Brakke's Surface Evolver program:

$$\vec{w} = \frac{z^n}{(x^2+y^2)(x^2+y^2+z^2)^{\frac{n}{2}}}(y\hat{i} - x\hat{j}) \tag{13-6}$$

This will be shown below, in a MATHEMATICA® example, to have the property:

$$\nabla \times \vec{w} = \frac{nz^{n-1}}{(x^2+y^2+z^2)^{1+\frac{n}{2}}}(x\hat{i} + y\hat{j} + z\hat{k}) \tag{13-7}$$

which is spherically symmetric for $n = 1$ and convenient for turning surface integrals over a portion of a sphere, into a path-integral, over a curve, on a sphere.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Computing and Visualizing Curl Fields

Examples of curls are computing for a particular family of vector fields. Visualization is produced with the `VectorFieldPlot3D` function from the `VectorFieldPlots` package.

```
LeavingKansas[x_, y_, z_, n_] :=
                  z^n
 ------------------------------------ {y, -x, 0}
 (x^2 + y^2) (x^2 + y^2 + z^2)^(n/2)
```
1

```
Needs["VectorFieldPlots`"];

VectorFieldPlot3D[LeavingKansas[x, y, z, 3],
 {x, -1, 1}, {y, -1, 1},
 {z, -0.5, 0.5}, VectorHeads → True,
 ColorFunction → (Hue[#1 0.66`] &),
 PlotPoints → 21, ScaleFactor → 0.5`]
```
2

```
VectorFieldPlot3D[
 LeavingKansas[x, y, z, 3], {x, 0, 1},
 {y, 0, 1}, {z, 0.0, 0.5}, VectorHeads → True,
 ColorFunction → (Hue[#1 0.66] &),
 PlotPoints → 15, ScaleFactor → 0.5]
```
3

```
Curl[LeavingKansas[x, y, z, 3],
  Cartesian[x, y, z]] // Simplify
```
4

```
Glenda[x_, y_, z_, n_] :=
 Simplify[Curl[LeavingKansas[x, y, z, n],
   Cartesian[x, y, z]]]
```
5

```
VectorFieldPlot3D[
 Evaluate[Glenda[x, y, z, 1]],
 {x, -0.5, 0.5}, {y, -0.5, 0.5},
 {z, -0.25, 0.25}, VectorHeads → True,
 ColorFunction → (Hue[#1 0.66`] &),
 PlotPoints → 21]
```
6

Demonstrate that the divergence of the curl vanishes for the above function independent of n

```
DivCurl =
 Div[Glenda[x, y, z, n], Cartesian[x, y, z]]
```
7

```
Simplify[DivCurl]
```
8

1: *LeavingKansas* is the family of vector fields indicated by 13-6.

2–3: The function will be singular for $n > 1$ along the $z - axis$. This singularity will be reported during the numerical evaluations for visualization. There are two visualizations—the second one is over a sub-region but is equivalent because of the cylindrical symmetry of *LeavingKansas* . The singularity in the second case could be removed easily by excluding points near $z = 0$, but MATHEMATICA® seems to handle this fine without doing so.

4–6: This demonstrates the assertion, that for Eq. 13-7, the curl has cylindrical symmetry for arbitrary n, and spherical symmetry for $n = 1$.

7–8: This demonstrates that the divergence of the curl of $\vec{w}$ vanishes for any n; this is true for any differentiable vector field.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

One important result that has physical implications is that the curl of a gradient is always zero: $f(\vec{x}) = f(x, y, z)$:

$$\nabla \times (\nabla f) = 0 \tag{13-8}$$

Therefore *if some vector function* $\vec{F}(x, y, z) = (F_x, F_y, F_z)$ *can be derived from a scalar potential,* $\nabla f = \vec{F}$*, then the curl of* $\vec{F}$ *must be zero.* This is the property of an *exact* differential $df = (\nabla f) \cdot (dx, dy, dz) = \vec{F} \cdot (dx, dy, dz)$. Maxwell's relations follow from equation 13-8:

$$\begin{aligned}
0 &= \frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z} = \frac{\partial \frac{\partial f}{\partial z}}{\partial y} - \frac{\partial \frac{\partial f}{\partial y}}{\partial z} = \frac{\partial^2 f}{\partial z \partial y} - \frac{\partial^2 f}{\partial y \partial z} \\
0 &= \frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} = \frac{\partial \frac{\partial f}{\partial x}}{\partial z} - \frac{\partial \frac{\partial f}{\partial z}}{\partial x} = \frac{\partial^2 f}{\partial x \partial z} - \frac{\partial^2 f}{\partial z \partial x} \\
0 &= \frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} = \frac{\partial \frac{\partial f}{\partial y}}{\partial x} - \frac{\partial \frac{\partial f}{\partial x}}{\partial y} = \frac{\partial^2 f}{\partial y \partial x} - \frac{\partial^2 f}{\partial x \partial y}
\end{aligned} \tag{13-9}$$

Another interpretation is that gradient fields are *curl-free, irrotational, or conservative.*

The notion of "conservative" means that, if a vector function can be derived as the gradient of a scalar potential, then integrals of the vector function over any path is zero for a closed curve—meaning that there is no change in "state;" energy is a common state function.

Here is a picture that helps visualize why the curl invokes names associated with spinning, rotation, etc.

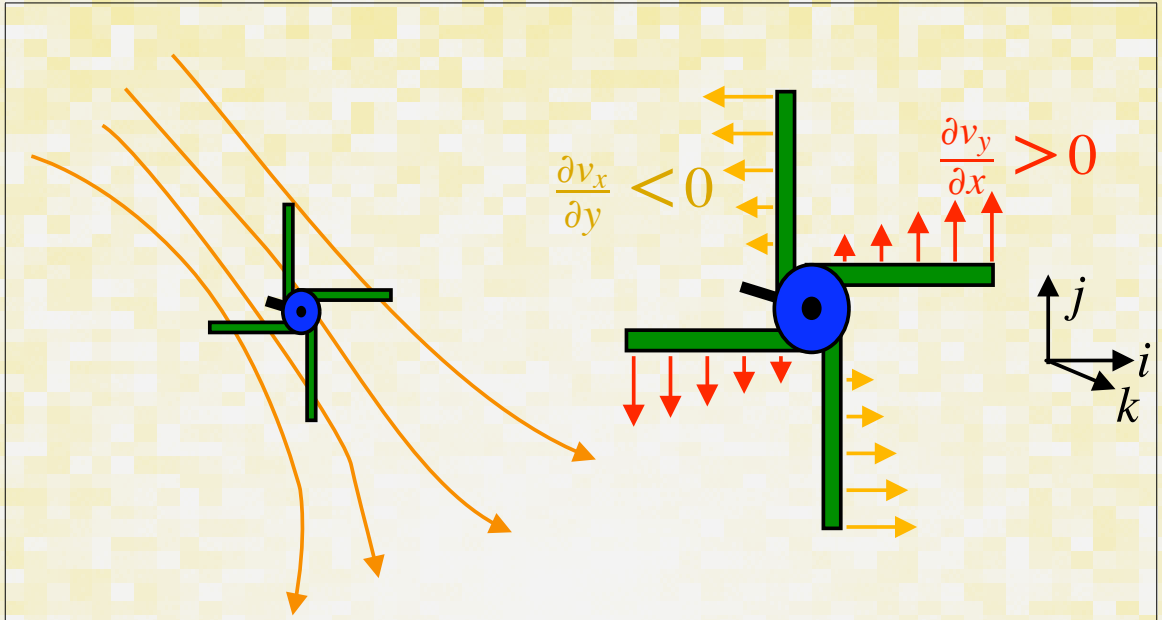


Figure 13-10: Consider a small paddle wheel placed in a set of stream lines defined by a vector field of position. If the v_y component is an increasing function of x, this tends to make the paddle wheel want to spin (positive, counter-clockwise) about the $\hat{k}$-axis. If the v_x component is a decreasing function of y, this tends to make the paddle wheel want to spin (positive, counter-clockwise) about the $\hat{k}$-axis. The net impulse to spin around the $\hat{k}$-axis is the sum of the two. Note that this is independent of the reference frame because a constant velocity $\vec{v} = \text{const.}$ and the local acceleration $\vec{v} = \nabla f$ can be subtracted because of Eq. 13-10.

Another important result is that divergence of any curl is also zero, for $\vec{v}(\vec{x}) = \vec{v}(x, y, z)$:

$$\nabla \cdot (\nabla \times \vec{v}) = 0 \tag{13-10}$$

Lecture 14: Integrals along a Path

Reading:
Kreyszig Sections: 10.1, 10.2, 10.3 (pages420–425, 426–432, 433–439)

Integrals along a Curve

Consider the type of integral that everyone learns initially:

$$E(b) - E(a) = \int_a^b f(x)dx \tag{14-1}$$

The equation implies that f is integrable and

$$dE = f dx = \frac{dE}{dx} dx \tag{14-2}$$

so that the integral can be written in the following way:

$$E(b) - E(a) = \int_a^b dE \tag{14-3}$$

where a and b represent "points" on some *line* where E is to be evaluated.

Of course, there is no reason to restrict integration to a straight line—the generalization is the integration along a curve (or a path) $\vec{x}(t) = (x_1(t), x_2(t), \ldots, x_n(t))$.

$$E(b) - E(a) = \int_{\vec{x}(a)}^{\vec{x}(b)} \vec{f}(\vec{x}) \cdot d\vec{x} = \int_a^b g(x(\vec{t}))dt = \int_a^b \nabla E \cdot \frac{d\vec{x}}{dt} dt = \int_a^b dE \tag{14-4}$$

This last set of equations assumes that the gradient exists–i.e., there is some function E that has the gradient $\nabla E = \vec{f}$.

Path-Independence and Path-Integration

If the function being integrated along a simply-connected path (Eq. 14-4) is a gradient of some scalar potential, then the path between two integration points does not need to be specified: the integral is independent of path. It also follows that for closed paths, the integral of the gradient of a scalar potential is zero.[7] A simply-connected path is one that does not self-intersect or can be shrunk to a point without leaving its domain.

There are familiar examples from classical thermodynamics of simple one-component fluids that satisfy this property:

$$\oint dU = \oint \nabla_{\vec{\mathcal{S}}} U \cdot d\vec{\mathcal{S}} = 0 \qquad \oint dS = \oint \nabla_{\vec{\mathcal{S}}} S \cdot d\vec{\mathcal{S}} = 0 \qquad \oint dG = \oint \nabla_{\vec{\mathcal{S}}} G \cdot d\vec{\mathcal{S}} = 0 \tag{14-5}$$

$$\oint dP = \oint \nabla_{\vec{\mathcal{S}}} P \cdot d\vec{\mathcal{S}} = 0 \qquad \oint dT = \oint \nabla_{\vec{\mathcal{S}}} T \cdot d\vec{\mathcal{S}} = 0 \qquad \oint dV = \oint \nabla_{\vec{\mathcal{S}}} V \cdot d\vec{\mathcal{S}} = 0 \tag{14-6}$$

Where $\vec{\mathcal{S}}$ is any other set of variables that sufficiently describe the equilibrium state of the system (i.e, $U(S,V)$, $U(S,P)$, $U(T,V)$, $U(T,P)$ for U describing a simple one-component fluid).

The relation curl grad $f = \nabla \times \nabla f = 0$ provides method for testing whether some *general* $\vec{F}(\vec{x})$ is independent of path. If

$$\vec{0} = \nabla \times \vec{F} \tag{14-7}$$

or equivalently,

$$0 = \frac{\partial F_j}{\partial x_i} - \frac{\partial F_i}{\partial x_j} \tag{14-8}$$

for all variable pairs x_i, x_j, then $\vec{F}(\vec{x})$ is independent of path. These are the Maxwell relations of classical thermodynamics.

[7] In fact, there are some extra requirements on the domain (i.e., the space of all paths that are supposed to be path-independent) where such paths are defined: the scalar potential must have continuous second partial derivatives everywhere in the domain.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Path Dependence of Integration of Vector Function: Non-Vanishing Curl

The path dependence of a vector field with a non-vanishing curl ($\vec{v}(\vec{x}) = xyz(\hat{i} + \hat{k} + \hat{z})$) is demonstrated with a family of closed curves.

Integrals over a Curve, Multidimensional Integrals

Examples of Path-Dependent Integrals: Vector Fields with Non-Vanishing Curl

Here is a vector function (xyz, xyz, xyz) for which the curl does not vanish anywhere, except the origin

```
Needs["VectorAnalysis`"];
VectorFunction = {x y z, x y z, y x z}
CurlVectorFunction = Simplify[Curl[VectorFunction, Cartesian[x, y, z]]]
```

```
{x y z, x y z, x y z}
```

```
{x (-y + z), y (x - z), (-x + y) z}
```

These are the conditions that the curl is zero:

```
ConditionsOfZeroCurl = Table[0 == CurlVectorFunction[[i]], {i, 3}]
```

```
{0 == x (-y + z), 0 == y (x - z), 0 == (-x + y) z}
```

There is only one point where this occurs:

```
FindInstance[ConditionsOfZeroCurl, {x, y, z}]
```

```
{{x → 0, y → 0, z → 0}}
```

Let's evaluate the integral of the vector potential ($\oint \vec{v} \cdot d\vec{s}$) for any curve that wraps around a cylinder of radius R with an axis that coincides with the z-axis

Any curve that wraps around the cylinder can be parameritized as (x(t), y(t), z(t)) = (R cos(t), R sin(t), A $P_{2\pi}$(t)) where $P_{2\pi}(t) = P_{2\pi}(t + 2\pi)$ and in particular $P_{2\pi}(0) = P_{2\pi}(2\pi)$.

Therefore $\vec{ds} = \left(-R \sin(t), R \cos(t), P'_{2\pi}(t)\right) dt = \left(-y(t), x(t), A\, P'_{2\pi}(t)\right) dt$

The integrand for an integral of "VectorFunction" around such a curve is (written in terms of an arbitrary P(t):

1: *VectorFunction* (xyz, xyz, xyz) is an example vector field that has a *non-vanishing curl*. The curl is computed with `Curl` which is in the `VectorAnalysis` package. Here, the particular coordinate system is specified with `Cartesian` argument to `Curl`.

2–3: The curl vanishes only at the origin—this is shown with `FindInstance` called with a list of equations corresponding to the vanishing curl.

4: This is the integrand $\vec{v} \cdot d\vec{s}$ computed as indicated in the figure, $d\vec{s} = -(y(t), x(t), P'(t))dt$. $P(\theta)$ represents any periodic function, but $(x, y) = R(\cos\theta, \sin\theta)$ representing paths that wrap around cylinders.

5: *PathDepInt* is an integral for $\vec{v}$ represented by *VectorFunction* an arbitrary path wrapping around the cylinder.

6–9: These are examples of a computation by using a replacement for a periodic $P(\theta)$ (i.e., each of the $P(\theta)$ begin and end at the same point, but the path between differs). The examples use $P(t) = \sin(t)$, $\cos(t)$, and $t(t - 2\pi)$. That the results differ shows that $\vec{v}$ is path-dependent—this is a general result for non-vanishing curl vector functions.

9–10: These results show that, for *some* closed paths, the result will be path-independent (here, for $P(t) = \cos(nt)$ the path-integral vanishes for integer n. This doesn't imply path-independence for all paths.

11: Our last result seems to contradict the result in **7** for which the integral was zero. However, computing the limit resolves the contradiction.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Examples of Path-Independence of Curl-Free Vector Fields

A curl-free vector field can be generated from any scalar potential, in this case $\vec{w} = \nabla e^{xyz} = \vec{w}(\vec{x}) = e^{xyz}(yz\hat{i} + zx\hat{k} + xy\hat{z})$ will be shown to be curl-free.

Try the path dependence with a conservative (curl free, or exact) Vector Function:

Start with a scalar potential

```
temp = Grad[Exp[x y z], Cartesian[x, y, z]]
```
1

Create another vector function that should have a zero curl

```
AnotherVFunction = {e^(x y z) y z, e^(x y z) x z, e^(x y z) x y}
Simplify[
 Curl[AnotherVFunction, Cartesian[x, y, z]]]
```
2

```
anothervf =
 AnotherVFunction.{-y, x, D[P[t], t]} /.
   {x → Radius Cos[t], y → Radius Sin[t],
    z → P[t]} // Simplify
```
3

The integral depends doesn't on the choice of P(t)

```
PathDepInt = Integrate[anothervf, t]
```
4

$$e^{\text{Radius}^2 \cos[t]\, P[t]\, \sin[t]}$$

```
(PathDepInt /. t → 2 Pi) - (PathDepInt /. t → 0)
```
5

0

1: To ensure that we will have a zero-curl, a vector field is generated from a gradient of a scalar potential. The curl vanishes because $\nabla \times \nabla f = 0$.

2: This is a demonstration that the curl does indeed vanish.

3: Here is the integrand for $\oint \vec{v} \cdot d\vec{s}$ for the family of paths that wrap around a cylinder for the particular case of this conservative fields.

4: This is the general result for the family of curves indicated by $P(\theta)$.

5: This demonstrates that the path integral closes for any periodic $P(\theta)$—which is the same as the condition that the curve is closed.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Examples of Path-Independence of Curl-Free Vector Fields on a Restricted Subspace

If a path-integral is path-dependent for an arbitrary three path, it is possible that path-independence can occur over closed paths restricted to some surface where the curl vanishes. To find a function that is curl-free on a restricted subspace (for example, the vector function $\vec{v}(\vec{x}) = (x^2 + y^2 - R^2)\hat{z}$ vanishes on the surface of a cylinder) one needs to find a $\vec{m}$ such that $\nabla \times \vec{m} = \vec{v}$ (for this case

For a last example, suppose the curl vanishes on the cylindrical surface defined above:

Suppose we can find a function that has a non-vanishing curl on this surface

We want to find a function which is generally non-curl free, but for which the curl vanishes on a surface. Let's pick the cylinder as our surface.

```
VanishOnCylinder = x^2 + y^2 - Radius^2
```
1

If a function can be found, that has the following curl, then we will have constructed such a function,

```
CurlOfOneStooge = {0, 0, VanishOnCylinder}
```
2

It is easy to see that this is the curl of Stooge, where we construct Stooge by integrating.

```
Stooge =
 {-1 / 2 Integrate[VanishOnCylinder, y],
  1 / 2 Integrate[VanishOnCylinder, x], 0}
```
3

In fact, we could add to Stooge, any vector function that has vanishing curl--there are an infinite number of these

```
Simplify[Curl[Stooge, Cartesian[x, y, z]]]
```
4

Its integral doesn't care which path around the cylinder it takes, the integrand doesn't depend on P(t)

```
WhyIOughta = Stooge.{-y, x, D[P[t], t]} /.
   {x → Radius Cos[t],
    y → Radius Sin[t], z → P[t]} // Expand
```
5

*This is the value for *any* path on the cylinder that is closed.*

```
Integrate[WhyIOughta, {t, 0, 2 Pi}]
```
6

$$-\frac{\pi\,\text{Radius}^4}{2}$$

1–3: This demonstrates a method to find a vector field for which the curl that vanishes on a on a surface. This is an example for the cylinder surface. The zero constraint, *VanishOnCylinder* , is used to produce a vector field that will represent the curl, *CurlOfOneStooge* . The formula for the curl is integrated to find the vector function, *Stooge* , that has the specified curl.

4: This demonstrates that the curl is what we designed it to be.

5–6: This demonstrates that the integral of *Stooge* is path-independent on the cylinder and its value is $-\pi R^4/2$.

3.016 Home
Full Screen
Close
Quit

Multidimensional Integrals

Perhaps the most straightforward of the higher-dimensional integrations (e.g., vector function along a curve, vector function on a surface) is a scalar function over a domain such as, a rectangular block in two dimensions, or a block in three dimensions. In each case, the integration over a dimension is uncoupled from the others and the problem reduces to pedestrian integration along a coordinate axis.

Sometimes difficulty arises when the domain of integration is not so easily described; in these cases, the limits of integration become functions of another integration variable. While specifying the limits of integration requires a bit of attention, the only thing that makes these cases difficult is that the integrals become tedious and lengthy. MATHEMATICA® removes some of this burden.

A short review of various ways in which a function's variable can appear in an integral follows:

	The Integral	Its Derivative
Function of limits	$p(x) = \int_{\alpha(x)}^{\beta(x)} f(\xi) d\xi$	$\frac{dp}{dx} = f(\beta(x))\frac{d\beta}{dx} - f(\alpha(x))\frac{d\alpha}{dx}$
Function of integrand	$q(x) = \int_{a}^{b} g(\xi, x) d\xi$	$\frac{dq}{dx} = \int_{a}^{b} \frac{\partial g(\xi, x)}{\partial x} d\xi$
Function of both	$r(x) = \int_{\alpha(x)}^{\beta(x)} g(\xi, x) d\xi$	$\frac{dr}{dx} = f(\beta(x))\frac{d\beta}{dx} - f(\alpha(x))\frac{d\alpha}{dx} + \int_{\alpha(x)}^{\beta(x)} \frac{\partial g(\xi, x)}{\partial x} d\xi$

Using Jacobians to Change Variables in Thermodynamic Calculations

Changing of variables is a topic in multivariable calculus that often causes difficulty in classical thermodynamics.

This is an extract of my notes on thermodynamics: http://pruffle.mit.edu/3.00/

Alternative forms of differential relations can be derived by changing variables.

To change variables, a useful scheme using Jacobians can be employed:

$$\begin{aligned}\frac{\partial(u,v)}{\partial(x,y)} &\equiv \det\begin{vmatrix}\frac{\partial u}{\partial x} & \frac{\partial u}{\partial y}\\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y}\end{vmatrix}\\ &= \frac{\partial u}{\partial x}\frac{\partial v}{\partial y} - \frac{\partial u}{\partial y}\frac{\partial v}{\partial x}\\ &= \left(\frac{\partial u}{\partial x}\right)_y\left(\frac{\partial v}{\partial y}\right)_x - \left(\frac{\partial u}{\partial y}\right)_x\left(\frac{\partial v}{\partial x}\right)_y\\ &= \frac{\partial u(x,y)}{\partial x}\frac{\partial v(x,y)}{\partial y} - \frac{\partial u(x,y)}{\partial y}\frac{\partial v(x,y)}{\partial x}\end{aligned} \tag{14-9}$$

$$\begin{aligned}\frac{\partial(u,v)}{\partial(x,y)} &= -\frac{\partial(v,u)}{\partial(x,y)} = \frac{\partial(v,u)}{\partial(y,x)}\\ \frac{\partial(u,v)}{\partial(x,v)} &= \left(\frac{\partial u}{\partial x}\right)_v\\ \frac{\partial(u,v)}{\partial(x,y)} &= \frac{\partial(u,v)}{\partial(r,s)}\frac{\partial(r,s)}{\partial(x,y)}\end{aligned} \tag{14-10}$$

For example, the heat capacity at constant volume is:

$$\begin{aligned}C_V &= T\left(\frac{\partial S}{\partial T}\right)_V = T\frac{\partial(S,V)}{\partial(T,V)}\\ &= T\frac{\partial(S,V)}{\partial(T,P)}\frac{\partial(T,P)}{\partial(T,V)} = T\left[\left(\frac{\partial S}{\partial T}\right)_P\left(\frac{\partial V}{\partial P}\right)_T - \left(\frac{\partial S}{\partial P}\right)_T\left(\frac{\partial V}{\partial T}\right)_P\right]\left(\frac{\partial P}{\partial V}\right)_T\\ &= T\frac{C_P}{T} - T\left(\frac{\partial P}{\partial V}\right)_T\left(\frac{\partial V}{\partial T}\right)_P\left(\frac{\partial S}{\partial P}\right)_T\end{aligned} \tag{14-11}$$

3.016 Home

Full Screen

Close

Quit

Using the Maxwell relation, $\left(\frac{\partial S}{\partial P}\right)_T = -\left(\frac{\partial V}{\partial T}\right)_P$,

$$C_P - C_V = -T\frac{\left[\left(\frac{\partial V}{\partial T}\right)_P\right]^2}{\left(\frac{\partial V}{\partial P}\right)_T} \tag{14-12}$$

which demonstrates that $C_P > C_V$ because, for any stable substance, the volume is a decreasing function of pressure at constant temperature.

. Example of a Multiple Integral: Electrostatic Potential above a Charged Region

This will be an example calculation of the spatially-dependent energy of a unit point charge in the vicinity of a charged planar region having the shape of an equilateral triangle. The calculation superimposes the charges from each infinitesimal area by integrating a $1/r$ potential from each point in space to each infinitesimal patch in the equilateral triangle The energy of a point charge $|e|$ due to a surface patch on the plane $z = 0$ of size $d\xi d\eta$ with surface charge density $\sigma(x, y)$ is:

$$dE(x, y, z, \xi, \eta) = \frac{|e|\sigma(\xi, \eta)d\xi d\eta}{\vec{r}(x, y, z, \xi, \eta)} \tag{14-13}$$

for a patch with uniform charge,

$$dE(x, y, z, \xi, \eta) = \frac{|e|\sigma d\xi d\eta}{\sqrt{(x-\xi)^2 + (y-\eta)^2 + z^2}} \tag{14-14}$$

For an equilateral triangle with sides of length one and center at the origin, the vertices can be located at $(0, \sqrt{3}/2)$ and $(\pm 1/2, -\sqrt{3}/6)$.
The integration becomes

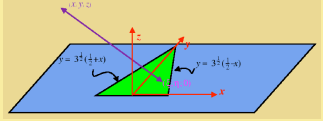
$$E(x, y, z) \propto \int_{-\sqrt{3}/6}^{\sqrt{3}/2} \left(\int_{\eta-\sqrt{3}/2}^{\sqrt{3}/2-\eta} \frac{d\xi}{\sqrt{(x-\xi)^2 + (y-\eta)^2 + z^2}} \right) d\eta \tag{14-15}$$

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Integrals over Variable Domains

This will demonstrate how MATHEMATICA® handles multiple integrals; in particular, when the domains depend on the integration variables. The goal is to find a function that will give the potential in the vicinity of a triangular patch with uniform charge density.

We will attempt to model the energy of ion just above one half of a triangular capacitor. Suppose there is a uniformly charged surface ($\sigma \equiv$ charge/area=1) occupying an equilaterial triangle in the z=0 plane:



what is the energy (voltage) of a unit positive charge located at (x,y,z)

The electrical potential goes like $\frac{1}{r}$, therefore the potential of a unit charge located at (x,y,z) from a small surface patch at (ξ,η,0) is

$$\frac{\sigma\, d\xi\, d\eta}{r} = \frac{d\xi\, d\eta}{\sqrt{(x-\xi)^2+(y-\eta)^2+z^2}}$$

Therefore it remains to integrate this function over the domain $\eta \in (0, \frac{\sqrt{3}}{2})$ and $\xi \in (\frac{\eta}{\sqrt{3}} - \frac{1}{2}), (\frac{1}{2} - \frac{\eta}{\sqrt{3}}))$

$$\int_0^{\frac{\sqrt{3}}{2}} \int_{\frac{\eta}{\sqrt{3}}-\frac{1}{2}}^{\frac{1}{2}-\frac{\eta}{\sqrt{3}}} \frac{d\xi\, d\eta}{\sqrt{(x-\xi)^2+(y-\eta)^2+z^2}}\, d\xi\, d\eta$$

```
Integrate[f[x, y], y, x]
Integrate[f[x, y], {y, Yi, Yf}, {x, Xi, Xf}]
Integrate[f[x, y],
 {y, Yi, Yf}, {x, Xi[y], Xf[y]}]
```

1

For example,consider the difference in the following two cases: First, we integrate over x and y using the two iterators in Integrate with the order {y,0,1}, {x,0,y}. Then explicitely using two separate steps

```
Integrate[Exp[3 x], {y, 0, 1}, {x, 0, y}]
interx = Integrate[Exp[3 x], {x, 0, y}]
Integrate[interx, {y, 0, 1}]
```

2

Compared to integrate over x and y using the two iterators in Integrate with the order {x,0,y},{y,0,1}. Then explicitely using two separate steps

```
Integrate[Exp[3 x], {x, 0, y}, {y, 0, 1}]
intery = (Integrate[Exp[3 x], {y, 0, 1}])
Integrate[intery, {x, 0, y}]
```

3

1: These examples demonstrate that MATHEMATICA® integrates over the last iterator which appears in the argument-list of `Integrate` first: LIFI-FILI (last iterator, first integrated; first iterator, last integrated).

2–3: Here we demonstrate the order of integration explicitly, by first integrating with two iterators, and then integrating in two step-sequence. The methods are equivalent.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Potential near a Charged and Shaped Surface Patch: Brute Force

A example of a multiple integral and its numerical evaluation for the triangular charged patch.

```
TrianglePotentialNumeric[x_, y_, z_] :=
 NIntegrate[1 / Sqrt[(x - ξ)^2 + (y - η)^2 + z^2], {η, 0,
   Sqrt[3] / 2}, {ξ, η / Sqrt[3] - 1 / 2, 1 / 2 - η / Sqrt[3]}]
```

1

```
TrianglePotentialNumeric[1, 3, .01]
```

2

```
Plot[TrianglePotentialNumeric[x, x, 1 / 40],
 {x, -1, 1}]
```

3

```
cplot[h_] := cplot[h] = ContourPlot[
   TrianglePotentialNumeric[x, y, h],
   {x, -1, 1}, {y, -0.5, 1.5}, Contours →
    Table[v, {v, .25, 2, .25}], ColorFunction →
    ColorData["TemperatureMap"],
   ColorFunctionScaling -> False,
   PlotPoints → 11]
Timing[cplot[1 / 10]]
```

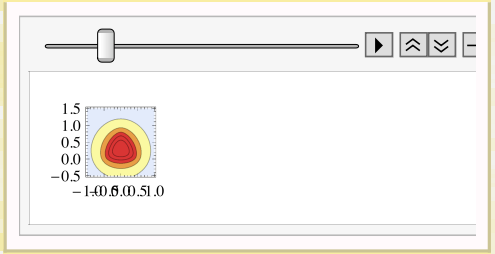
4

```
Row[{TextCell[
   "Computing ContourPlots a different
     h: Progress:  ", "Text"],
  ProgressIndicator[Dynamic[h], {0, .5}]}]
cplots = Table[cplot[h], {h, .025, .5, .025}];
```

5

```
ListAnimate[cplots]
```

6



1: MATHEMATICA® can't seem to find a closed-form solution to this integral over the triangular domain, However, the energy can be integrated numerically. Here is a function that calls `NIntegrate` for a location given by its arguments. We will call this function at different heights z. Multidimensional integration is generally computationally expensive.

2–3: Here are examples calling the numerical function *TrianglePotentialNumeric* . First, the function is evaluated at a single point; next, it is evaluated and plotted along a °45-line parallel in the $z = 1/40$ plane.

4: The function *cplot* calls *TrianglePotentialNumeric* repeatedly at variable x and y to generate a `ContourPlot` at height specified by the argument to *cplot* . These plots will eventually appear in an animation, so `ColorFunctionScaling` is set to false so that the colors will be consistent between frames. The `Contours` are set explicitly so that they are also consistent across frames. `Timing` indicates that each plot consumes a large number of cpu cycles.

5: Because each frame is expensive to compute, it is not a good idea to compute them *within* an animation. Here, we use `Table` to generate individual frames (n.b., the *cplots* stores its previous calculations in memory). Because this is time consuming, we add a *progress monitor* that will dynamically update as each `cplot[h]` is computed. We use `ProgressIndicator` on the argument `Dynamic[h]`. `Dynamic` informs MATHEMATICA® that a particular variable will be changing; therefore the object that calls it will need to be updated.

6: We use `ListAnimate` on the pre-computed frames.

3.016 Home
Full Screen
Close
Quit

Lecture 15: Surface Integrals and Some Related Theorems

Reading:
Kreyszig Sections: 10.4, 10.5, 10.6, 10.7 (pages439–444, 445–448, 449–458, 459–462)

Green's Theorem for Area in Plane Relating to its Bounding Curve

Reappraise the simplest integration operation, $g(x) = \int f(x)dx$. Temporarily ignore all the tedious mechanical rules of finding and integral and concentrate on what integration *does*.

Integration replaces a fairly complex process—adding up all the contributions of a function $f(x)$—with a clever new function $g(x)$ that only needs end-points to return the result of a complicated summation.

It is perhaps initially astonishing that this complex operation on the interior of the integration domain can be incorporated merely by the domain's endpoints. However, careful reflection provides a counterpoint to this marvel. How could it be otherwise? The function $f(x)$ is specified and there are no surprises lurking along the x-axis that will trip up dx as it marches merrily along between the endpoints. All the facts are laid out and they willingly submit to the process their preordination by $g(x)$ by virtue of the endpoints.[8]

The idea naturally translates to higher dimensional integrals and these are the basis for Green's theorem in the plane, Stoke's theorem, and Gauss (divergence) theorem. Here is the idea:

[8] I do hope you are amused by the evangelistic tone. I am a bit punchy from working non-stop on these lectures and wondering if anyone is really reading these notes. Sigh.

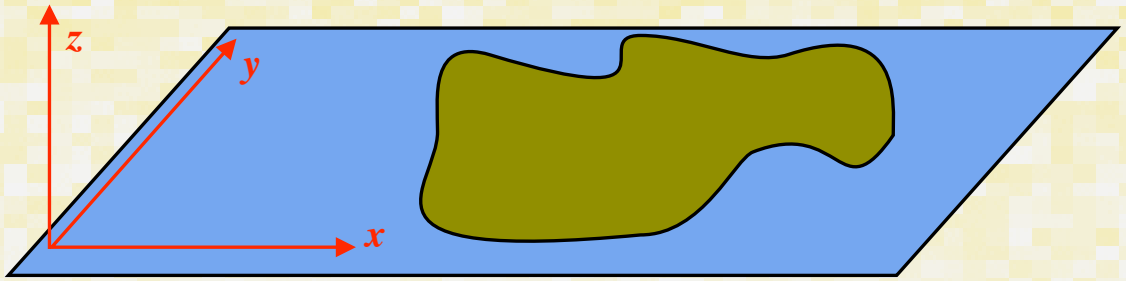


Figure 15-11: An irregular region on a plane surrounded by a closed curve. Once the closed curve (the edge of region) is specified, the area inside it is already determined. This is the simplest case as the area is the integral of the function $f = 1$ over $dxdy$. If some other function, $f(x, y)$, were specified on the plane, then its integral is also determined by summing the contributions along the boundary. This is a generalization $g(x) = \int f(x)dx$ and the basis behind Green's theorem in the plane.

The analog of the "*Fundamental Theorem of Differential and Integral Calculus*"[9] for a region $\mathcal{R}$ bounded in a plane with normal $\hat{k}$ that is bounded by a curve $\partial\mathcal{R}$ is:

$$\int\int_{\mathcal{R}} (\nabla \times \vec{F}) \cdot \hat{k} dxdy = \oint_{\partial\mathcal{R}} \vec{F} \cdot d\vec{r} \tag{15-1}$$

The following figure motivates Green's theorem in the plane:

[9] This is the theorem that implies the integral of a derivative of a function is the function itself (up to a constant).

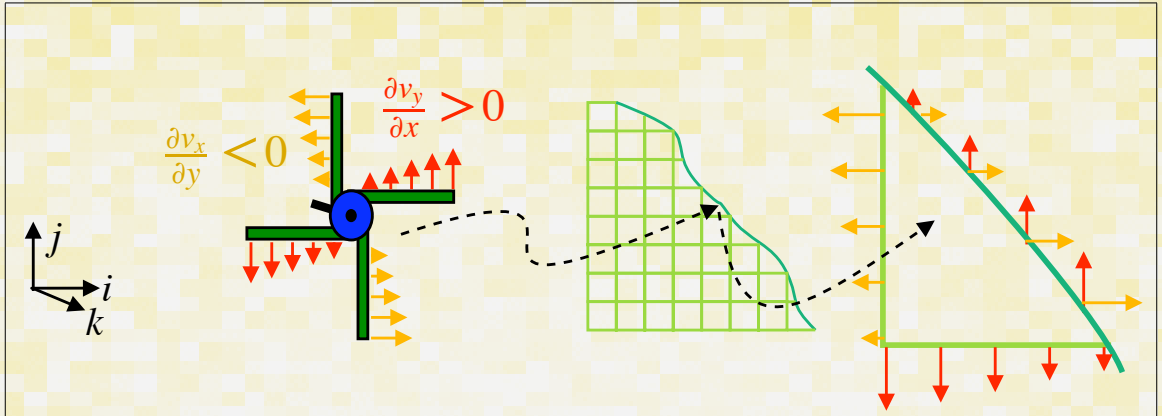


Figure 15-12: Illustration of how a vector valued function in a planar domain "spills out" of domain by evaluating the curl everywhere in the domain. Within the domain, the rotational flow ($\nabla \times F$) from one cell moves into its neighbors; however, at the edges the local rotation is a net loss or gain. The local net loss or gain is $\vec{F} \cdot (dx, dy)$.

The generalization of this idea to a surface $\partial\mathcal{B}$ bounding a domain $\mathcal{B}$ results in Stokes' theorem, which will be discussed later.

In the following example, Green's theorem in the plane is used to simplify the integration to find the potential above a triangular path that was evaluated in a previous example. The result will be a considerable increase of efficiency of the numerical integration because the two-dimensional area integral over the interior of a triangle is reduced to a path integral over its sides.

The objective is to turn the integral for the potential

$$E(x,y,z) = \iint_R \frac{d\xi d\eta}{\sqrt{(x-\xi)^2 + (y-\eta)^2 + z^2}} \tag{15-2}$$

into a path integral using Green's theorem in the x–y plane:

$$\int\int_R \left(\frac{\partial F_2}{\partial x} - \frac{\partial F_1}{\partial y} \right) dxdy = \int_{\partial R} (F_1 dx + F_2 dy) \tag{15-3}$$

To find the vector function $\vec{F} = (F_1, F_2)$ which matches the integral in question, set $F_2 = 0$ and integrate to find F_1 via

$$\int \frac{d\eta}{\sqrt{(x-\xi)^2 + (y-\eta)^2 + z^2}} \tag{15-4}$$

3.016 Home

Full Screen

Close

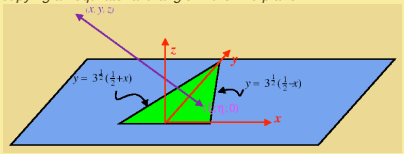
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Converting an area-integral over a variable domain into a path-integral over its boundary

We reproduce the example from Lecture 14 where the potential was calculated in the vicinity of a triangular patch, but with much improved accuracy and speed. The previous example's two dimensional numerical integration which requires $\mathcal{O}(N^2)$ calculations into a path integration around the boundary which requires $\mathcal{O}(N)$ evaluations for the same accuracy. The path of integration must be determined (i.e., $(x(t), y(t))$) and then the integration is obtained via $(dx, dy) = (x'dt, y'dt)$.

Suppose there is a uniformly charged surface (σ=charge/area=1) occupying an equilateral triangle in the z=0 plane:




```
F1[x_, y_, z_] =
 -Integrate[1/Sqrt[(x - ξ)^2 + (y - η)^2 + z^2], η]
```
1

The third (horizontal) boundary of the triangle patch looks like the easiest, let's see if an integral can be found over that patch:

```
Bottomside =
 F1[x, y, z] /. {ξ → t - 1/2, η → 0} // Simplify
```
2

```
NEside = F1[x, y, z] /.
  {ξ → (1 - t)/2, η → Sqrt[3] t/2} // Simplify
```
3

```
NWside = F1[x, y, z] /.
  {ξ → -t/2, η → Sqrt[3] (1 - t)/2} // Simplify
```
4

```
integrand =
 Simplify[-(NEside + NWside)/2 + Bottomside]
```
5

1: We use Green's theorem in the plane to turn our original integral

$$\iint_{\substack{\text{triangle}\\ \text{region}}} \left(\frac{\partial F_2}{\partial \eta} - \frac{\partial F_1}{\partial \xi}\right) d\xi d\eta = \phi(x,y,z)$$
$$= \iint \frac{d\eta d\xi}{r(x-\xi, y-\eta, z)} = \oint_{\substack{\text{triangle}\\ \text{perimeter}}} \vec{F} \cdot d\vec{s}$$

A closed form for F_1 (as indicated in Equation 15-4) is obtained with `Integrate`.

2: The bottom part of the triangle can be written as the curve: $(\zeta(t), \eta(t)) = (t - \frac{1}{2}, 0)$ for $0 < t < 1$; the integrand over that side is obtained by suitable replacement.

3–4: The remaining two legs of the triangle can be written similarly as: $((1-t)/2, \sqrt{3}t/2)$ and $(-t/2, \sqrt{3}(1-t)/2)$.

5: This is the integrand for the entire triangle to be integrated over $0 < t < 1$. Note, as t goes from 0 to 1, each leg of the triangle is traversed; this integrand sums all three contributions.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Faster and More Accurate Numerical Integration by Using Green's Theorem.

Continuing the example above, we are now able to find the potential over a triangular patch with uniform charge density, with a one-dimensional numerical integration, instead of the two-dimensional numerical integration in the last lecture.

Doing the same integral as in the previous lecture numerically, but this time over the boundary of the triangle instead of the triangle area.

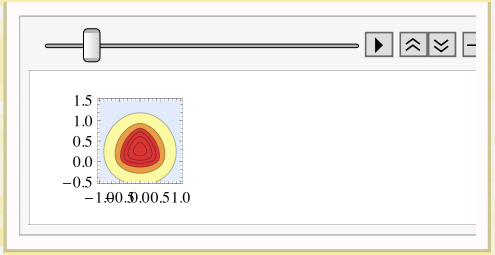
```
Pot[X_, Y_, Z_] := NIntegrate[Evaluate[
   integrand /. {x → X, y → Y, z → Z}], {t, 0, 1}]
```
1

We will create contourplots (level sets of constant potential) at as a function of different heights. We check the timing of the computation to compare to method in the last lecture.

```
ncplot[h_] :=
 ncplot[h] = ContourPlot[Pot[a, b, h],
   {a, -1, 1}, {b, -.5, 1.5}, Contours →
    Table[v, {v, .25, 2, .25}], ColorFunction →
    ColorData["TemperatureMap"],
   ColorFunctionScaling -> False,
   PlotPoints → 11 , ImageSize → {96, 72}]
Timing[ncplot[.05]]
```
2

```
Row[{TextCell[
   "Computing ContourPlots a different
     h: Progress:  ", "Text"],
  ProgressIndicator[Dynamic[h], {0, .5}]}]
ncplots = Table[ncplot[h],
   {h, .025, .5, .025}];
```
3

```
ListAnimate[ncplots]
```
4



1: There is no free lunch—the closed form of the integral is either unknown or takes too long to compute. However, `NIntegrate` is much more efficient because the problem has been reduced to a single integral instead of the double integral in the previous example.

2: A `ContourPlot` showing the level sets of the scalar potential field at a particular height h is obtained by a single call to the function *ncplot* . `Timing` shows that a speed-up factor of two is obtained for a single plot.

3: Here, we calculate a sequence of contour plots and store them for subsequent animation. Because this calculation takes a while to finish, we add a `ProgressIndicator`.

4: This is an animation for the potential in a plane as we increase the height of the plane above the triangular patch.

3.016 Home
Full Screen
Close
Quit

Representations of Surfaces

Integration over the plane $z = 0$ in the form of $\int f(x, y)dxdy$ introduces surface integration—over a planar surface—as a straightforward extension to integration along a line. Just as integration over a line was generalized to integration over a curve by introducing two or three variables that depend on a *single* variable (e.g., $(x(t), y(t), z(t))$), a surface integral can be conceived as introducing three (or more) variables that depend on two parameters (i.e., $(x(u, v), y(u, v), z(u, v))$).

However, there are different ways to formulate representations of surfaces:

Surfaces and interfaces play fundamental roles in materials science and engineering. Unfortunately, the mathematics of surfaces and interfaces frequently presents a hurdle to materials scientists and engineering. The concepts in surface analysis can be mastered with a little effort, but there is no escaping the fact that the algebra is tedious and the resulting equations are onerous. Symbolic algebra and numerical analysis of surface alleviates much of the burden.

Most of the practical concepts derive from a second-order Taylor expansion of a surface near a point. The first-order terms define a tangent plane; the tangent plane determines the surface normal. The second-order terms in the Taylor expansion form a matrix and a quadratic form that can be used to formulate an expression for curvature. The eigenvalues of the second-order matrix are of fundamental importance.

The Taylor expansion about a particular point on the surface takes a particularly simple form if the origin of the coordinate system is located at the point and the z-axis is taken along the surface normal as illustrated in the following figure.

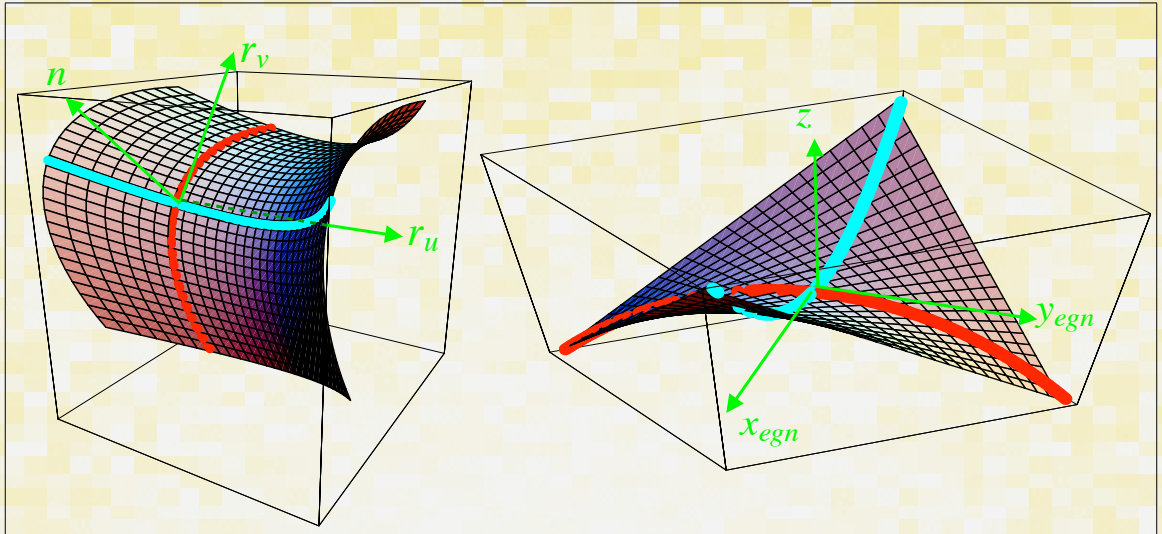


Figure 15-13: **Parabolic approximation to a surface and local eigenframe.** The surface on the left is a second-order approximation of a surface at the point where the coordinate axes are drawn. The surface has a local normal at that point which is related to the cross product of the two tangents of the coordinate curves that cross at the that point. The three directions define a coordinate system. The coordinate system can be translated so that the origin lies at the point where the surface is expanded and rotated so that the normal $\hat{n}$ coincides with the z-axis as in the right hand curve.

In this coordinate system, the Taylor expansion of $z = f(x, y)$ must be of the form

$$\Delta z = 0dx + 0dy + \frac{1}{2}(dx, dy) \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{pmatrix} \begin{pmatrix} dx \\ dy \end{pmatrix}$$

If this coordinate system is rotated about the z-axis into its eigenframe where the off-diagonal components vanish, then the two eigenvalues represent the maximum and minimum curvatures. The sum of the eigenvalues is invariant to transformations and the sum is known as the *mean curvature* of the surface. The product of the eigenvalues is also invariant—this quantity is known as the *Gaussian curvature*.

The method in the figure suggests a method to calculate the normals and curvatures for a surface. Those results are tabulated below.

Level Set Surfaces: Tangent Plane, Surface Normal, and Curvature

$$F(x,y,z) = \text{const}$$

Tangent Plane ($\vec{x} = (x,y,z)$, $\vec{\xi} = (\xi,\eta,\zeta)$)

$$\nabla F \cdot (\vec{\xi} - \vec{x}) \text{ or } \frac{\partial F}{\partial x}(\xi - x) + \frac{\partial F}{\partial y}(\eta - y) + \frac{\partial F}{\partial z}(\zeta - z)$$

Normal

$$\frac{\xi - x}{\frac{\partial F}{\partial x}} = \frac{\eta - y}{\frac{\partial F}{\partial y}} = \frac{\zeta - z}{\frac{\partial F}{\partial z}}$$

Mean Curvature

$$\nabla \cdot \left(\frac{\nabla F}{\|\nabla F\|} \right) \text{ or}$$

$$\frac{\left[\begin{array}{c} \left(\frac{\partial^2 F}{\partial y^2} + \frac{\partial^2 F}{\partial z^2} \right) \left(\frac{\partial F}{\partial x} \right)^2 + \left(\frac{\partial^2 F}{\partial z^2} + \frac{\partial^2 F}{\partial x^2} \right) \left(\frac{\partial F}{\partial y} \right)^2 + \left(\frac{\partial^2 F}{\partial x^2} + \frac{\partial^2 F}{\partial y^2} \right) \left(\frac{\partial F}{\partial z} \right)^2 \\ -2 \left(\frac{\partial F}{\partial x} \frac{\partial F}{\partial y} \frac{\partial^2 F}{\partial x \partial y} + \frac{\partial F}{\partial y} \frac{\partial F}{\partial z} \frac{\partial^2 F}{\partial y \partial z} + \frac{\partial F}{\partial z} \frac{\partial F}{\partial x} \frac{\partial^2 F}{\partial z \partial x} \right) \end{array} \right]}{\left(\frac{\partial F}{\partial x}^2 + \frac{\partial F}{\partial y}^2 + \frac{\partial F}{\partial z}^2 \right)^{3/2}}$$

3.016 Home

Full Screen

Close

Quit

Parametric Surfaces: Tangent Plane, Surface Normal, and Curvature

$$\vec{x} = (p(u,v), q(u,v), s(u,v)) \text{ or } x = p(u,v) y = q(u,v) z = s(u,v)$$

Tangent Plane ($\vec{x} = (x,y,z)$, $\vec{\xi} = (\xi,\eta,\zeta)$)

$$(\vec{\xi} - \vec{x}) \cdot (\frac{d\vec{x}}{du} \times \frac{d\vec{x}}{dv}) \det \begin{pmatrix} \xi - x & \eta - y & \zeta - z \\ \frac{\partial p}{\partial u} & \frac{\partial q}{\partial u} & \frac{\partial s}{\partial u} \\ \frac{\partial p}{\partial v} & \frac{\partial q}{\partial v} & \frac{\partial s}{\partial v} \end{pmatrix} = 0$$

Normal

$$\frac{\xi - x}{\frac{\partial(q,s)}{\partial(u,v)}} = \frac{\eta - y}{\frac{\partial(s,p)}{\partial(u,v)}} = \frac{\zeta - z}{\frac{\partial(p,q)}{\partial(u,v)}}$$

Mean Curvature

$$\frac{\left(\frac{d\vec{x}}{du} \cdot \frac{d\vec{x}}{du}\right)\left(\frac{d\vec{x}}{du} \times \frac{d\vec{x}}{dv} \cdot \frac{d^2\vec{x}}{dv^2}\right) - 2\left(\frac{d\vec{x}}{du} \cdot \frac{d\vec{x}}{dv}\right)\left(\frac{d\vec{x}}{du} \times \frac{d\vec{x}}{dv} \cdot \frac{d^2\vec{x}}{dudv}\right) + \left(\frac{d\vec{x}}{dv} \cdot \frac{d\vec{x}}{dv}\right)\left(\frac{d\vec{x}}{du} \times \frac{d\vec{x}}{dv} \cdot \frac{d^2\vec{x}}{du^2}\right)}{\left(\frac{d\vec{x}}{du} \times \frac{d\vec{x}}{dv} \cdot \frac{d\vec{x}}{du} \times \frac{d\vec{x}}{dv}\right)^{3/2}}$$

3.016 Home

Full Screen

Close

Quit

Graph Surfaces: Tangent Plane, Surface Normal, and Curvature

$$z = f(x, y)$$

Tangent Plane ($\vec{x} = (x, y, z)$, $\vec{\xi} = (\xi, \eta, \zeta)$)

$$\frac{\partial f}{\partial x}(\xi - x) + \frac{\partial f}{\partial y}(\eta - y) = (\zeta - z)$$

Normal

$$\frac{\xi - x}{\frac{\partial f}{\partial x}} = \frac{\eta - y}{\frac{\partial f}{\partial y}} = \frac{\zeta - z}{-1}$$

Mean Curvature

$$\frac{(1 + \frac{\partial f}{\partial x}^2)\frac{\partial^2 f}{\partial y^2} - 2\frac{\partial f}{\partial x}\frac{\partial f}{\partial y}\frac{\partial^2 f}{\partial x \partial y} + (1 + \frac{\partial f}{\partial y}^2)\frac{\partial^2 f}{\partial x^2}}{\sqrt{1 + \frac{\partial f}{\partial x}^2 + \frac{\partial f}{\partial y}^2}}$$

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Representations of Surfaces: Graphs $z = f(x, y)$ (part 1)

Visualization examples of surfaces represented by the graph $z = f(x, y)$; Examples of the use of `MeshFunctions` and `ColorFunction` to visualize various surface properties are given.

```
GraphFunction[x_, y_] :=
 (x - y) (x + y) / (1 + (x + y)^2)
assump = {x ∈ Reals, y ∈ Reals}
```
1

```
plotdefault = Plot3D[GraphFunction[x, y],
  {x, -3, 3}, {y, -3, 3}, PlotLabel → "Default"]
```
2

```
plotlevels =
 Plot3D[GraphFunction[x, y], {x, -3, 3},
  {y, -3, 3}, MeshFunctions → (#3 &),
  ColorFunction → "Rainbow",
  PlotLabel → "Constant Heights"]
```
3

```
angle[x_] := ((Pi / 2 + ArcTan[x]) / Pi)
angle[x_, y_] := ((Pi / 2 + ArcTan[x, y]) / Pi)
```
4

```
plotcircles = Plot3D[
  GraphFunction[x, y], {x, -3, 3}, {y, -3, 3},
  MeshFunctions → (Sqrt[#1^2 + #2^2] &),
  ColorFunction -> (Hue[angle[#1, #2] * 0.5] &),
  ColorFunctionScaling → False,
  PlotLabel → "Cylindrical Coordinates"]
```
5

```
CurvatureOfGraph[f_, x_, y_] :=
 FullSimplify[Module[
   {dfdx = D[f[x, y], x], dfdy = D[f[x, y], y],
    d2fdx2 = D[f[x, y], {x, 2}],
    d2fdy2 = D[f[x, y], {y, 2}],
    d2fdxdy = D[f[x, y], x, y]} ,
   Return[((1 + dfdx^2) d2fdx2 - 2 dfdx
        dfdy d2fdxdy + (1 + dfdy^2) d2fdy2) /
     Sqrt[1 + dfdx^2 + dfdy^2]]],
  Assumptions → assump]
```
6

```
CurvFunc = Function[{x, y}, Evaluate[
   CurvatureOfGraph[GraphFunction, x, y]]]
```
7

1: We will use *GraphFunction* as an example to show different ways to visualize a graph over an area.

2: `Plot3D` is used to plot *GraphFunction* with default settings.

3: Here is an example of using `MeshFunctions` to draw lines at constant altitude (i.e, constant values of $f(x, y)$)

4: This function, *angle* , which maps angles to the range $(0, 1)$ will be useful for visualization examples below (e.g., **5** and the following sections **2**).

5: This will help visualize a cylindrical- in addition to the Cartesian-coordinate system. The `MeshFunctions` option is used to plot concentric circles; `ColorFunction` illustrates the angular coordinate, θ, with `Hue`.

6: Our goal is to visualize curvature on top of the graph. This is a somewhat advanced example. Here we construct a function (*CurvatureOfGraph*) that computes the curvature $H(x, y)$ of an $f(x, y)$, and uses `FullSimplify` with assumptions that the coordinate are real numbers.

7: Here we use `Function` to create a symbol representing a function of two variables for the particular instance of the curvature of $f =$*GraphFunction* . `Evaluate` is used in the definition to ensure that the curvature computation is performed only once.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Representations of Surfaces: Graphs $z = f(x, y)$ (part 2)

We continue the example by visualizing the curvature and the inclination of the graph.

```
dfdx = Function[{x, y}, Evaluate[
   FullSimplify[D[GraphFunction[x, y], x],
    Assumptions → assump]]]
dfdy = Function[{x, y}, Evaluate[
   FullSimplify[D[GraphFunction[x, y], y],
    Assumptions → assump]]]
```

1

This is the surface with lines of constant curvature superimposed, and with colors associated with the local normal.

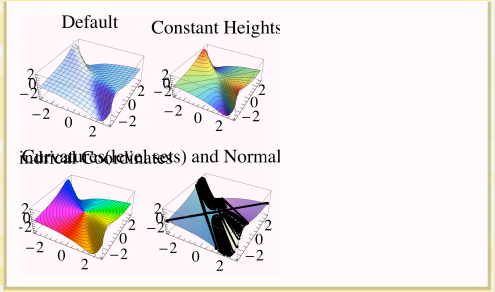
```
plotcurvature = Plot3D[
  GraphFunction[x, y], {x, -3, 3}, {y, 3, -3},
  MeshFunctions → (CurvFunc[#1, #2] &),
  MeshStyle → Thick, PlotLabel →
   "Curvatures(level sets) and Normals(color
     variation)", ColorFunction →
   (Glow[RGBColor[angle[dfdx[#1, #2]],
      angle[dfdy[#1, #2]], 0.75]] &),
  ColorFunctionScaling → False,
  Lighting → None]
```

2

Visualizing all the examples together.

```
GraphicsGrid[{{plotdefault, plotlevels},
  {plotcircles, plotcurvature}},
 ImageSize → 2 {72, 72}]
```

3



1: Two more symbols for functions of two arguments are created. Each represents a the slope of the tangent plane in the directions of the coordinate axes.

2: `Plot3D` is used to illustrate the local tangent-plane with `ColorFunction` which points to a red-scale for the surface slope in the x-direction and a blue-scale for the y-slope. We use `Glow` with `Lighting` set to none.

3: Finally, we use `GraphicsGrid` to illustrate the four graphic-examples together.

3.016 Home
Full Screen
Close
Quit

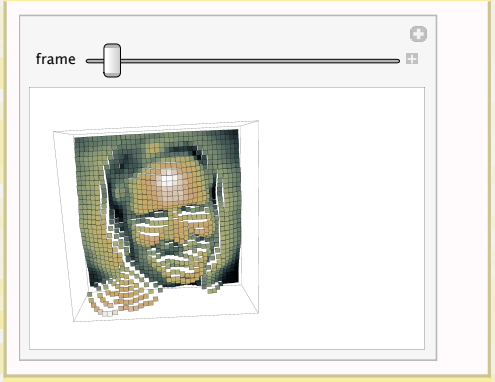
notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

A Frivolous Example for Graphs $z = f(x, y)$: Floating Pixels from Images in 3D

We demonstrate how to read a grey-scale image into MATHEMATICA® , and then use the pixel brightness values to displace the images according to $z = \text{brightness}(x, y)$.

```
MinMax[alist_List] :=
 Module[{flatlist = Flatten[alist]},
  Return[{Min[flatlist], Max[flatlist]}]]
mug = Import[
   "http://pruffle.mit.edu/~ccarter/ch_face
     _frames/Carter_2000_verysmall.png"];
ProgressIndicator[Dynamic[i], {1, 64}]
vp[i_] := {.1 Sin[(i - 1) Pi / 31],
   Sin[(i - 1) 2 Pi / 31], 2 Cos[2 (i - 1) Pi / 63]};
minmax = MinMax[mug[[1, 1]]];
Table[mugshot[i] =
    ListPlot3D[mug[[1, 1]], MeshStyle -> None,
     Mesh → None, InterpolationOrder → 0,
     ColorFunction → "GreenBrownTerrain",
     Axes → False, ViewPoint → vp[i],
     PlotRange → minmax,
     ImageSize → Full,
     SphericalRegion → True];, {i, 1, 64}];
Manipulate[mugshot[frame], {frame, 1, 64, 1}]
```

1



1: We first construct a function that will pick out the largest and smallest numbers in a list, and this will allow us to set `PlotRange` between the darkest and brightest pixels. (This function should probably check to ensure that the list contains only numeric entries, so that `Max` and `Min` return sensible results.) We will create a 3D rendering of pixels and "fly" through it. The function *vp* will provide the "orbit" for our flight through the pixels.

`Table` is used to create `Graphics3D` objects from different viewpoints for subsequent animation. Each graphics object is created with `ListPlot3D` with an array of pixel values for the first argument (`mug[[1,1]]`). Using `InterpolationOrder` set to zero implies that the plot's discrete values will not be continuously connected (i.e., the pixels are not "warped" to ensure continuity).

I used a modified version of this example to add an animation to my homepage

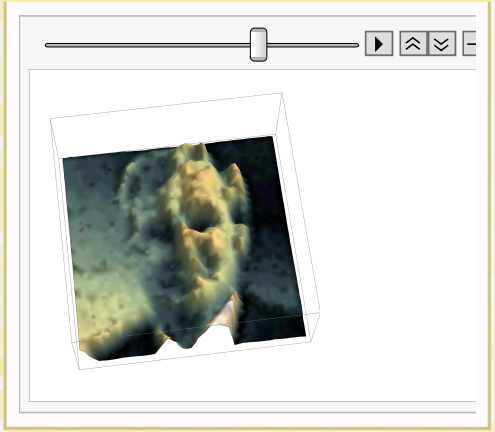
notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

A Frivolous Example for Graphs $z = f(x, y)$: Creating and Animating Surfaces from Image Sequences

We read in a sequence of images and use their pixel values to create an interpolation function for a surface $z = \text{brightness}(x, y)$. `Plot3D` calls the interpolation function produces a 3D animation from a 2D one.

```
Table[chface[read] = Import[
    "http://pruffle.mit.edu/~ccarter/ch_face
       _frames/ch_face." <>
     ToString[100 + read - 1] <> ".png"];
  facedata[read] = ListInterpolation[
    chface[read][[1, 1]], {{0, 1}, {0, 1}}];
  If[read == 1, minmax =
     MinMax[chface[read][[1, 1]]];, minmax =
    MinMax[{minmax, chface[read][[1, 1]]}]];,
  {read, 1, 28, 1}];
pface[i_] := Plot3D[facedata[i][x, y],
  {y, 0, 1}, {x, 0, 1}, PlotRange → minmax,
  ColorFunction → "GreenBrownTerrain",
  Mesh → False, Axes → False,
  ViewPoint → {-0.25, -2, 5}, ImageSize → All]
ListAnimate[Table[pface[gcomp],
  {gcomp, 1, 28, 1}], DefaultDuration → 10]
```

1



1: `Table` is used to iteratively read images that were created from a typical web-animation. (I am working on a way to do this directly from a single image file with multiple frames (with color), but haven't finished yet. `ListInterpolation` is used to create a continuous function of x and y in the domain $0 < |x|\&|y| < 1$. The height of the function corresponds to the brightness of the pixel. The function *pface* `[i]` produces a `Graphics3D` object for each frame in the animation. `ListAnimate` produces the animation from the image-functions.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Representations of Surfaces: Parametric Surfaces $\vec{x}(u, v)$

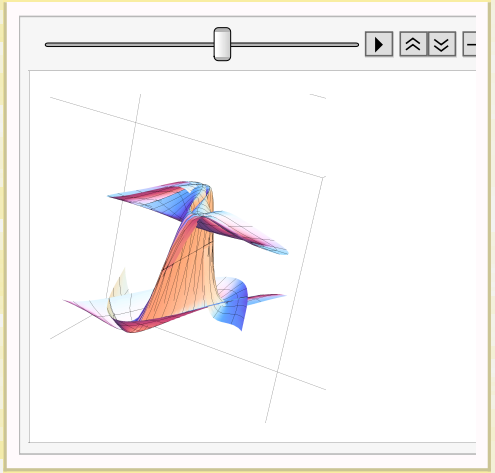
Visualization techniques for surfaces of the form $(x(u,v), y(u,v), z(u,v))$ are presented.

```
SurfaceParametric[u_, v_] := {Cos[u] v,
  u Cos[u + v], Cos[u] / (.1 + Cos[u]^2)}
```
1

```
ParametricPlot3D[
 Evaluate[SurfaceParametric[u, v]],
 {u, -2, 2}, {v, -2, 2}]
```
2

Using Manipulate, we can vary the boundary domain, and provide a more intuitive way to understand this complicated surface.

```
evolution = Table[ParametricPlot3D[
    Evaluate[SurfaceParametric[u, v]],
    {u, -ep, ep}, {v, -ep, ep}, PlotRange →
     {{-4, 4}, {-4, 4}, {-4, 4}}, PlotPoints →
     {1 + Round[ep / .125], 1 + Round[ep / .125]},
    ImageSize → Full], {ep, .125, 4.25, .125}];
ListAnimate[evolution, ImageSize → Full]
```
3



1–2: Using `ParametricPlot3D` to visualize a surface of the form $(x(u,v), y(u,v), z(u,v))$ given by *SurfaceParametric* . The lines of constant u and v generate the "square mesh" of the approximation to the surface. Each line on the surface is of the form: $\vec{r_1}(u) = (x(u, v = \text{const}), y(u, v = \text{const}), z(u, v = \text{const}))$ and $\vec{r_2}(v) = (x(u = \text{const}, v), y(u = \text{const}, v), z(u = \text{const}, v))$. The set of all crossing lines $\vec{r_1}(u)$ and $\vec{r_2}(v)$ is the surface. Each little "square" surface patch provides a convenient way to define the local surface normal—because both the vectors $d\vec{r_1}/du$ and $d\vec{r_2}/dv$ are tangent to the surface, their cross-product is either an inward-pointing normal or outward-pointing normal.

3: The nature of parametric surfaces are typically much more complicated than for graphs. Because the surface often folds over and through itself, it is difficult to comprehend its shape. For this case, it is useful to visualize the *evolution* of the surface as the domain of (u, v) increases. Here we use `Table` to iteratively increase the size of the domain, and then use `ListAnimate` to visualize its evolution.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Representations of Surfaces: Level Sets constant = $f(x, y, z)$

Visualization examples of surfaces represented their level sets constant = $F(x, y, z)$ are presented. This type of surface representation is particularly convenient when surfaces are disconnected, or merge during an evolution. Level sets are used extensively in *phase field models of microstructural evolution.*

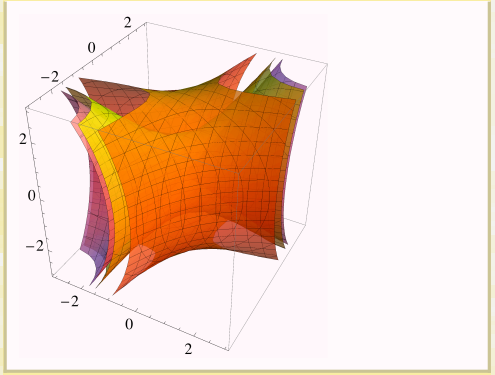
```
ConstFunction = x^2 - 4 x y + y^2 + z^2
```
1

```
ContourPlot3D[ConstFunction, {x, -1, 1},
 {y, -1, 1}, {z, -1, 1}, Contours → {2.5}]
```
2

The following statements produce contour plots of the same function, using two different methods for colorizing the surfaces...

```
cpa = ContourPlot3D[ConstFunction, {x, -3, 3},
  {y, -3, 3}, {z, -3, 3}, Contours → {0, 2, 8}]
```
3

```
cpb = ContourPlot3D[ConstFunction,
  {x, -3, 3}, {y, -3, 3}, {z, -3, 3},
  Contours → {0, 2, 8}, ContourStyle → {
    Directive[Pink, Opacity[0.8]],
    Directive[Yellow, Opacity[0.8]],
    Directive[Orange, Opacity[0.8]]}]
```
4




```
Manipulate[
 ContourPlot3D[ConstFunction, {x, -3, 3},
  {y, -3, 3}, {z, -3, 3}, Contours -> {i},
  ImageSize → Full], {i, -2, 10, .25}]
```
5

1: *ConstFunction* will be used for the following visualization examples.

1–2: A contour in two-dimensions is a curve; we have seen examples of such curves with `ContourPlot`. A contour in three-dimensions is a surface and we will use `ContourPlot3D` to visualize the level set formulation of a surface constant = $F(x, y, z)$ given by *ConstFunction* . Here, we explicitly specify those x, y, and z for which $x^2 - 4xy + y^2 + z^2 = 2.5$.

3: Here is an example of specifying three different level sets by passing several `Contours` to `ContourPlot3D`. It is difficult to distinguish which surface belongs to a particular level set.

4: The surfaces can be distinguished from one another with by giving each a different graphics `Directive` its own color. Setting `Opacity` to a value less than one helps eliminate the 'hidden surface' problem.

5: The evolution of level sets can be visualized with `Manipulate` by varying the value that is passed to `Contours`. It is apparent why this surface representation is useful when surfaces undergo topological changes. It may be helpful to consider these changes as a higher dimensional effect: consider $t = f(x, y, z)$ as a graph 'over' 3D region, or a four-dimensional surface. As a lower dimensional example (i.e., $t = f(x, y)$), consider the curves that develop as a torus (*ummmm doughnut*) is slice sequentially from one side. Initially the perimeter is an single closed elongated loop, which eventually begins to pinch in the middle and then break into isolated curves.

3.016 Home
Full Screen
Close
Quit

Integration over Surfaces

Integration of a function over a surface is a straightforward generalization of $\int\int f(x,y)dxdy = \int f(x,y)dA$. The set of all little rectangles $dxdy$ defines a planar surface. A non-planar surface $\vec{x}(u,v)$ is composed of a set of little parallelogram patches with sides given by the infinitesimal vectors

$$\begin{aligned} \vec{r_u}du &= \frac{\partial \vec{x}}{\partial u}du \\ \vec{r_v}du &= \frac{\partial \vec{x}}{\partial v}dv \end{aligned} \tag{15-5}$$

Because the two vectors $\vec{r_u}$ and $\vec{r_v}$ are not necessarily perpendicular, their cross-product is needed to determine the magnitude of the area in the parallelogram:

$$dA = \|\vec{r_u} \times \vec{r_v}\|dudv \tag{15-6}$$

and the integral of some scalar function, $g(u,v) = g(x(u,v), y(u,v)) = g(\vec{x}(u,v))$, on the surface is

$$\int g(u,v)dA = \int\int g(u,v)\|\vec{r_u} \times \vec{r_v}\|dudv \tag{15-7}$$

However, the operation of taking the norm in the definition of the surface patch dA indicates that some information is getting lost—this is the local normal orientation of the surface. There are two choices for a normal (inward or outward).

When calculating some quantity that does not have vector nature, only the magnitude of the function over the area matters (as in Eq. 15-7). However, when calculating a vector quantity, such as the flow through a surface, or the total force applied to a surface, the surface orientation matters and it makes sense to consider the surface patch as a vector quantity:

$$\begin{aligned} \vec{A}(u,v) &= \|\vec{A}\|\hat{n}(u,v) = A\hat{n}(u,v) \\ d\vec{A} &= \vec{r_u} \times \vec{r_v} \end{aligned} \tag{15-8}$$

where $\hat{n}(u,v)$ is the local surface unit normal at $\vec{x}(u,v)$.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Example of an Integral over a Parametric Surface

The surface energy of single crystals often depends on the surface orientation. This is especially the case for materials that have covalent and/or ionic bonds. To find the total surface energy of such a single crystal, one has to integrate an *orientation-dependent* surface energy, $\gamma(\hat{n})$, over the surface of a body. This example compares the total energy of such an anisotropic surface energy integrated over a sphere and a cube that enclose the same volume.

```
sphere[u_, v_] :=
 R {Cos[v] Cos[u], Cos[v] Sin[u], Sin[v]}
```
1

```
Ru[u_, v_] = D[sphere[u, v], u] // Simplify
Rv[u_, v_] = D[sphere[u, v], v] // Simplify
```
2

```
Needs["VectorAnalysis`"]
NormalVector[u_, v_] =
 CrossProduct[Ru[u, v], Rv[u, v]] // Simplify
NormalMag = FullSimplify[
  Norm[NormalVector[u, v]], Assumptions →
   {R ≥ 0, 0 ≤ u ≤ 2π, -π/2 < v < π/2}]
UnitNormal[u_, v_] =
 NormalVector[u, v] / NormalMag
```
3

```
SurfaceTension[nvec_] :=
 1 + gamma111 * nvec[[1]]^2 nvec[[2]]^2 nvec[[3]]^2
```
4

```
SphericalPlot3D[
 SurfaceTension[UnitNormal[u, v]] /.
  gamma111 → 12,
 {u, 0, 2 Pi}, {v, -Pi / 2, Pi / 2}]
```
5

```
SphereEnergy = Integrate[Integrate[
   SurfaceTension[UnitNormal[u, v]] Cos[v],
   {u, 0, 2π}], {v, -π/2, π/2}]
```
6

```
CubeSide = (4π/3)^(1/3)
```
7

```
CubeEnergy =
 6 (CubeSide^2 SurfaceTension[{1, 0, 0}])
```
8

```
EqualEnergies =
 Solve[CubeEnergy == SphereEnergy,
   gamma111] // Flatten
```
9

```
N[gamma111 /. EqualEnergies]
```
10

1: This is the parametric equation of the sphere in terms of longitude $v \in (0, 2\pi)$ and latitude $u \in (-\pi/2, \pi/2)$.

2: Calculate the tangent plane vectors $\vec{r_u}$ and $\vec{r_v}$

3: Using `CrossProduct` from the `VectorAnalysis` package to calculate a vector that is normal to the surface, $\vec{r_u} \times \vec{r_v}$, for subsequent use in the surface integral. Using `Norm` to find the magnitude of the local normal, we can produce a function to return the unit normal vector $\hat{n}$, *UnitNormal* , as a function of the surface parameters.

4: This is just an example of a $\gamma(\hat{n})$ that depends on direction that will be used for purposes of illustration.

5: Using `SphericalPlot3D`, the form of *SurfaceTension* for the particular choice of $\gamma_{111} = 12$ is visualized.

6: Using the result from $|\vec{r_u} \times \vec{r_v}|$, the total surface energy of a spherical body of radius $R = 1$ is computed by integrating $\gamma\hat{n}$ over the entire surface.

7–8: This would be the energy of a cubical body with the same volume as the sphere with unit radius. The cube is oriented so that its faces are normal to $\langle 100 \rangle$.

9–10: This calculation is not very meaningful, but it is the value of the surface anisotropy factor γ_{111} such that the cube and sphere have the same total surface energy. The total-surface-energy minimizing shape for a fixed volume is calculated using the *Wulff theorem*.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Lecture 16: Integral Theorems

Reading:
Kreyszig Sections: 10.8, 10.9 (pages463–467, 468–473)

Higher-dimensional Integrals

The fundamental theorem of calculus was generalized in a previous lecture from an integral over a single variable to an integration over a region in the plane. Specifically, for generalizing to Green's theorem in the plane, a vector derivative of a function integrated over a line and evaluated at its endpoints was generalized to a vector derivative of a function integrated over the plane.

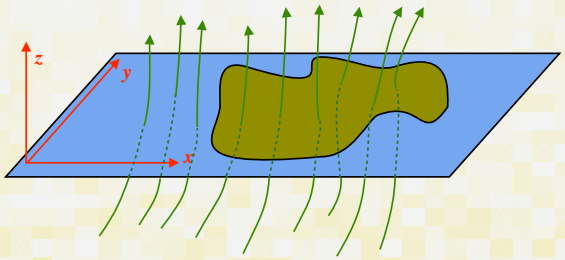


Figure 16-14: Illustrating how Green's theorem in the plane works. If a *known* vector function is integrated over a region in the plane then that integral should only depend on the bounding curve of that region.

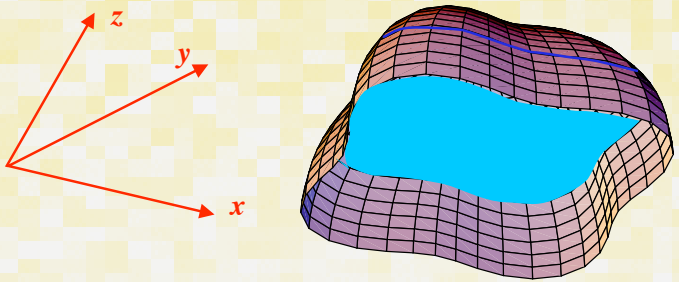


Figure 16-15: Illustration of a generalization to the Green's theorem in the plane: Suppose there is a bowl of a known shape submerged in a fluid with a trapped bubble. The bubble is bounded by two different surfaces, the bowl down to $z = 0$ and the planar liquid surface at that height. Integrating the function $\int_{V_B} dV$ over the bubble gives its volume. The volume must also be equal to an integral $\int \int_{\partial V_B} z dx dy$ over the (oriented) surface of the liquid. However, the volume of bubble can be determined from only the curve defined by the intersection of the bowl and the planar liquid surface; so the volume must also be equal to $\oint_C (\text{some function}) ds$.

The Divergence Theorem

Suppose there is "stuff" flowing from place to place in three dimensions.

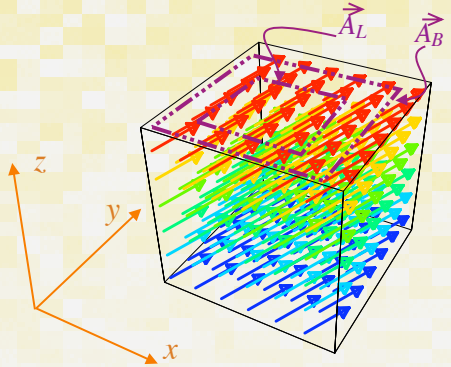


Figure 16-16: Illustration of a vector "flow field" $\vec{J}$ near a point in three dimensional space. If each vector represents the rate of "stuff" flowing per unit area of a plane that is normal to the direction of flow, then the dot product of the flow field integrated over a planar oriented area $\vec{A}$ is the rate of "stuff" flowing through that plane. For example, consider the two areas indicated with purple (or dashed) lines. The rate of "stuff" flowing through those regions is $\vec{J} \cdot \vec{A_B} = \vec{J} \cdot \hat{k} A_B$ and $\vec{J} \cdot \vec{A_L} = \vec{J} \cdot \hat{k} A_L$.

If there are no sources or sinks that create or destroy stuff inside a small box surrounding a point, then the change in the amount of stuff in the volume of the box must be related to some integral over the box's surface:

$$\begin{aligned} \frac{d}{dt}(\text{amount of stuff in box}) &= \frac{d}{dt}\int_{\text{box}} (\frac{\text{amount of stuff}}{\text{volume}}) dV \\ &= \int_{\text{box}} \frac{d}{dt}(\frac{\text{amount of stuff}}{\text{volume}}) dV \\ &= \int_{\text{box}} (\text{some scalar function related to } \vec{J}) dV \\ &= \int_{\substack{\text{box} \\ \text{surface}}} \vec{J} \cdot d\vec{A} \end{aligned} \tag{16-1}$$

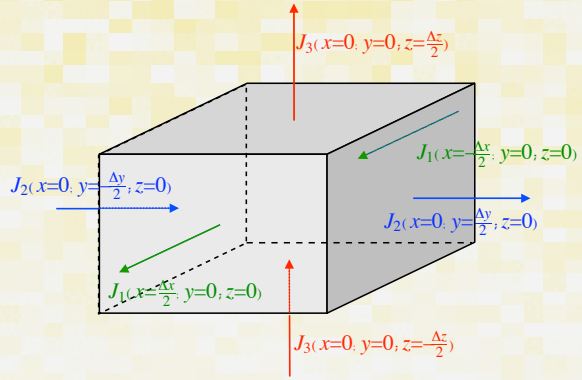


Figure 16-17: Integration of a vector function near a point and its relation to the change in that vector function. The rate of change of stuff is the integral of flux over the outside—and in the limit as the box size goes to zero, the rate of change of the amount of stuff is related to the sum of derivatives of the flux components at that point.

To relate the rate at which "stuff M" is flowing into a small box of volume $\delta V = dxdydz$ located at (x, y, z) due to a flux $\vec{J}$, note that the amount that M changes in a time Δt is:

$$\begin{aligned}
\Delta M(\delta V) &= (M \text{ flowing out of } \delta V) - (M \text{ flowing in } \delta V) \\
&= \vec{J}(x - \tfrac{dx}{2})\hat{i}dydz - \vec{J}(x + \tfrac{dx}{2}) \cdot \hat{i}dydz \\
&+ \vec{J}(y - \tfrac{dy}{2})\hat{j}dzdx - \vec{J}(y + \tfrac{dy}{2}) \cdot \hat{j}dzdx \ \Delta t \\
&+ \vec{J}(z - \tfrac{dz}{2})\hat{k}dxdy - \vec{J}(z + \tfrac{dz}{2}) \cdot \hat{k}dxdy \\
&= -(\frac{\partial J_x}{\partial x} + \frac{\partial J_y}{\partial y} + \frac{\partial J_z}{\partial z})\delta V \Delta t + \mathcal{O}(dx^4)
\end{aligned} \tag{16-2}$$

3.016 Home
Full Screen
Close
Quit

If $C(x,y,z) = M(\delta V)/\delta V$ is the concentration (i.e., stuff per volume) at (x,y,z), then in the limit of small volumes and short times:

$$\frac{\partial C}{\partial t} = -(\frac{\partial J_x}{\partial x} + \frac{\partial J_y}{\partial y} + \frac{\partial J_z}{\partial z}) = -\nabla \cdot \vec{J} = -\text{div}\vec{J} \tag{16-3}$$

For an arbitrary closed volume V bounded by an oriented surface ∂V:

$$\frac{dM}{dt} = \frac{d}{dt}\int_V C dV = \int_V \frac{\partial C}{\partial t} dV = -\int_V \nabla \cdot \vec{J} dV = -\int_{\partial V} \vec{J} \cdot d\vec{A} \tag{16-4}$$

The last equality

$$\int_V \nabla \cdot \vec{J} dV = \int_{\partial V} \vec{J} \cdot d\vec{A} \tag{16-5}$$

is called the Gauss or the divergence theorem.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

London Dispersion Potential due to a Finite Body

If the London interaction (i.e., energy between two induced dipoles) can be treated as a $1/r^6$ potential, then the potential due to a volume is an integration over each point in the volume and and arbitrary point in space. This calculation will be made much more efficient by turning the volume integral into a surface integral by using the divergence theorem.

Numerical integration is a cpu-time-consuming numerical procedure. If there is a way to reduce the dimensionality of the integration, then we can reap rewards for our cleverness. One trick is to use the divergence theorem to push the integration over a volume, to an integration over a surface. For example, we could use the divergence theorem:

$\iiint_{volume} \nabla\cdot\vec{P}\, dV = \iint_{surface} \vec{P}\cdot d\vec{A}$

For a $1/r^6$ potential , we must find a vector potential $\vec{P}$ such that $\nabla\cdot\vec{P} = -1/|\vec{r}-\vec{x}|^6$ where $\vec{r}$ is a position in the integrated volume and $\vec{x}$ is a point at which the potential is measured.

```
PVecLondon =
                1
-------------------------------
3 ((CX - X)^2 + (CY - Y)^2 + (CZ - Z)^2)^3
 {CX - X, CY - Y, CZ - Z}
```
1

```
Needs["VectorAnalysis`"]
```
2

```
FullSimplify[
 Div[PVecLondon, Cartesian[CX, CY, CZ]]]
```
3

We will integrate over a cylinder of radius R and length L along the z-axis, with its middle at the origin. First, let's use the radius of the cylinder to scale all the length variables: Let (X,Y,Z)/R = (x,y,z); (CX,CY,CZ)/R = (cx,cy,cz), and L/R = λ (the cylinder's aspect ratio).

```
ScaleRules = {X → x R, Y → y R, Z → z R,
   CX → cx R, CY → cy R, CZ → cz R};
PvecR5 = FullSimplify[R^5 PVecLondon /.
   ScaleRules, Assumptions -> R > 0]
```
4

Therefore, $\phi(\vec{x}) = \iiint_{volume} \frac{-1}{(\vec{r}-\vec{x})^6}\, dV = \iint_{surface} \overrightarrow{PVecLondon}\cdot d\vec{A} =$

$(\iint_{\substack{cylinder\\surface}} \overrightarrow{PVecR5}\cdot d\vec{A} + \iint_{\substack{cylinder\\ends}} \overrightarrow{PVecR5}\cdot d\vec{A})/R^5$

is the total interaction between a point and a cylinder. We can exploit the symmetry of the cylinder: $\rho = \sqrt{x\text{^}2 + y\text{^}2}$ and z.

We will do three integrals over the cylindrical surfaces using this expression to define the cylinder: (cx,cy,cz) = (Cos[θ], Sin[θ], cz):

The cylindrical surface is the domain $\theta \in (0, 2\pi)$, $cz \in (-\frac{\lambda}{2}, \frac{\lambda}{2})$

The two caps $r \in (0,1)$, $\theta \in (0, 2\pi)$, $cz = \pm\frac{\lambda}{2}$

1: To find a vector potential, $\vec{F}$ which has a divergence that is equal to $\nabla\cdot\vec{F} = -1/\|\vec{X} - C\vec{X}\|^6$, *PVecLondon* is a 'guess.' The $C\vec{X}$ will vary over the solid body and $\vec{X}$ is an arbitrary point at which the potential is to be determined.

2: We will need `Div` from the `VectorAnalysis` package.

3: this will show that the guess *PVecLondon* is a correct vector function for the $-1/r^6$ potential.

4: Our calculation will be for a cylinder of radius R and aspect ratio $\lambda \equiv L/R$. We will use R to scale all length variables and introduce dimensionless variables: $x = X/R$, $y = Y/R$, $z = Z/R$, $cx = CX/R$, $cy = CY/R$, and $cz = CZ/R$. The variables are scaled by introducing *ScaleRules* which are rules to be used in a replacement. Because the potential has a $1/(\text{length}^5)$ length scale, we multiply it by R^5 to remove that dimension. We use `FullSimplify` after the replacement with `Assumptions` of a positive radius to find the simplest possible form of the non-dimensionlized vector potential.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Cylinder Surface and Integrands

We parameterize the cylinder surface and compute the local oriented surface area and then find the integrand which is to be used for the cylinder surface.

The following is a parametric representation of a cylinder surface that is coaxial with the z-axis (the cylinder ends will be included later)

```
CylSurf = {Cos[θ], Sin[θ], cz}
```
1

The infinitessimal surface vectors $\vec{R_u}$ and $\vec{R_v}$ for the cylinder surface are obtained by differentiation; they will be used to find the surface patch $\vec{dA}$.

```
CylSurfRθ = D[CylSurf, θ]
CylSurfRcz  = D[CylSurf, cz]
```
2

The surface normal given by $\vec{R_u} \times \vec{R_v}$ for the cylinder surface, there for the following (multiplied by $d\theta\, dz$) is the infinitessimal oriented surface patch $\vec{dA}$.

```
NormalVecCylSurf =
 Cross[CylSurfRθ, CylSurfRcz]
```
3

The integrand to be evaluated over the cylinder surface is the vector potential, dotted into the normal vector. Because of the cylindrical symmetry of this model, we can convert to cylindrical coordinates. One set of coordinates is for the cylinder surface ($\xi \to R\, Cos[\theta]$, $\eta \to R\, Sin[\theta]$) for fixed radius R (which is a model parameter) and another set of coordinates for where we will be testing the potential ($x \to \rho\, Cos[\alpha]$, $y \to \rho\, Sin[\alpha]$). Because the potential must be independent of α, we might as well set it to zero.

```
CylinderIntegranddθdζ = FullSimplify[
  (PvecR5 /. {cx → Cos[θ], cy → Sin[θ],
      x → ρ , y → 0}).NormalVecCylSurf]
```
4

We have a choice whether to integrate over cz $\in (-\frac{1}{2}, \frac{1}{2})$ or $\theta \in (0, 2\pi)$ first. If we can a closed form for the cylinder surface over cz and then the cylinder end over r, then we can integrate the sum of these over θ together.

In the next section, we will see if we can do one of the two integrals---we have a choice of integrating over θ or (ζ for the cylinder sides, and R) for the cylinder ends. We find a closed form for integrating ζ for the sides and R for the top, and then subsequently numerically integrate θ for (0,2 π).

1: This is the cylinder surface in terms of cz and θ

2: These are the differential quantities that define the local tangent plane to the cylindrical surface.

3: This will be the multiplier elemental area for a parameterized cylindrical surface $d\vec{r}/d\theta \times d\vec{r}/dz$, this is the local normal to the surface; here it is the unit normal because we have scaled all length quantities by R

4: *CylinderIntegrand*$\theta\zeta$ is the integrand (i.e., the vector potential evaluated on the parameterized cylinder surface) for the cylindrical surface. Because of the cylindrical symmetry of the potential, the potential must be depend only on the normalized distance from the cylinder axis, ρ, and the height above the mid-plane, z: this conversion to cylindrical coordinates is effected by a rule-replacement operation.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Integrating over the Cylinder Surface

We have a choice whether to integrate over $\theta \in (0, 2\pi)$ or over $z \in -\lambda/2, \lambda/2$ first. We calculate the integral over cz first which will leave a form that we can numerically integrate over θ. (Note: As of 23 Oct. 2007, I've determined that *it is possible* to find the definite integral over θ and then over cz; therefore, this integral does has a closed form solution. For purposes of this demonstration, we will leave the integral over θ to be computed by a numerical integration. To demonstrate the idea of reducing the triple numerical integration, over a single numerical integration, I'll have to find a more complicated surface to integrate over in the future.)

```
UpperPlane = {λ > 0, ρ > 0, z > 0 , 0 < θ < 2 π};
CylinderIntegrandUpperZdθ =
 FullSimplify[Integrate[
   CylinderIntegranddθdζ, {cz, -λ / 2, λ / 2} ,
   Assumptions → UpperPlane],
  Assumptions → UpperPlane]
```
1

Here we restrict z to the upper half-space. We will treat z=0 below.

Here is the limit of the integral for z> 0 (CylinderIntegranddθ) in the limit as z → 0.

```
CylinderIntegranddθZeroLimit = FullSimplify[
  Limit[CylinderIntegrandUpperZdθ, z → 0]]
```
2

Here is the limit of the integral z=0, it is not obvious that the limit and its value at z=0 are the same.

```
FewerAssumptions =
  {R > 0, λ > 0, ρ > 0, 0 < θ < 2 π};
CylinderIntegrandAtZerodθ = Integrate[
  Evaluate[CylinderIntegranddθdζ /. z → 0],
  {cz, -λ / 2, λ / 2} ,
  Assumptions → FewerAssumptions]
```
3

The limit as z -> 0 and the integrand at z = 0 are the same, so we can use a single integrand

```
CylinderIntegranddθ[
  dist_, height_, AspectRat_] :=
 Evaluate[CylinderIntegrandUpperZdθ /.
   {ρ → dist, z → height, λ → AspectRat}]
?CylinderIntegranddθ
```
4

```
CylinderContribution[
  dist_, height_, AspectRat_] :=
 NIntegrate[CylinderIntegranddθ[dist,
   height, AspectRat], {θ, 0, 2 π}]
```
5

1: Because of the mirror symmetry of the function about the $z = 0$ plane, we can restrict the integral to $z > 0$ and use this as an assumption to aid the definite integral over cz. (Note this is a time-consuming integral and simplification, in the notebook form distributed with these notes, there is a dialogue that allows the user to download a precomputed result.)

2: To determine whether we can use this integrand at the mid-plane ($z = 0$), we check to see if the limit as $z \to 0$ is the same as evaluating the integrand at $z = 0$ first, and then finding the integral that applies for $z = 0$. Here, we check the limit.

3: Here, we set $z = 0$ and integrate.

4: The limit and the case of $z = 0$ are the same, so we use the form of the integrand, *CylinderIntegrandUpperZdθ* , calculated above. We turn the expression into a function by using `Evaluate` after the rule-replacement. This method of subverting the delayed evaluation, (`:=`), will work *so long as the function's variables have not been assigned.* These methods will be discussed in a section below . In practice, it is probably safer to replace variables with temporary, undefined, symbols and then cut-and-paste. (It is difficult to demonstrate the cut-and-paste with static notes like these.)

5: The function defined above, *Cylinderintegranddθ* , is used as the argument to the numerical integration, `NIntegrate`, over $\theta \in (0, 2\pi)$. The produces a function, *CylinderContribution* , that gives the contribution by integrating over the cylinder surface.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Integrating over the Cylinder's Top Surface

We parameterize the cylinder's top end-cap in terms of r (dimensionless $r < 1$) and θ, and then find a closed-form solution for the double integral over the top surface.

```
TopSurf = {r Cos[θ], r Sin[θ], λ / 2}
```
1

```
TopSurfRθ = D[TopSurf, θ]
TopSurfRr  = D[TopSurf, r]
```
2

```
NormalVecTopSurf =
 FullSimplify[Cross[TopSurfRr, TopSurfRθ]]
```
3

```
EndAssumptions =
  {λ > 0, ρ > 0 , z > 0, -1 ≤ Cos[θ] < 1};
TopIntegranddθdr = FullSimplify[
  (PvecR5 /. {cx → r Cos[θ], cy → r Sin[θ], cz →
      λ / 2, x → ρ, y → 0}).NormalVecTopSurf,
  Assumptions → EndAssumptions]
```
4

```
InsideAbovedr = Integrate[TopIntegranddθdr,
   {θ, 0, 2 π}, Assumptions →
    {0 < r < 1, λ > 0, ρ < 1 , z > λ / 2} ];
InsideBelowdr = Integrate[TopIntegranddθdr,
   {θ, 0, 2 π}, Assumptions →
    {0 < r < 1, λ > 0, ρ < 1 , z < λ / 2} ];
OutsideAbovedr = Integrate[TopIntegranddθdr,
   {θ, 0, 2 π}, Assumptions →
    {0 < r < 1, λ > 0, ρ > 1 , z > λ / 2} ];
OutsideBelowdr = Integrate[TopIntegranddθdr,
   {θ, 0, 2 π}, Assumptions →
    {0 < r < 1, λ > 0, ρ > 1 , z < λ / 2} ];
Grid[{{InsideAbovedr, InsideBelowdr},
  {OutsideAbovedr, OutsideBelowdr}}]
```
5

```
TopIntegranddr = ©InsideAbovedr
```
6

```
TopPart = Integrate[TopIntegranddr, {r, 0, 1},
  Assumptions → {λ > 0, ρ > 0 , z > 0, z ≠ λ / 2}]
```
7

```
TopContribution[dist_, height_,
  AspectRat_] := Evaluate[TopPart /.
   {ρ → dist,  z → height, λ →  AspectRat}]
? TopContribution
```
8

1–4: As in the case for the cylinder's curved surface, the top surface is parameterized, then the local tangent is computed, and the local oriented surface differential element is computed. The integrand is produced with the inner-product with the vector potential evaluated at the cylinder's top.

5–6: There is a singularity at the cylinder surface that produces a little extra work on our part to ensure that we don't evaluate at this singularity. To get a closed form of the integral over θ, it is useful to divide space into four regions where the potential is to be measured: 1) Inside the cylinder radius and above the cylinder top; 2) Inside the cylinder radius and below the cylinder top; 3) Outside the cylinder radius and above the cylinder top; 4) Outside the cylinder radius and below the cylinder top. These give the same result, so long as we don't evaluate at the cylinder's surface.

7: The top integrand in r can be integrated for $r \in (0, 1)$ and produces a closed form.

8: A function for the contribution of the upper disk, *TopContribution* , is defined.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Integrating over the Cylinder's Bottom Surface

We parameterize the cylinder's bottom end-cap ($cz = -\lambda/2$) in terms of r (dimensionless $r < 1$) and θ, and then find a closed-form solution for the double integral over the bottom surface.

```
BotSurf = {r Cos[θ], r Sin[θ], -λ / 2}
```
1

```
BotSurfRθ = D[BotSurf, θ]
BotSurfRr  = D[BotSurf, r]
```
2

```
NormalVecBotSurf =
 FullSimplify[Cross[BotSurfRθ, BotSurfRr]]
```
3

```
BotIntegranddθdr =
 FullSimplify[(PvecR5 /. {cx → r Cos[θ],
     cy → r Sin[θ], cz → -λ / 2, x → ρ, y → 0}).
   NormalVecBotSurf, Assumptions →
   EndAssumptions]
```
4

```
inside = { 0 < r < 1, λ > 0, ρ < 1 , z > 0};
outside = { 0 < r < 1, λ > 0, ρ > 1 , z > 0};
BotIntegrandInsidedr =
 Simplify[Integrate[BotIntegranddθdr,
   {θ, 0, 2 π}, Assumptions → inside],
  Assumptions → inside]
BotIntegrandOutsidedr =
 Simplify[Integrate[BotIntegranddθdr,
   {θ, 0, 2 π}, Assumptions → outside],
  Assumptions → outside ]
```
5

```
BotIntegranddr = BotIntegrandOutsidedr
```
6

```
BotPart = Integrate[BotIntegranddr, {r, 0, 1},
  Assumptions → {λ > 0, ρ > 0 , z > 0}]
```
7

```
BotContribution[dist_, height_,
  AspectRat_] := Evaluate[BotPart /.
   {ρ → dist,  z → height, λ →  AspectRat}]
```
8

```
LondonCylinderPotential[dist_, height_,
  AspectRat_]  := CylinderContribution[
   dist, height, AspectRat] +
  TopContribution[dist, height, AspectRat] +
  BotContribution[dist, height, AspectRat]
```
9

1–4: As above for the cylinder's outside and for its top surface, the bottom disk is parameterized, then the local tangent is computed, and the local oriented surface differential element is computed. The integrand is produced with the inner-product with the vector potential evaluated at the cylinder's bottom.

5–6: Similar to our method of avoiding the singularity at the top surface To get a closed form of the bottom-disk integral over θ, space is divided into two regions where the potential is to be measured: 1) Inside the cylinder; 2) Outside the cylinder. These give the same result.

7: The bottom integrand in r can be integrated for $r \in (0, 1)$ and produces a closed form.

8: A function for the contribution of the bottom disk, *BotContribution* , is defined.

9: We can produce a function to compute the potential at any point in space by summing the contributions from all three cylinder surfaces. The first function is the most expensive because it contains a numerical integration over θ.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Efficiency and Speed Issues: When to Evaluate the Right-Hand-Side of a Function in MATHEMATICA® .

The standard practice is to define functions in Mathematica with `:=`. However, sometimes it makes sense to evaluate the right-hand-side when the function definition is made. These are the cases where the right hand side would take a long time to evaluate—each time the function is called, the evaluation would be needed again and again. The following example illustrates a case where it makes sense to use `Evaluate` in a function definition (or, equivalently defining the function with immediate assignment `=`).

As in the use of `(=)`, this can result in errors if the function's variables have been defined previously. In cases where it is desirable to create a function from an expression, it is probably safest to use rule-replacement with undefined variables, observe the result, and then use cut-and-paste to define a function with a delayed evaluation in terms of these demonstrably undefined variables.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

To Evaluate or Not to Evaluate when Defining Functions

This example illustrates a case in which immediate evaluation `=` would be preferable to delayed evaluation `:=`

Let's set a baseline to check efficiency. Here we check timing to integrate something

```
Timing[Integrate[Exp[Tan[x]], {x, 0, c}]]
```
1

We check the same thing again, because Mathematica may have spent some time loading algorithms to integrate

```
Timing[Integrate[Exp[Tan[x]], {x, 0, c}]]
```
2

Here, we time how long it takes to create a function (with delayed assignment), but using Evaluate on the rhs.

```
Timing[DelayedEvaluated[c_] :=
  Evaluate[Integrate[Exp[Tan[x]], {x, 0, c}]]]
```
3

The following is equivalent to the above (safer) definition---and will work so long as c is not assigned to an expression.

```
Timing[Immediate[c_] =
  Integrate[Exp[Tan[x]], {x, 0, c}]]
```
4

The following should take the "least" amount of time to perform, but as we shall see is not as efficient in the long run.

```
Timing[FunctionDef[c_] :=
  Integrate[Exp[Tan[x]], {x, 0, c}]]
```
5

```
?DelayedEvaluated
?Immediate
?FunctionDef
```
6

The following should give a rapid result

```
Timing[DelayedEvaluated[0.5]]
Timing[Immediate[0.5]]
```
7

The following will not be rapid, because it has to do the symbolic integration before returning the result.

```
Timing[FunctionDef[0.5]]
```
8

1: When a non-trivial integral is done for the first time, Mathematica loads various libraries. Notice the difference in timing between this first computation of $\int \exp[\tan(x)]dx$ and the following one.

2: The second evaluation is faster. Now, a baseline time has been established for evaluating this integral symbolically.

3: Here, to make a function definition for the integral, the symbolic integral is obtained and so the function definition *takes longer* than it would if we had not used `Evaluate`.

4: Using an `=` is roughly equivalent to using `Evaluate` above and the time to make the function assignment should be approximately the same.

5: Here, the symbolic integration is delayed until the function is called (later). Therefore, the function assignment is very rapid.

6: We can use the `?`-operator to investigate the stored forms of the three function definitions. The first two forms are roughly equivalent, except for the delayed versus immediate function definition. The third form uses the unevaluated integral in the definition. symbolic information.

7: The function evaluation is much faster in the case where the symbolic integration is not needed. This would be the preferred form if the function were to be called many times.

8: The relatively slow speed of the function which contains the unevaluated integral indicates that it would be a poor choice when numerical efficiency is an issue. Therefore, if we were to use `ContourPlot`, or some other function that would need to compute the result at many different points, then the integration would be done at each point, instead of having its closed form evaluated. Thus, the function with the embedded closed form is preferable.

3.016

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualizing the Hamaker Potential of a Finite Cylinder: Contours of Constant Potential

We use the function that we have defined above as the argument to `ContourPlot`. Because the function is singular at the cylinder surface, we choose to plot the logarithm of the potential instead. Because the potential is negative outside of the cylinder, we must use an absolute value before taking the log. To remind ourselves that the potential is negative outside the cylinder, we multiply the log by minus-one.

```
LogAbsCyl[ρ_, h_, AR_] :=
 Log[Abs[LondonCylinderPotential[ρ, h, AR]]]
```
1

```
Plot[LogAbsCyl[0.0, h, 4], {h, 0, 2.5},
 Exclusions → {2.0}, PlotRange → {0, 18},
 PlotStyle → {Thick, Darker[Blue]},
 BaseStyle → {Medium}]
```
2

```
Plot[LogAbsCyl[x, 0, 1],
 {x, 0.0, 1.5}, Exclusions → {1.0},
 PlotRange → {0, 20}, PlotStyle → {Thick, Red},
 BaseStyle → {Medium}, ImageSize → Large]
```
3

```
conplotouter =
 ContourPlot[-LogAbsCyl[dist, h, 4],
  {dist, 0, 1.5}, {h, 0, 2.5},
  RegionFunction → Function[{dist, h},
    dist > 1.01 || h > 2.01],
  PlotRange → {-15, 1}, Contours → 15,
  ColorFunction → "AvocadoColors",
  AspectRatio → Automatic,
  ImageSize → Medium, Exclusions →
   {{dist == 1.0, Abs[dist - 1.0] < 0.001},
    {h == 2.0, Abs[h - 2] < 0.001}}]
```
4

```
conplotinner =
 ContourPlot[LogAbsCyl[dist, h, 4]∧],
{dist, 0, 1.5}, {h, 0, 2.5}, RegionFunction →
 Function[{dist, h}, dist < 0.99 && h < 1.99],
PlotRange → {0, 15}, Contours → 16,
ColorFunction → "LakeColors",
AspectRatio → Automatic, ImageSize → Medium,
BaseStyle -> {Medium}, AspectRatio → Automatic
```
5

```
Show[conplotinner, conplotouter]
```

6

1: We define a short-hand function to wrap around the potential function so that the $\log(|P|)$ is computed.

2–3: To get an idea of what the function looks like, we plot the potential first along the cylinder axis, and then for a distance within the mid-plane.

4: We break the contour-plots into an inner and an outer graphic. Here we use `ContourPlot` to plot (minus) the logarithm of the potential outside the cylinder. `RegionFunction` is used to to limit the region over which the plot is computed and displayed. Furthermore, the numerical integration is ill-behaved along lines that continue from the cylinder's corner; we use `Exclusions` to avoid these regions. We use a green tone, `AvocadoColors`, to indicate the negative values.

5: Here, we produce the contour-plot for the region inside the cylinder. Again, we use the `RegionFunction`-option of `ContourPlot`. We use blue tones, `LakeColors`, to indicate the positive potential values.

6: We combine the inner and outer regions into a single plot by using `Show`.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualizing the Hamaker Potential of a Finite Cylinder: Three-Dimensional Plots

We produce and equivalent visualization with `Plot3D`. From the form of these plots, it is clear that a non-polar molecule would be attracted to the cylinder, with a force that becomes unbounded in the vicinity of the cylinder. The barrier to cross into the cylinder is infinite at the cylinder surface. However, within a cylinder there is a force that pushes a foreign particle to the center of the cylinder. The Hamaker force would tend to push pores towards the middle of a dielectric cylinder.

```
plotoutside3D =
 Plot3D[-LogAbsCyl[dist, h, 4], {dist, 0, 1.5},
  {h, 0, 2.5}, RegionFunction → Function[
    {dist, h}, dist > 1.01 || h > 2.01],
  PlotRange → {-15, 1}, MeshFunctions -> {#3 &},
  ColorFunction → "AvocadoColors",
  AspectRatio → Automatic,
  ImageSize → Large, BaseStyle -> {Medium},
  AspectRatio → Automatic]
```

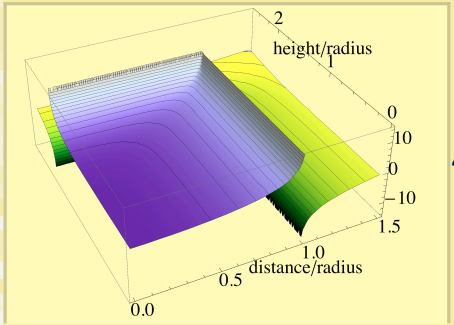
1

```
plotinside3D =
 Plot3D[LogAbsCyl[dist, h, 4], {dist, 0, 1.5},
  {h, 0, 2.5}, RegionFunction → Function[
    {dist, h}, dist < 0.99 && h < 1.99],
  PlotRange → {0, 15}, MeshFunctions -> {#3 &},
  ColorFunction → "LakeColors",
  AspectRatio → Automatic,
  ImageSize → Large, BaseStyle -> {Medium},
  AspectRatio → Automatic]
```

2

```
Show[plotinside3D,
 plotoutside3D, PlotRange → {-15, 15}]
```

3



4

1–2: `Plot3D` is used as in the previous example with the `RegionFunction` option to separate the inner- from the outer-evaluation. The `MeshFunctions` option is used to produce shading that is consistent with the contour plots in the previous example.

3: We use `Show` with an extended `PlotRange` to produce the combined three dimensional surface representing the potential as a function of distance from the axis cylinder and height above its mid-plane.

3.016 Home
Full Screen
Close
Quit

Stokes' Theorem

The final generalization of the fundamental theorem of calculus is the relation between a vector function integrated over an oriented surface and another vector function integrated over the closed curve that bounds the surface.

A simplified version of Stokes's theorem has already been discussed—Green's theorem in the plane can be written in full vector form:

$$\begin{aligned}\int\int_R \left(\frac{\partial F_2}{\partial x} - \frac{\partial F_1}{\partial y}\right) dxdy &= \int_R \nabla \times \vec{F} \cdot d\vec{A} \\ &= \oint_{\partial R} (F_1 dx + F_2 dy) = \oint_{\partial R} \vec{F} \cdot \frac{d\vec{r}}{ds} ds\end{aligned} \tag{16-6}$$

as long as the region R lies entirely in the $z =$ constant plane.

In fact, Stokes's theorem is the same as the full vector form in Eq. 16-6 with R generalized to an oriented surface embedded in three-dimensional space:

$$\int_R \nabla \times \vec{F} \cdot d\vec{A} = \oint_{\partial R} \vec{F} \cdot \frac{d\vec{r}}{ds} ds \tag{16-7}$$

Plausibility for the theorem can be obtained from Figures 16-14 and 16-15. The curl of the vector field summed over a surface "spills out" from the surface by an amount equal to the vector field itself integrated over the boundary of the surface. In other words, if a vector field can be specified everywhere for a *fixed* surface, then its integral should only depend on some vector function integrated over the boundary of the surface.

Maxwell's equations

The divergence theorem and Stokes's theorem are generalizations of integration that invoke the divergence and curl operations on vectors. A familiar vector field is the electromagnetic field and Maxwell's equations depend on these vector derivatives as

well:

$$\nabla \cdot \vec{B} = 0 \qquad \nabla \times \vec{E} = \frac{\partial \vec{B}}{\partial t}$$
$$\nabla \times \vec{H} = \frac{\partial \vec{D}}{\partial t} + \vec{j} \qquad \nabla \cdot \vec{D} = \rho \tag{16-8}$$

in MKS units and the total electric displacement $\vec{D}$ is related to the total polarization $\vec{P}$ and the electric field $\vec{E}$ through:

$$\vec{D} = \vec{P} + \epsilon_o \vec{E} \tag{16-9}$$

where ϵ_o is the dielectric permittivity of vacuum. The total magnetic induction $\vec{B}$ is related to the induced magnetic field $\vec{H}$ and the material magnetization through

$$\vec{B} = \mu_o(\vec{H} + \vec{M}) \tag{16-10}$$

where μ_o is the magnetic permeability of vacuum.

Ampere's Law

Ampere's law that relates the magnetic field lines that surround a static current is a macroscopic version of the (static) Maxwell equation $\nabla \times \vec{H} = \vec{j}$:

Gauss' Law

Gauss' law relates the electric field lines that exit a closed surface to the total charge contained within the volume bounded by the surface. Gauss' law is a macroscopic version of the Maxwell equation $\nabla \cdot \vec{D} = \rho$:

Lecture 17: Function Representation by Fourier Series

Reading:
Kreyszig Sections: 11.1, 11.2, 11.3 (pages478–485, 487–489, 490–495)

Periodic Functions

Periodic functions should be familiar to everyone. The keeping of time, the ebb and flow of tides, the patterns and textures of our buildings, decorations, and vestments invoke repetition and periodicity that seem to be inseparable from the elements of human cognition.[10] Although other species utilize music for purposes that we can only imagine—we seem to derive emotion and enjoyment from making and experience of music.

[10] I hope you enjoy the lyrical quality of the prose. While I wonder again if anyone is reading these notes, my wistfulness is taking a poetic turn:

They repeat themselves
What is here, will be there
It wills, willing, to be again
spring; neap, ebb and flow, wane; wax
sow; reap, warp and woof, motif; melody.
The changed changes. We remain
Perpetually, Immutably, Endlessly.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Playing with Audible Periodic Phenomena

Several example of creating sounds using mathematical functions are illustrated for education and amusement.

Sounds will not be available on PDF or HTML versions

Let's begin by "looking" at a familiar periodic phenomena:

We index the notes and write an indexed set of frequency (in Hertz) for each of the notes for one octave above middle-c. We write a function to create a Sound for each note.

```
c = 1; d = 2; e = 3; f = 4; g = 5; a = 6; b = 7;
freq[c] = 261.6;
freq[d]  = 293.7;
freq[e] = 329.6;
freq[f] = 349.2;
freq[g] = 392.0;
freq[a] = 440.0;
freq[b] = 493.9;
purenote[note_Integer] := purenote[note] =
  Play[Sin[2 π freq[note] t], {t, 0, 1}]
```

1

We extend the function to get simultaneous notes from a List. We use Thread which takes f[{a,b,c}] to {f[a],f[b],f[c]}

```
notes[note_List] :=
 Sound[Thread[purenote[note]]]
```

2

Here are examples of their use.

```
cnote = purenote[c]
notes[{a, c, e}]
```

3

We can play with variable amplitudes for a fixed frequency, here we can hear the increased, but non-singular amplitude through zero.

```
Plot[Sin[540 x] / x, {x, -.1, .1},
 PlotRange → All, Filling → Axis]
Play[Sin[540 x] / x, {x, -1, 1}]
```

4

We can vary amplitudes and frequencies. Warning, playing with this function can become addictive...

```
Play[
 2 Sin[20 x Sin[x Exp[-x / .2]  + Sin[x] / x]] +
  Exp[(1 + Cos[x])] Sin[x Exp[x / 10]]
   Sin[1500 x], {x, 0.01, 20}]
```

5

1: The seven musical notes around middle C indexed here with integers and then their frequencies (in hertz) are defined with a `freq`. The function *Note* takes one of the seven indexed notes and creates a wave-form for that note. The function `Play` takes the waveform and produces audio output. We introduce a function, *purenote* , that takes an integer argument and plays the corresponding exact note.

2: To play a sequence of notes, a list of notes must be passed to `Sound`. We write a function, *notes* , that uses `Thread` to create a list of object created by applications of `purenote`. In other words, `Thread[function[{la,lb,lc}]]` returns `{function[la], function[lb], function[lc]}`.

3: This is an example of the use of *note* and *purenote* .

4: This is noise generated from a function; we can modulate the amplitude and frequency. Enjoy the fact that $\sin(x)/x$ is not singular at $x = 0$.

5: This is a function that I cooked up. Enjoy.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

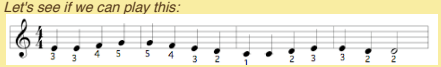
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Music and Instruments

Having no musical talent whatsoever, I try to write a program to make music.

Let's see if we can play this:



```
twoframes = {e, e, f, g, g, f, e, d, c, c, d, e};
```
1

We will play it, but it probably not what was intended...

```
notes[twoframes]
```
2

We create a rest of a fixed length, and then use Riffle to insert a rest after each note. We use a new function, called SoundNote, with None as the first argument, we get no sound.

```
purenote[rest] = SoundNote[None, .15];
```
3

```
notes[Riffle[twoframes, rest]]
```
4

SoundNote can take general strings for arguments as well, here we enter the musical notes from above (as characters), but one octave lower..

```
TwoFramesLower = {"E3", "E3", "F3", "G3", "G3",
   "F3", "E3", "D3", "C3", "C3", "D3", "E3"};
```
5

The default is to use a piano to make the sound; here we ask for a duration of 6/10 of a second.

```
piano[note_String] := SoundNote[note, .6]
```
6

```
Sound[Thread[piano[TwoFramesLower]]]
```
7

We can use other MIDI instruments as well; here is a bagpipe

```
bagpipe[note_String] :=
 SoundNote[note, .6, "Bagpipe"]
Sound[Thread[bagpipe[TwoFramesLower]]]
```
8

And, now for the birds.

```
avian[note_String] :=
 SoundNote[note, .2, "Bird"]
avian[rest] = SoundNote[None, .4];
Sound[
 Thread[avian[Riffle[TwoFramesLower, rest]]]]
```
9

1: Someone who knows how to read music told me what these notes were; so, I entered them into a list.

2: This is musical score with one-second duration notes played every 1 second. Oh, Joy.

3: This is probably not what Ludwig Van had in mind; so let's figure out how to insert a 'rest.' We use MATHEMATICA® 's built-in `SoundNote` for .15 seconds to define a *purenote* for rest `rest`.

4: `Riffle[{l1,l2,l3},x]` intersperses `x` into the list and returns `{l1,x,l2,x,l3,x}`. Calling *notes* on the resulting structure returns the pure notes with rests in between.

5: `SoundNote` will also play MIDI sounds and take string arguments for notes. We recreate a sting version of the musical score for notes one octave below middle-c.

6: By default, `SoundNote` returns a MIDI piano sound. We create a function, *piano* , to play a single note for a 0.6 second duration.

7: By using `Thread` again, we play the 12-note musical score on the piano.

8: There are other MIDI instruments; here we create the function *bagpipe* and play the score with a simulated bagpipe.

9: And finally, we introduce a 'cheep' little function, *avian* , to let the birds sing their own joy.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Random Notes and Instruments

Just because we can, let's see how sequences of random notes sound. We'll add random instruments and rests too.

Let's hear what random notes sound like: SoundNote[n] will play n semitones above middle C. Here we make a list of random notes and play them.

```
RandomNotes = Table[SoundNote[
    RandomInteger[{-15, 20}], .2], {36}];
Sound[RandomNotes, 10]
```

1

Here, we mix in some rests at random

```
RanRest[] := Module[{rdur = .2},
  If[RandomReal[] > 0.5, rdur = .4];
  SoundNote[None, rdur]]
RandomNotesandRests =
  Table[If[RandomReal[] ≥ .33,
    SoundNote[RandomInteger[{-15, 20}], .2],
    RanRest[]], {96}];
Sound[RandomNotesandRests, 20]
```

2

Now, we ask for random instruments as well, with "chords" of up to 5 instruments at each beat.

```
RandomInstruments =
  Table[If[RandomReal[] > 0.5, Table[
     SoundNote[
      Table[RandomInteger[{-15, 20}],
       {RandomInteger[{1, 5}]}],
      Round[RandomReal[{1, 2}], .2],
      RandomInteger[{1, 15}]],
     {RandomInteger[{1, 4}]}], RanRest[]],
   {48}];
Sound[RandomInstruments, 20]
```

3

Finally, some random percussion events.

```
percs = {SoundNote["Clap", 1],
   SoundNote["Sticks", 1], SoundNote["Shaker",
    1], SoundNote["LowWoodblock", 1],
   SoundNote["Castanets", 1],
   SoundNote[None, 1]};
perctable = Table[RandomChoice[percs], {50}];
Sound[perctable, 20]
```

4

1: *RandomNotes* creates a 36 member random set of single pitches from 15 semi-tones below to 20 semi-tones above middle-c. We play them for ten seconds.

2: To introduce some variety into the random melody, we write a program, *RanRest* , which will be used to introduce rests with lengths .2 and .4 with equal probability. A list, *RandomNotesandRests* , is created with random notes which call *RanRest* for approximately 1/3 of the members, and from the same random set as *RandomNotes* for the remainder.

3: Now, we introduce random MIDI instruments into our score: *RandomInstruments* is a `Table` of length 48, and each member is a list of a random number, between 1 and 5, of random instruments with random notes. These list-elements create a "chord." Rests are introduced randomly, at about 1/2 of the beats.

4: Here we play with random percussion MIDI instruments. Dancing to this is not necessarily advised.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

A function that is periodic in a single variable can be expressed as:

$$f(x+\lambda) = f(x)$$
$$f(t+\tau) = f(t) \tag{17-1}$$

The first form is a suggestion of a spatially periodic function with wavelength λ and the second form suggests a function that is periodic in time with period τ. Of course, both forms are identical and express that the function has the same value at an infinite number of points ($x = n\lambda$ in space or $t = n\tau$ in time where n is an integer.)

Specification of a periodic function, $f(x)$, within one period $x \in (x_o, x_o + \lambda)$ defines the function everywhere. The most familiar periodic functions are the trigonometric functions:

$$\sin(x) = \sin(x + 2\pi) \quad \text{and} \quad \cos(x) = \cos(x + 2\pi) \tag{17-2}$$

However, any function can be turned into a periodic function.

Using Mod to Create Periodic Functions

Periodic functions are often associated with the "modulus" operation. $\texttt{Mod}[x, \lambda]$ is the remainder of the result of *recursively* dividing x by λ until the result lies in the domain $0 \leq \text{Mod}[x, \lambda] < \lambda$). Another way to think of modulus is to find the "point" where are periodic function should be evaluated if its primary domain is $x \in (0, \lambda)$.

Mod is a very useful function that can be used to force objects to be periodic. Mod[x,λ] return that part of x that lies within 0 and λ. Or, in other words if we map the real line x to a circle with circumference λ, then Mod[x,λ] returns were x is mapped onto the circle.

```
modmatdemo[n_Integer, λ_Integer] :=
  Table[{i, Mod[i, λ]}, {i, 1, n}] //
   MatrixForm;
modcircledemo[n_Integer, λ_Integer] :=
  Module[{xpos, angle, cpos},
   Graphics[
    Table[xpos = 3 Quotient[i - 1, λ];
     angle = 2 Pi Mod[i, λ] / λ;
     cpos = {Cos[angle], Sin[angle]};
     {Circle[{xpos, 0}],
      Text[i,
       Flatten[{{xpos, 0} + 1.2 * cpos}]],
      Text[Mod[i, λ], Flatten[{{xpos, 0} +
          0.8 * cpos}]]}, {i, 1, n}]]];
```

1

```
GraphicsColumn[
 {modmatdemo[13, 5], modcircledemo[26, 5]},
 ImageSize → Full]
```

2

Boomerang uses Mod to force a function, f, with a single argument, x, to be periodic with wavelength λ

```
Boomerang[f_ , x_ , λ_ ] := f[Mod[x, λ]]
```

3

```
AFunction[x_ ] := ((3 - x) ^ 3) / 27
```

4

*The following step uses **Boomerang** to produce a periodic repetition of **AFunction** over the range 0 < x < 6:*

```
Plot[Boomerang[AFunction, x, 6],
 {x, -12, 12}, PlotRange → All]
```

5

1: We create two visualization methods to show how `Mod` works: *modmatdemo* creates a matrix with two columns `(i, Mod[i,λ])`; *modcircledemo* wraps the the counting numbers and their moduli around a `Graphics- Circle` with a λ sectors, after each circle becomes filled a new circle is created for subsequent filling. *modcircledemo* should show how `Mod` is related to mapping to a periodic domain.

2: We show both visualization demonstrations in a `GraphicsColumn`.

3: *Boomerang* uses `Mod` on the argument of any function `f` of a single argument to map the argument into the domain $(0, \lambda)$. Therefore, calling *Boomerang* on any function will create a infinitely periodic repetition of the function in the domain $(0, \lambda)$.

4: *AFunction* is created as an example to pass to *Boomerang*

5: `Plot` called on the periodic extension of wavelength $\lambda = 6$ of *AFunction* . This illustrates how *Boomerang* uses `Mod` to create a periodic function with a specified period.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

Odd and Even Functions

The trigonometric functions have the additional properties of being an *odd* function about the point $x = 0$: $f_{\text{odd}} : f_{\text{odd}}(x) = -f_{\text{odd}}(-x)$ in the case of the sine, and an *even* function in the case of the cosine: $f_{\text{even}} : f_{\text{even}}(x) = f_{\text{even}}(-x)$.

This can generalized to say that a function is even or odd about a point $\lambda/2$: $f_{\text{odd}\frac{\lambda}{2}} : f_{\text{odd}\frac{\lambda}{2}}(\lambda/2 + x) = -f_{\text{odd}\frac{\lambda}{2}}(\lambda/2 - x)$ and $f_{\text{even}\frac{\lambda}{2}} : f_{\text{even}\frac{\lambda}{2}}(\lambda/2 + x) = f_{\text{even}\frac{\lambda}{2}}(\lambda/2 - x)$.

Any function can be decomposed into an odd and even sum:

$$g(x) = g_{\text{even}} + g_{\text{odd}} \tag{17-3}$$

The sine and cosine functions can be considered the odd and even parts of the generalized trigonometric function:

$$e^{ix} = \cos(x) + \imath \sin(x) \tag{17-4}$$

with period 2π.

Representing a particular function with a sum of other functions

A Taylor expansion approximates the behavior of a suitably defined function, $f(x)$ in the neighborhood of a point, x_o, with a bunch of functions, $p_i(x)$, defined by the set of powers:

$$p_i \equiv \vec{p} = (x^0, x^1, \ldots, x^j, \ldots) \tag{17-5}$$

The polynomial that approximates the function is given by:

$$f(x) = \vec{A} \cdot \vec{p} \tag{17-6}$$

where the vector of coefficients is defined by:

$$A_i \equiv \vec{A} = (\frac{1}{0!} f(x_o), \frac{1}{1!} \frac{df}{dx}\Big|_{x_o}, \ldots, \frac{1}{j!} \frac{d^j f}{dx^j}\Big|_{x_o}, \ldots) \tag{17-7}$$

The idea of a vector of infinite length has not been formally introduced, but the idea that as the number of terms in the sum in Eq. 17-6 gets larger and larger, the approximation should converge to the function. In the limit of an infinite number of terms in the sum (or the vectors of infinite length) the series expansion will converge to $f(x)$ if it satisfies some technical continuity constraints.

However, for periodic functions, the domain over which the approximation is required is only one period of the periodic function—the rest of the function is taken care of by the definition of periodicity in the function.

Because the function is periodic, it makes sense to use functions that have the same period to approximate it. The simplest periodic functions are the trigonometric functions. If the period is λ, any other periodic function with periods $\lambda/2$, $\lambda/3$, λ/N, will also have period λ. Using these "sub-periodic" trigonometric functions is the idea behind Fourier Series.

Fourier Series

The functions $\cos(2\pi x/\lambda)$ and $\sin(2\pi x/\lambda)$ each have period λ. That is, they each take on the same value at x and $x+\lambda$.

There are an infinite number of other simple trigonometric functions that are periodic in λ; they are $\cos[2\pi x/(\lambda/2))]$ and $\sin[2\pi x/(\lambda/2))]$ and which cycle two times within each λ, $\cos[2\pi x/(\lambda/3))]$ and $\sin[2\pi x/(\lambda/3))]$ and which cycle three times within each λ, and, in general, $\cos[2\pi x/(\lambda/n))]$ and $\sin[2\pi x/(\lambda/n))]$ and which cycle n times within each λ.

The constant function, $a_0(x) = \text{const}$, also satisfies the periodicity requirement.

The superposition of multiples of any number of periodic function must also be a periodic function, therefore any function $f(x)$ that satisfies:

$$\begin{aligned} f(x) &= \mathcal{E}_0 + \sum_{n=1}^{\infty} \mathcal{E}_n \cos\left(\frac{2\pi n}{\lambda}x\right) + \sum_{n=1}^{\infty} \mathcal{O}_n \sin\left(\frac{2\pi n}{\lambda}x\right) \\ &= \mathcal{E}_{k_0} + \sum_{n=1}^{\infty} \mathcal{E}_{k_n} \cos(k_n x) + \sum_{n=1}^{\infty} \mathcal{O}_{k_n} \sin(k_n x) \end{aligned} \tag{17-8}$$

where the k_i are the *wave-numbers* or *reciprocal wavelengths* defined by $k_j \equiv 2\pi j/\lambda$. The k's represent inverse wavelengths—

3.016 Home
Full Screen
Close
Quit

large values of k represent short-period or high-frequency terms.

If any periodic function $f(x)$ could be represented by the series in in Eq. 17-8 by a suitable choice of coefficients, then an alternative representation of the periodic function could be obtained in terms of the simple trigonometric functions and their amplitudes.

The "inverse question" remains: "How are the amplitudes $\mathcal{E}_{k_n}$ (the even trigonometric terms) and $\mathcal{O}_{k_n}$ (the odd trigonometric terms) determined for a given $f(x)$?"

The method follows from what appears to be a "trick." The following three integrals have simple forms for integers M and N:

$$\begin{aligned}
&\int_{x_0}^{x_0+\lambda} \sin\left(\frac{2\pi M}{\lambda}x\right)\sin\left(\frac{2\pi N}{\lambda}x\right)dx = \begin{cases} \frac{\lambda}{2} \text{ if } M = N \\ 0 \text{ if } M \neq N \end{cases} \\
&\int_{x_0}^{x_0+\lambda} \cos\left(\frac{2\pi M}{\lambda}x\right)\cos\left(\frac{2\pi N}{\lambda}x\right)dx = \begin{cases} \frac{\lambda}{2} \text{ if } M = N \\ 0 \text{ if } M \neq N \end{cases} \\
&\int_{x_0}^{x_0+\lambda} \cos\left(\frac{2\pi M}{\lambda}x\right)\sin\left(\frac{2\pi N}{\lambda}x\right)dx = 0 \text{ for any integers } M, N
\end{aligned} \tag{17-9}$$

The following shows a demonstration of this *orthogonality relation for the trigonometric functions.*

Orthogonality of Trigonometric Functions

This is a Demonstration that the relations in Eq. 17-9 are true.

```
fassume = {Minteger ∈ Integers,
  Ninteger ∈ Integers, xo ∈ Reals, λ > 0}
coscos = Integrate[
  Cos[2 π Minteger x / λ] Cos[2 π Ninteger x / λ] ,
  {x, xo, xo + λ}, Assumptions → fassume]
```
1

Demonstrating $\int_{xo}^{xo+\lambda} \cos(2\,m\pi x/\lambda)\cos(2\,n\pi x/\lambda)\,dx = 0$ *for* $m \neq n$

```
Table[
 {{mrand, nrand} = RandomInteger[{1, 50}, 2]},
  Simplify[coscos /. {Minteger → mrand ,
     Ninteger → nrand}]}, {20}]
```
2

when n=m give indeterminate values, for these we should use a Limit.

```
Limit[coscos, Minteger → Ninteger,
 Assumptions → fassume]
```
3

```
cossin = Integrate[
  Cos[2 π Minteger x / λ] Sin[2 π Ninteger x / λ] ,
  {x, xo, xo + λ}, Assumptions → fassume]
```
4

```
Table[
 {{mrand, nrand} = RandomInteger[{1, 50}, 2]},
  Simplify[cossin /. {Minteger → mrand ,
     Ninteger → nrand}]}, {20}]
```
5

```
Limit[cossin, Minteger → Ninteger,
 Assumptions → fassume]
```
6

```
sinsin = Integrate[
  Sin[2 π Minteger x / λ] Sin[2 π Ninteger x / λ] ,
  {x, xo, xo + λ}, Assumptions → fassume]
```
7

```
Table[
 {{mrand, nrand} = RandomInteger[{1, 50}, 2]},
  Simplify[sinsin /. {Minteger → mrand ,
     Ninteger → nrand}]}, {20}]
```
8

```
Limit[sinsin, Minteger → Ninteger,
 Assumptions → fassume]
```
9

1: Using `Integrate` for $\cos(2\pi Mx/\lambda)\cos(2\pi Nx/\lambda)$ over a definite interval of a single wavelength, does not produce a result that obviously vanishes for $M \neq N$.

2: However, random replacement of the symbolic integers with integers results in a zero. So, one the orthogonality relation is plausible.

3: Using `Assuming` and `Limit`, one can show that the relation ship vanishes for $N = M$. Although, it is a bit odd to be use continuous limits with integers.

4–6: This shows the same process for $\int \cos(2\pi Mx/\lambda)\sin(2\pi Nx/\lambda)dx$, which always returns zeroes.

7–9: And, $\int \sin(2\pi Mx/\lambda)\sin(2\pi Nx/\lambda)dx$ returns zeroes unless $M = N$.

3.016 Home

Full Screen

Close

Quit

Using this orthogonality trick, any amplitude can be determined by multiplying both sides of Eq. 17-8 by its conjugate trigonometric function and integrating over the domain. (Here we pick the domain to start at zero, $x \in (0, \lambda)$, but any other starting point would work fine.)

$$\begin{aligned}
\cos(k_M x) f(x) &= \cos(k_M x) \left(\mathcal{E}_{k_0} + \sum_{n=1}^{\infty} \mathcal{E}_{k_n} \cos(k_n x) + \sum_{n=1}^{\infty} \mathcal{O}_{k_n} \sin(k_n x) \right) \\
\int_0^{\lambda} \cos(k_M x) f(x) dx &= \int_0^{\lambda} \cos(k_M x) \left(\mathcal{E}_{k_0} + \sum_{n=1}^{\infty} \mathcal{E}_{k_n} \cos(k_n x) + \sum_{n=1}^{\infty} \mathcal{O}_{k_n} \sin(k_n x) \right) dx \qquad (17\text{-}10) \\
\int_0^{\lambda} \cos(k_M x) f(x) dx &= \frac{\lambda}{2} \mathcal{E}_{k_M}
\end{aligned}$$

This provides a formula to calculate the even coefficients (amplitudes) and multiplying by a sin function provides a way to calculate the odd coefficients (amplitudes) for $f(x)$ periodic in the fundamental domain $x \in (0, \lambda)$.

$$\begin{aligned}
\mathcal{E}_{k_0} &= \frac{1}{\lambda} \int_0^{\lambda} f(x) dx \\
\mathcal{E}_{k_N} &= \frac{2}{\lambda} \int_0^{\lambda} f(x) \cos(k_N x) dx \qquad k_N \equiv \frac{2\pi N}{\lambda} \qquad (17\text{-}11) \\
\mathcal{O}_{k_N} &= \frac{2}{\lambda} \int_0^{\lambda} f(x) \sin(k_N x) dx \qquad k_N \equiv \frac{2\pi N}{\lambda}
\end{aligned}$$

The constant term has an extra factor of two because $\int_0^{\lambda} \mathcal{E}_{k_0} dx = \lambda \mathcal{E}_{k_0}$ instead of the $\lambda/2$ found in Eq. 17-9.

Other forms of the Fourier coefficients

Sometimes the primary domain is defined with a different starting point and different symbols, for instance Kreyszig uses a centered domain by using $-L$ as the starting point and $2L$ as the period, and in these cases the forms for the Fourier coefficients look a bit different. One needs to look at the domain in order to determine which form of the formulas to use.

Extra Information and Notes

Potentially interesting but currently unnecessary

The "trick" of multiplying both sides of Eq. 17-8 by a function and integrating comes from the fact that the trigonometric functions form an orthogonal basis *for functions with inner product defined by*

$$f(x) \cdot g(x) = \int_0^\lambda f(x)g(x)dx$$

Considering the trigonometric functions as components of a vector:

$$\begin{aligned}
\vec{e_0}(x) &= (1, 0, 0, \dots,) \\
\vec{e_1}(x) &= (0, \cos(k_1 x), 0, \dots,) \\
\vec{e_2}(x) &= (0, 0, \sin(k_1 x), \dots,) \\
\dots &= \quad \vdots \\
\vec{e_n}(x) &= (\dots\dots, \sin(k_n x), \dots,)
\end{aligned}$$

then these "basis vectors" satisfy $\vec{e_i} \cdot \vec{e_j} = (\lambda/2)\delta_{ij}$, *where* $\delta_{ij} = 0$ *unless* $i = j$. *The trick is just that, for an arbitrary function represented by the basis vectors,* $\vec{P}(x) \cdot \vec{e_j}(x) = (\lambda/2)P_j$.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Calculating Fourier Series Amplitudes

Functions are developed which compute the even (cosine) amplitudes and odd (sine) amplitudes for an input function of one variable. These functions are extended to produce the first N terms of a Fourier series.

First we will "do it the hard way" and write short programs that evaluate Fourier coefficients; then we will demonstrate how to make use of built-in functions in Mathematica's ***FourierTransform*** *package...*

Define functions based on the formulas derived for the fourier amplitudes

The constant term:

```
EvenTerms[0, function_ , λ_] :=
 1/λ ∫_0^λ function[dum] ddum
```
1

A function that defines each even amplitude individually (this is not very efficient, it would be better to evaluate the integral once and use that result)

```
EvenTerms[SP_Integer, function_ , λ_] :=
 EvenTerms[SP, function , wavelength] =
  2/λ ∫_0^λ function[dum] Cos[2 SP π dum/λ] ddum
```
2

Define the zeroth odd term as zero for symmetry with the even terms:

```
OddTerms[0, function_ , λ_] := 0
```
3

```
OddTerms[SP_Integer, function_ , λ_] :=
 OddTerms[SP, function , λ] =
  2/λ ∫_0^λ function[dum] Sin[2 SP π dum/λ] ddum
```
4

A function to create a vector of amplitudes for the odd terms and one for the even terms

```
OddAmplitudeVector[
  NTerms_Integer, function_, λ_] :=
 Table[OddTerms[i, function, λ],
  {i, 0, NTerms}]
```
5

```
EvenAmplitudeVector[
  NTerms_Integer, function_, λ_] :=
 Table[EvenTerms[i, function, λ],
  {i, 0, NTerms}]
```
6

1–2: *EvenTerms* computes symbolic representations of the even (cosine) coefficients using the formulas in Eq. 17-11. The $N = 0$ term is computed with a supplemental definition because of its extra factor of 2. The domain is chosen so that it begins at $x = 0$ and ends at $x = \lambda$.

3–4: *OddTerms* performs a similar computation for the sine-coefficients; the $N = 0$ amplitude is set to zero explicitly. It will become convenient to include the zeroth-order coefficient for the odd (sine) series which vanishes by definition. *The functions work by doing an integral for each term—this is not very efficient. It would be more efficient to calculate the integral symbolically once and then evaluate it once for each term.*

5–6: *OddAmplitudeVector* and *EvenAmplitudeVectors* create *amplitude vectors* for the cosine and sine terms with specified lengths and domains.

5: This function, $f(x) = x(1-x)^2(2-x)$, will be used for particular examples of Fourier series, note that it is an even function over $0 < x < 2$.

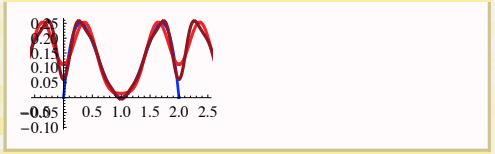
3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Approximations to Functions with Truncated Fourier Series

Example of using Eq. 17-11 to calculate a Fourier Series for a particular function.

```
myfunction[x_] := (x * (2 - x) * (1 - x)^2)                                    1

OriginalPlot = Plot[myfunction[x], {x, 0, 2},                                  2
  PlotStyle → {Hue[.66], Thickness[0.015]}]

EvenAmplitudeVector[6, myfunction, 2]                                          3

OddAmplitudeVector[6, myfunction, 2]                                           4

OddBasisVector[NTerms_Integer, var_, λ_] :=                                    5
 Table[Sin[2 π i var / λ], {i, 0, NTerms}]

OddBasisVector[6, x, 2]                                                        6

EvenBasisVector[NTerms_Integer, var_, λ_] :=                                   7
 Table[Cos[2 π i var / λ], {i, 0, NTerms}]

EvenBasisVector[6, x, 2]                                                       8

FourierTruncSeries[n_, function_, var_,
 λ_] := EvenAmplitudeVector[n, function, λ].                                   9
   EvenBasisVector[n, var, λ] +
  OddAmplitudeVector[n, function, λ].
   OddBasisVector[n, var, λ]

FourierTruncSeries[6, myfunction, x, 2]                                        10

FPlot[n_Integer] := FPlot[n] = Plot[Evaluate[
    FourierTruncSeries[n, myfunction, x, 2]],                                  11
  {x, -2, 4}, PlotStyle →
   {Thick, ColorData[n, "ColorList"]}]

Show[OriginalPlot, FPlot[3], FPlot[6],                                         12
 PlotRange → {{-0.5, 2.5}, {-0.1, 0.26}}]
```



1: We introduce and example function $x(2-x)(1-x^2)$ that vanishes at $x = 0, 1, 2$ that will be used to produce a periodic function with $\lambda = 2$item We store the example function's graphical representation in `OriginalPlot`. Note that there will be a sharp discontinuity in the derivative at the edges of the periodic domain.

3–4: The Fourier coefficients, truncated at six terms, are computed with the functions that we defined above, *OddAmplitudeVector* and *EvenAmplitudeVector* . Note that because of the even symmetry of the function about the middle, all of the odd coefficients vanish.

5–8: *OddBasisVector* and *EvenBasisVector* , create vectors of *basis functions* of specified lengths and periodic domains.

9–10: The Fourier series up to a certain order can be defined as the sum of two inner (dot) products: the inner product of the odd coefficient vector and the sine basis vector, and the inner product of the even coefficient vector and the cosine basis vector.

11–12: This will illustrate the approximation for a truncated ($N = 6$) Fourier series. *FPlot* takes an integer truncation-argument and generates a plot for that truncation, but only for `myfunction`. It might be useful to extend this example so that it takes a function as an argument, but it makes more sense to leave this example and use MATHEMATICA® 's built-in Fourier series methods.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Demonstration the used of functions defined in the `FourierSeries`-package

Fourier series expansions are a common and useful mathematical tool, and it is not surprising that MATHEMATICA® would have a package to do this and replace the inefficient functions defined in the previous example.

```
Needles["FourierSeries`"]
```
1

```
AFunction[x_] := (x - 3)^3 / 27
```
2

```
Plot[AFunction[x], {x, 0, 6}]
```
3

Mathematica's Fourier Series functions are defined for function that are periodic in the domain x ∈ (-1/2,1/2). So we need to map the periodic functions to this domain

```
ReduceHalfHalf[f_, x_, λ_] :=
 f[(x + 1/2) * λ]
```
4

```
ReducedFunction =
 ReduceHalfHalf[AFunction, x, 6] // Simplify
```
5

$$8\,x^3$$

```
ExactPlot = Plot[ReducedFunction,
  {x, -1/2, 1/2}, PlotRange → All, PlotStyle →
   {Red, Opacity[0.5], Thickness[0.01]}]
```
6

```
FourierCosCoefficient[ReducedFunction, x, n]
```
7

```
FourierSinCoefficient[ReducedFunction, x, n]
```
8

$$\frac{2\,(-1)^n\,\left(6 - n^2\,\pi^2\right)}{n^3\,\pi^3}$$

```
FourierTrigSeries[ReducedFunction, x, 5]
```
9

$$\frac{2\,\left(-6 + \pi^2\right)\,\text{Sin}[2\,\pi\,x]}{\pi^3} + \frac{\left(3 - 2\,\pi^2\right)\,\text{Sin}[4\,\pi\,x]}{2\,\pi^3} + \frac{2\,\left(-2 + 3\,\pi^2\right)\,\text{Sin}[6\,\pi\,x]}{9\,\pi^3} + \frac{\left(3 - 8\,\pi^2\right)\,\text{Sin}[8\,\pi\,x]}{16\,\pi^3} + \frac{2\,\left(-6 + 25\,\pi^2\right)\,\text{Sin}[10\,\pi\,x]}{125\,\pi^3}$$

1: The functions in `FourierSeries` to operate on the unit period located at $x \in (-1/2, 1/2)$ by default. Therefore, the domains of functions of interest can be mapped onto this domain by a change of variables.

2–3: We introduce another function that will be approximated by a Fourier series. This function will be made periodic with $\lambda = 6$ in the untransformed variables.

4–6: *ReduceHalfHalf* is an example of a function design to do the required mapping. First the length of original domain is mapped to unity by dividing through by λ and then the origin is shifted by mapping the x (that the MATHEMATICA® functions will see) to $(-1/2, 1/2)$ with the transformation $x \to x + \frac{1}{2}$. *ReducedFunction* shows an example on the function defined above.

8–9: Particular amplitudes of the properly remapped function can be obtained with the functions `FourierCosCoefficient` and `FourierSinCoefficient`. In this example, a symbolic n is entered and a symbolic representation of the n^{th} amplitude is returned. Because the function is odd about the middle, all of the cosine-coefficients are zero.

9: A truncated Fourier series can be obtained symbolically to any order with `FourierTrigSeries`.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Recursive Calculation of a Truncated Fourier Series

In this example, we build up a set of recursive function that will be utilized for efficient computation of a truncated Fourier series. These functions will be used in a subsequent visualization example.

```
ManipulateTruncatedFourierSeries[function_,
   {truncationstart_, truncationend_,
    truncjump_}] := Manipulate[Plot[Evaluate[
     FourierTrigSeries[function, x, itrunc]],
    {x, -1, 1}, PlotRange → {-2, 2}],
   {itrunc, {truncationstart,
     truncationend, truncjump}}];
```

1

The function above will work, but it is horribly inefficient! Because it asks FourierTrigSeries to calculate one more term each time, it is doing some redundant work. We can fix this up by having it calculate one new term and adding to the sum calculated previously. Here it is:

```
costerm[function_, x_, n_Integer] :=
 Simplify[FourierCosCoefficient[
    function, x, n]] Cos[2 π n x]
sinterm[function_, x_, n_Integer] :=
 Simplify[FourierSinCoefficient[
    function, x, n]] Sin[2 π n x]

TruncatedFourierSeries[function_, x_, 0] :=
 TruncatedFourierSeries[function, x, 0] =
  costerm[function, x, 0] +
   sinterm[function, x, 0]

TruncatedFourierSeries[
  function_, x_, n_Integer] :=
 TruncatedFourierSeries[function, x, n] =
  TruncatedFourierSeries[function, x, n - 1] +
   costerm[function, x, n] +
   sinterm[function, x, n]
```

2

1: *ManipulateTruncatedFourierSeries* is a simple example of visualization function for the truncated Fourier series. It uses the `Manipulate` function with three arguments in the iterator for the initial truncation `truncationstart`, final truncation, and the number to skip in between.

2: However, because the entire series is recomputed for each frame, the function above is not very efficient. In this second version, only two arguments are supplied to the iterator. At each function call, the two N^{th} Fourier terms are added to those computed in the $(N-1)^{th}$ and then stored in memory. The recursion stops at the defined $N = 0$ term.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

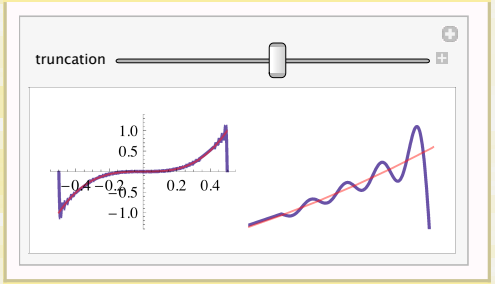
Visualizing Convergence of the Fourier Series: Gibbs Phenomenon

Functions that produce visualizations with `Manipulate` (each frame representing a different order of truncation of the Fourier series) are developed. This example illustrates *Gibbs phenomenon* where the approximating function oscillates wildly near discontinuities in the original function. In the `Manipulate` function, we use the option `Initialization` so that all evaluations during graphical output will be rapid.

The following will demonstrate how convergence is difficult where the function changes rapidly---this is known as Gibbs' Phenomenon

```
Manipulate[GraphicsRow[
  {plt = Show[Plot[theapprx[truncation],
      {x, -0.4999, 0.4999},
      PlotRange → {{-0.55, 0.55}, {-1.4, 1.4}},
      PlotStyle → {Thick, ColorData[1,
         truncation]}], ExactPlot], Show[plt,
    PlotRange → {{0.4, 0.5}, {0.5, 1.2}}]}],
 ImageSize → Full],
{{truncation, 100},
 1, 100, 1},
Initialization → (Table[
    theapprx[i] = TruncatedFourierSeries[
      ReducedFunction, x, i], {i, 1, 100}];)]
```

1



1: Because *ReducedFunction* has a discontinuity (its end-value and its initial-value differ), this visualization will show Gibbs phenomena near the edges of the domain. The approximation is fine everywhere except in the neighborhood of the discontinuity. At the discontinuity, the oscillations about the exact value do not dampen out with increased truncation N, but the domain where the oscillations are ill-behaved shrinks with increased N.

3.016 Home
Full Screen
Close
Quit

Complex Form of the Fourier Series

The behavior of the Fourier coefficients for both the odd (sine) and for the even (cosine) terms was illustrated above. Functions that are even about the center of the fundamental domain (reflection symmetry) will have only even terms—all the sine terms will vanish. Functions that are odd about the center of the fundamental domain (reflections across the center of the domain and then across the x-axis.) will have only odd terms—all the cosine terms will vanish.

Functions with no odd or even symmetry will have both types of terms (odd and even) in its expansion. This is a restatement of the fact that any function can be decomposed into odd and even parts (see Eq. 17-3).

This suggests a short-hand in Eq. 17-4 can be used that combines both odd and even series into one single form. However, because the odd terms will all be multiplied by the imaginary number $\imath$, the coefficients will generally be complex. Also because $\cos(nx) = (\exp(inx) + \exp(-inx))/2$, writing the sum in terms of exponential functions only will require that the sum must be over both positive and negative integers.

For a periodic domain $x \in (0, \lambda)$, $f(x) = f(x + \lambda)$, the *complex form of the fourier series is given by:*

$$f(x) = \sum_{n=-\infty}^{\infty} \mathcal{C}_{k_n} e^{\imath k_n x} \qquad \text{where } k_n \equiv \frac{2\pi n}{\lambda}$$
$$\mathcal{C}_{k_n} = \frac{1}{\lambda} \int_0^{\lambda} f(x) e^{-\imath k_n x} dx \tag{17-12}$$

Because of the orthogonality of the basis functions $\exp(\imath k_n x)$, the domain can be moved to any wavelength, the following is also true:

$$f(x) = \sum_{n=-\infty}^{\infty} \mathcal{C}_{k_n} e^{\imath k_n x} \qquad \text{where } k_n \equiv \frac{2\pi n}{\lambda}$$
$$\mathcal{C}_{k_n} = \frac{1}{\lambda} \int_{-\lambda/2}^{\lambda/2} f(x) e^{-\imath k_n x} dx \tag{17-13}$$

although the coefficients may have a different form.

Lecture 18: The Fourier Transform and its Interpretations

Reading:
Kreyszig Sections: 11.4, 11.7, 11.8, 11.9 (pages496–498, 506–512 513–517, 518–523)

Fourier Transforms

Expansion of a function in terms of Fourier Series proved to be an effective way to represent functions that were periodic in an interval $x \in (-\lambda/2, -\lambda/2)$. Useful insights into "what makes up a function" are obtained by considering the amplitudes of the harmonics (i.e., each of the sub-periodic trigonometric or complex oscillatory functions) that compose the Fourier series. That is, the component harmonics can be quantified by inspecting their amplitudes. For instance, one could quantitatively compare the same note generated from a Stradivarius to an ordinary violin by comparing the amplitudes of the Fourier components of the notes component frequencies.

However there are many physical examples of phenomena that involve nearly, but not completely, periodic phenomena—and of course, quantum mechanics provides many examples of isolated events that are composed of wave-like functions.

It proves to be very useful to extend the Fourier analysis to functions that are not periodic. Not only are the same interpretations of contributions of the elementary functions that compose a more complicated object available, but there are many others to be obtained.

For example:

momentum/position The wavenumber $k_n = 2\pi n/\lambda$ turns out to be proportional to the momentum in quantum mechanics. The position of a function, $f(x)$, can be expanded in terms of a series of wave-like functions with amplitudes that depend

on each component momentum—this is the essence of the Heisenberg uncertainty principle.

diffraction Bragg's law, which formulates the conditions of constructive and destructive interference of photons diffracting off of a set of atoms, is much easier to derive using a Fourier representation of the atom positions and photons.

To extend Fourier series to non-periodic functions, the domain of periodicity will extended to infinity, that is the limit of $\lambda \to \infty$ will be considered. This extension will be worked out in a heuristic manner in this lecture—the formulas will be correct, but the rigorous details are left for the math textbooks.

Recall that the complex form of the Fourier series was written as:

$$\begin{aligned} f(x) &= \sum_{n=-\infty}^{\infty} \mathcal{A}_{k_n} e^{\imath k_n x} \qquad \text{where } k_n \equiv \frac{2\pi n}{\lambda} \\ \mathcal{A}_{k_n} &= \frac{1}{\lambda} \int_{-\lambda/2}^{\lambda/2} f(x) e^{-\imath k_n x} dx \end{aligned} \tag{18-1}$$

where $\mathcal{A}_{k_n}$ is the complex amplitude associated with the $k_n = 2\pi n/\lambda$ reciprocal wavelength or wavenumber.

This can be written in a more symmetric form by scaling the amplitudes with λ—let $\mathcal{A}_{k_n} = \sqrt{2\pi}\mathcal{C}_{k_n}/\lambda$, then

$$\begin{aligned} f(x) &= \sum_{n=-\infty}^{\infty} \frac{\sqrt{2\pi}\mathcal{C}_{k_n}}{\lambda} e^{\imath k_n x} \qquad \text{where } k_n \equiv \frac{2\pi n}{\lambda} \\ \mathcal{C}_{k_n} &= \frac{1}{\sqrt{2\pi}} \int_{-\lambda/2}^{\lambda/2} f(x) e^{-\imath k_n x} dx \end{aligned} \tag{18-2}$$

Considering the first sum, note that the difference in wave-numbers can be written as:

$$\Delta k = k_{n+1} - k_n = \frac{2\pi}{\lambda} \tag{18-3}$$

which will become infinitesimal in the limit as $\lambda \to \infty$. Substituting $\Delta k/(2\pi)$ for $1/\lambda$ in the sum, the more "symmetric result"

appears,

$$f(x) = \frac{1}{\sqrt{2\pi}} \sum_{n=-\infty}^{\infty} \mathcal{C}_{k_n} e^{\imath k_n x} \Delta k \qquad \text{where } k_n \equiv \frac{2\pi n}{\lambda}$$
$$\mathcal{C}_{k_n} = \frac{1}{\sqrt{2\pi}} \int_{-\lambda/2}^{\lambda/2} f(x) e^{-\imath k_n x} dx \tag{18-4}$$

Now, the limit $\lambda \to \infty$ can be obtained an the summation becomes an integral over a continuous spectrum of wave-numbers; the amplitudes become a continuous function of wave-numbers, $\mathcal{C}_{k_n} \to g(k)$:

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} g(k) e^{\imath k x} dk$$
$$g(k) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) e^{-\imath k x} dx \tag{18-5}$$

The function $g(k = 2\pi/\lambda)$ represents the density of the amplitudes of the periodic functions that make up $f(x)$. The function $g(k)$ is called *the Fourier Transform* of $f(x)$. The function $f(x)$ is called *the Inverse Fourier Transform* of $g(k)$, and $f(x)$ and $g(k)$ are a *the Fourier Transform Pair*.

Higher Dimensional Fourier Transforms

Of course, many interesting periodic phenomena occur in two dimensions (e.g., two spatial dimensions, or one spatial plus one temporal), three dimensions (e.g., three spatial dimensions or two spatial plus one temporal), or more.

The Fourier transform that integrates $\frac{dx}{\sqrt{2\pi}}$ over all x can be extended straightforwardly to a two dimensional integral of a function $f(\vec{r}) = f(x, y)$ by $\frac{dxdy}{2\pi}$ over all x and y—or to a three-dimensional integral of $f(\vec{r}) \frac{dxdydz}{\sqrt{(2\pi)^3}}$ over an infinite three-dimensional volume.

A wavenumber appears for each new spatial direction and they represent the periodicities in the x-, y-, and z-directions. It

is natural to turn the wave-numbers into a **wave-vector**

$$\vec{k} = (k_x, k_y, k_z) = (\frac{2\pi}{\lambda_x}, \frac{2\pi}{\lambda_y}, \frac{2\pi}{\lambda_y}) \tag{18-6}$$

where λ_i is the wavelength of the wave-function in the i^{th} direction.

The three dimensional Fourier transform pair takes the form:

$$\begin{aligned} f(\vec{x}) =& \frac{1}{\sqrt{(2\pi)^3}} \iiint_{-\infty}^{\infty} g(\vec{k}) e^{\imath \vec{k}\cdot\vec{x}} dk_x dk_y dk_z \\ g(\vec{k}) =& \frac{1}{\sqrt{(2\pi)^3}} \iiint_{-\infty}^{\infty} f(\vec{x}) e^{-\imath \vec{k}\cdot\vec{x}} dx dy dz \end{aligned} \tag{18-7}$$

Properties of Fourier Transforms

Dirac Delta Functions

Because the inverse transform of a transform returns the original function, this allows a definition of an interesting function called the Dirac delta function $\delta(x-x_o)$. Combining the two equations in Eq. 18-5 into a single equation, and then interchanging the order of integration:

$$\begin{aligned} f(x) =& \frac{1}{2\pi} \int_{-\infty}^{\infty} \left\{ \int_{-\infty}^{\infty} f(\xi) e^{-\imath k \xi} d\xi \right\} e^{\imath k x} dk \\ f(x) =& \int_{-\infty}^{\infty} f(\xi) \left\{ \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{\imath k (x-\xi)} dk \right\} d\xi \end{aligned} \tag{18-8}$$

Apparently, a function can be defined

$$\delta(x - x_o) = \frac{1}{2\pi} \int_{-\infty}^{\infty} e^{\imath k (x-\xi)} dk \tag{18-9}$$

that has the property

$$f(x_o) = \int_{-\infty}^{\infty} \delta(x - x_o) f(x) dx \tag{18-10}$$

in other words, δ picks out the value at $x = x_o$ and returns it outside of the integration.

Parseval's Theorem

The delta function can be used to derive an important *conservation theorem*.

If $f(x)$ represents the density of some function (i.e., a wave-function like $\psi(x)$), the square-magnitude of f integrated over all of space should be the total amount of material in space.

$$\int_{-\infty}^{\infty} f(x)\bar{f}(x)dx = \int_{-\infty}^{\infty} \left\{ \left(\frac{1}{\sqrt{2\pi}} g(k) e^{-\imath kx} dk \right) \left(\frac{1}{\sqrt{2\pi}} \bar{g}(\kappa) e^{-\imath \kappa x} d\kappa \right) \right\} dx \tag{18-11}$$

where the complex-conjugate is indicated by the over-bar. This exponentials can be collected together and the definition of the δ-function can be applied and the following simple result can is obtained

$$\int_{-\infty}^{\infty} f(x)\bar{f}(x)dx = \int_{-\infty}^{\infty} g(k)\bar{g}(k)dk = \tag{18-12}$$

which is Parseval's theorem. It says, that the magnitude of the wave-function, whether it is summed over real space or over momentum space must be the same.

Convolution Theorem

The *convolution* of two functions is given by

$$F(x) = p_1(x) \star p_2(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} p_1(\eta) p_2(x - \eta) d\eta \tag{18-13}$$

3.016 Home

Full Screen

Close

Quit

If p_1 and p_2 can be interpreted as densities in probability, then this convolution quantity can be interpreted as "the total joint probability due to two probability distributions whose arguments add up to x."[11]

The proof is straightforward that the convolution of two functions, $p_1(x)$ and $p_2(x)$, is a Fourier integral over the product of their Fourier transforms, $\psi_1(k)$ and $\psi_2(k)$:

$$p_1(x) \star p_2(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} p_1(\eta) p_2(x-\eta) d\eta = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \psi_1(k)\psi_2(k) e^{\imath kx} dk \tag{18-14}$$

This implies that Fourier transform of a convolution is a direct product of the Fourier transforms $\psi_1(k)\psi_2(k)$.

Another way to think of this is that "the net effect on the spatial function due two interfering waves is contained by product the fourier transforms." Practically, if the effect of an aperture (i.e., a sample of only a finite part of real space) on a wave-function is desired, then it can be obtained by multiplying the Fourier transform of the aperture and the Fourier transform of the entire wave-function.

[11] To think this through with a simple example, consider the probability that two dice sum up 10. It is the sum of $p_1(n)p_2(10-n)$ over all possible values of n.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Creating Images of Lattices for Subsequent Fourier Transform

A matrix of intensities (or, the density of scattering objects) is created as a set of "pixels" for imaging. We will use data like this to simulate

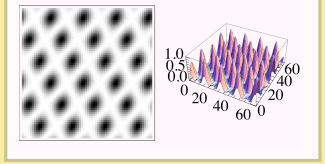
Here we create an image simulating what might be seen in an electron microscope. We will use this data to perform simulated diffraction through use of Fourier Transforms.

```
ISize = 64;
AtomDensity =
  N[Table[(1 + Sin[4 (x + y)
          2 Pi / ISize])
      (1 + Sin[2 (x - 2 y) 2
           Pi / ISize]) / 4,
    {x, 1, ISize}, {y,
     1, ISize}]];
```

1

```
GraphicsRow[
 {ArrayPlot[AtomDensity],
  ListPlot3D[AtomDensity]},
 ImageSize → Full]
```

2



1: `Table` to form a discrete set of points that we will use to approximate am image which image such as might be seen in an transmission electron microscope. For our first set of data, we use interference of two sine waves to produce a simulation of the density of scattering centers in an atomic lattice. Most of the physical aspects of atomic imaging and diffraction can be simulated with the two-dimensional techniques that are produced in these notes. We start with a 64×64 set of discrete points, this is fairly small but it will produce fairly computationally inexpensive results.

2: `ArrayPlot` produces a gray-scale image from an array of "pixel values" between 0 (black) and 1 (white); we use Plot3D to get an additional visualization of the density of scatterers.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

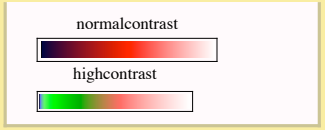
Improving Visualization Contrast with `ColorFunction`

Two examples of `ColorFunction`, *normalcontrast* and *highcontrast* are produced that aid in the interpretation of simulated data. The first, *normalcontrast* , provides a way to use a color, red, to interpolate between black at low intensities and white at high intensities. The second, *highcontrast* , compresses the color change at the low-intensity end; this provides a means to visualize "noise" at low intensities.

To view the data better, we create two versions of a contrast function, the first *(normalcontrast)* is useful when we wish to view the entire range of intensities, the second version *(highcontrast)* when we wish to resolve differences at the low-end of the intensities.

A

`ContrastGraphics` **4**



A: Three input expressions are not shown, but available in the links provided above. The first two define the two *Pure Functions* that will be used at the `ColorFunction` option to graphics objects. The third produces a scale that relates the colors to the intensities.

4: *ContrastGraphics* , defined in item A, shows the relation of colors to intensity.

ImagePlot

This will be our swiss-army knife visualizer for atomic image and diffraction image graphics.

```
ImagePlot[data_?MatrixQ,
  label_:None,
  colfunc_:highcontrast,
  imagesize_:Medium] :=
 Module[{absdata =
    Abs[data], min, max},
  ArrayPlot[absdata,
   ColorFunction→colfunc,
   BaseStyle→
    {Tiny, FontFamily→
      "Helvetica"},
   PlotLabel→label,
   ImageSize→imagesize]]
```

1

1: *ImagePlot* takes a rectangular array of (possibly complex-valued) intensities are produces graphics from them. It takes three optional functions for the `PlotLabel`, `ColorFunction`, and `ImageSize` which will have default values if not given. It uses `Abs` to find the magnitude of each pixel and then `ArrayPlot` to visualize it. Note, the units of the graphics are the number of pixels along the horizontal and vertical edges.

The fast fourier transform (FFT) is a very fast algorithm for compute discrete Fourier transforms (DFT) (i.e., the Fourier transform of a data set) and is widely used in the physical sciences. For image data, the Fourier transform is the diffraction pattern (i.e., the intensity of reflected waves from a set of objects, the pattern results from positive or negative reinforcement or coherence).

However, for FFT simulations of the diffraction pattern from an image, the question arises on what to do with the rest of space *which is not the original image*. In other words, the Fourier transform is taken over all space, but the image is finite. In the examples that follow, the rest of space is occupied by *periodic duplications* of the original image. Thus, because the original image is rectangular, there will always be an additional rectangular symmetry in the diffraction pattern due to scattering from the duplicate features in the neighboring images.

The result of a discrete Fourier transform is a also a discrete set. There are a finite number of pixels in the data, the same finite number of sub-periodic wave-numbers. In other words, the Discrete Fourier Transform of a $N \times M$ image will be a data set of $N \times M$ wave-numbers:

$$\begin{aligned}\text{Discrete FT Data} = 2\pi\left(\frac{1}{N\text{pixels}}, \frac{2}{N\text{pixels}}, \ldots, \frac{N}{N\text{pixels}}\right) \\ \times 2\pi\left(\frac{1}{M\text{pixels}}, \frac{2}{M\text{pixels}}, \ldots, \frac{M}{M\text{pixels}}\right)\end{aligned} \tag{18-15}$$

representing the amplitudes of the indicated periodicities.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Discrete Fourier Transforms on Simulated Lattices

Example of taking the Discrete Fourier Transform (DFT) of the simulated lattice created above and visualizing it Fourier transform.

We create a function that shows the original data, its Fourier transform, and then its inverse transform (hopefully) back to the original image.

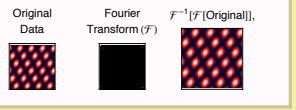
```
FourierRow[data_] :=
 Module[{fourdat =
    Fourier[data]},
  GraphicsRow[
   {ImagePlot[data, "",
     normalcontrast],
    ImagePlot[fourdat] ,
    ImagePlot[
     InverseFourier[
      fourdat], "",
     normalcontrast]},
   ImageSize → Small]]
```

1

Peaks will be located located near (kx,ky) = 2 π (a,b)/(size), where (a,b) = {(0,0), (size,0), (0,size), (size, size)}. These correspond to the longest wavelength periodicities.

```
FourierRow[AtomDensity]
```

2



1: *FourierRow* is a function to visualize, from left to right, the input intensity data, its Fourier transform, and the inverse Fourier Transform of the Fourier Transform. If the final image isn't the same as the first, then something went dreadfully wrong.

2: Notice that the Fourier Transform has very sharp features at the corners of the figure; this is because the original data lattice is a linear combination of sine waves. There are three unique peaks in the pattern. The first, and brightest, sits at $(k_x, k_y) = (2\pi/\lambda_x, 2\pi/\lambda_y) = (1, N_y)$. This peak comes from each pixel interfering with itself in a periodic repetition of the underlying rectangular (in this case, square) lattice. As the number of pixels becomes large, this peak converges to the infinite wavelength limit (or the data with no periodic correlations). This brightest peak is, in fact, the super position of four periodic peaks; the other three are at: $(1 + N_x, 1)$, $(1 + N_x, 1 + N_y)$, and $(N_x, 1 + N_y)$. The reason it appears in the upper left is related to our approximation of dx and dy with $1/N_x$ and $1/N_y$ and the underlying periodicity of the image: there are four choices on which corner to use as the bright spot, upper-left is the one that appears.

The second brightest peak is located at $^\circ 45$ to the highest intensity peak. The peak derives from the $\sin(x + y)$ in one of superposed waves—it is the signature of the planes oriented at $^\circ 45$ $(\bar{1}1)$ in the original data. Each peak corresponds to the superposition of two waves, and its intensity is one-half of the brightest spot which is the superposition of four.

There are just as many low-intensity peaks as mid-intensity. These peaks are derive from the $x - 2y$ modulated sine-wave; their intensity is less because the distance between the corresponding planes is larger.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Simulating Diffraction Patterns

Materials scientists, microscopists, and crystallographers observe the long wavelength peak at the middle of the diffraction pattern. We develop a data manipulation function that takes input data and outputs the same data, but with our approximation to $\vec{k} = 0$ at the center.

Materials Scientists, Microscopists, and Crystallographers are used to seeing the $\vec{k}$=0 spot at the center of the diffraction image; so we write a function that takes the Fourier data and manipulates to so as to move spots to the center.

```
KZeroMiddle[
  fourmat_?MatrixQ] :=
 Module[{nrows, ncols},
  {nrows, ncols} =
   Dimensions[fourmat];
  RotateRight[fourmat,
   {Quotient[nrows, 2],
    Quotient[ncols, 2]}]]
```

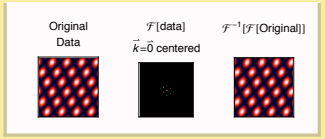
1

And, we modify our FourierRow function to use the k-at-zero transformation: FourierRowK0

A

```
FourierRowK0[AtomDensity]
```

3



1: *KZeroMiddle* uses `RotateRight` to "rolls" the data array so that the left edge goes to the center, followed by the right edge which ends up just to its left at the center; the two columns at the center roll to both edges. The same operation is performed in the vertical direction. To find the center, we use `Quotient` instead of dividing the number of columns and rows by 2 to anticipate the cases where there is an odd number of rows or columns. Fourier transform of the diffraction image are viewed side-by-side.

A–3: *FourierRowK0* duplicates the functionality of *FourierRow* , but the Fourier data is filtered with *KZeroMiddle* before display. The definition of the graphics function is straightforward and suppressed in these class notes versions. This simulates an observed fraction pattern, but with colors instead of gray-scale to indicate intensity of the image's periodicities. *KZeroAtCenter* divides the original matrix data into four approximately equal-sized parts,

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

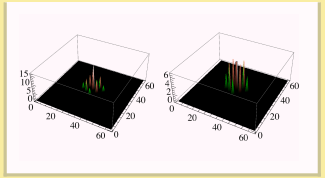
Alternative Representations of Diffraction Data

Because our data is organized as intensities over the x–y plane we can use the z-direction to add another component to visualization. We can also use the same contrast function that we employed for the two-dimensional simulation.

```
Spots3D[data_,
  range_:All] :=
 ListPlot3D[KZeroMiddle[
   Abs[Fourier[data]]],
  ColorFunction→
   (highcontrast[#3]&),
  PlotRange→range]
```
1

```
Spots3DRow[data_,
  range_] := Module[{plt},
  plt = Spots3D[data];
  GraphicsRow[
   {plt, Show[plt,
     PlotRange→range]}]]
```
2

```
Spots3DRow[
 AtomDensity, {0, 7}]
```
3



1: *Spots3D* uses `ListPlot3D` to convert our discrete two-dimensional data into a `Graphics3D` object. We create a default argument for the range of intensities to be plotted, and use *highcontrast* on the z-values.

2–3: *Spots3DRow* takes Fourier data and creates visualizations for the all the intensities, and a second argument for a range, which permits us to observe the finer structure of the diffraction intensities. In this case, because the data is the superposition of two sine-waves, discrete approximations to sharp peaks are observed.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Diffraction Patterns of Defective Lattices

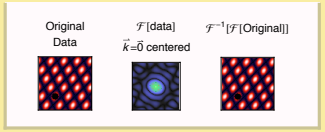
In this example, the simulated atomic density is modified to simulate the removal of one atom—in other words, we simulate a vacancy.

Algorithm to Create a Defect in a Simulated Atomic Density Image : Here, the algorithm will create new data which will be called *AtomDensityWithDefect*

A

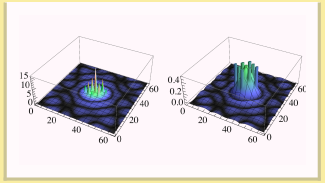
```
FourierRowK0[
 AtomDensityWithDefect]
```

2




```
Spots3DRow[
 AtomDensityWithDefect,
 {0, 1 / 2}]
```

3



A: *AtomDensityWithDefect* are simulated data with a vacancy (definition-algorithm suppressed in class notes). It selects one of the maximum intensity positions at random, and then sets data in, disk centered at that position, to zero.

2–3: The vacancy affects the diffraction pattern with diffuse, low-intensity scattering near $\vec{k} = 0$. The 3D version shows more clearly that the peaks remain the dominate feature, and we have to decrease the range to very small intensities to find the defect scattering all. Nevertheless, the intensity that is shed from the peaks into the entire spectrum reproduces the defect on the reconstructed image.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Diffraction Patterns from Lattices with Thermal 'Noise'

Functions to create a two-dimensional lattices of squares with a specifiable amount of randomness in their position are created. are developed with a variable that simulates random deviation from their ideal lattice positions.

Function to Create A Lattice of Squares : NoisyLattice[TotalSize, LatticeVector1, LatticeVector2, SquareSize, RandomDisplacements] A

```
NoNoise =
  NoisyLattice[64, {8, 4},
   {16, 16}, 1, {0, 0}];
```
2

```
FourierRowK0[NoNoise]
GraphicsRow[
 {Spots3D[NoNoise],
  Spots3D[NoNoise, {0, 2}]}]
```
3

```
SomeNoise =
  NoisyLattice[64, {8, 4},
   {16, 16}, 1, {1, 1}];
FourierRowK0[SomeNoise]
GraphicsRow[
 {Spots3D[SomeNoise],
  Spots3D[SomeNoise,
   {0, 2}]}]
```
4

A: *NoisyLattice* 's first argument is the size N of the $N \times N$ that is returned. It also takes input for the two lattice-vectors, the size of squares to place near the lattice positions, and a vector that specifies the magnitude of random displacements in the x and y directions. (The definition, which is a bit long and complicated, is suppressed in the class-notes.) This function will produce smaller unit cells if the lattice vectors are divisors of the data size.

2–3: This simulates data from 'perfect' lattice of squares. Note, that in this case, the entire diffraction pattern is filled. This is because the original data are not sine-waves, but superposed square-wave-patterns. This diffraction pattern is called the *reciprocal lattice* by materials scientists and crystallographers.

4: The data from *SomeNoise* will illustrate the effect of adding isotropic thermal noise (in a real crystal, the amplitude of the noise will be larger in the elastically soft directions, it would not be difficult to modify this function to take a matrix of compliances to multiply the random displacements) A diffuse ring of scattering about $\vec{k} = 0$ is superimposed onto the 'ideal' diffraction pattern.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Computational Microscopy

We produce functions to create circular apertures and interactively move and resize the apertures on the diffraction pattern.

Transmission electron microscopy works by accelerating electrons with a voltage difference and sending them towards a target. The electrons interact with the target and scatter. After the electrons have passed through the sample, they are focused with a magnetic objective lens (or typically lenses). This lens produces a plane at which electrons scattered in the same direction arrive at the same point—this is the diffraction pattern. Because diffraction transforms periodic elements into points, it is closely related to the fourier transform. An image of the target is created beyond this diffraction plane. An operator of an electron microscope can toggle between looking at the diffraction-plane or the image-plane.

The "Bright-Field Image" consists of using a central aperture around the direct beam to block off all others from contributing to the image.

The "Dark-Field Image" consists of selecting a specific diffracted peak with the aperture and using that to form an image.

A "Structure image," or a "lattice image," uses the direct beam and one or more diffracted beams to form the image. In this case, the apertures are typically much larger than for bright- or dark-field imaging.

Aperture size is effectively limited because of spherical aberrations that become significant for beams that are "off-axis" by a significant amount. So, in practice one can only use part of the Fourier spectrum (reciprocal space) to produce an image in TEM. You always lose some structure information in the image formation.

Function to Create Two Circular Aperatures (i.e, to remove all data from a Fourier Transform except the regions inside two specified circles:

CircAps[Center1, Center2, Radius1, Radius2, FourierData]

A

Function to Perform Diffraction Spot Microscopy on Real-Space Images

DiffractionMicroscopy[RealData] **Creates a Interactive Structure with Four Windows Arranged in a Square.**

NorthWest: All Fourier Data from Real Image with Movable Aperatures, use a clicked mouse to move aperatures to various diffraction peaks.

NorthEast: The original real space image.

SouthWest: The Fourier image of the aperature filtered data. (Don't try to move these aperatures)

SouthEast: The reconstructed image from the aperature filtered data.

B

A: *CircAps* is a function (definition suppressed in class-notes) designed to take the positions of the centers of two circular apertures, their radii, and input data (which is intended to be the Fourier transform of scattering density. It returns the data with zeroes everywhere except within the apertures, where it has the same value as the input data.

B: *DiffractionMicroscopy* (definition suppressed, but available at the links given above) takes an array of values representing scattering density, and creates an interactive simulation which allows the user to move the apertures with the mouse, and their radii with slider controls. This definition is only about 20 lines of code, and demonstrates the economy of MATHEMATICA® 6's new `Manipulate` function. We will demonstrate examples below.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualizing Simulated Selected Area Diffraction

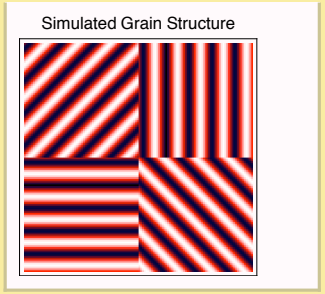
We create a function that creates a square array of square "grains". Each grain will be simplified by creating a sinusoidal modulation in each of the $N_g \times N_g$ with a random orientation picked from an equally spaced set of angles in $(0, N_g^2\pi)$.

GrainStructure[TotalSize, GrainSize]
Creates an array of "square grains" with stripes oriented in somewhat random orientations

A

```
Grains =
  GrainStructure[128, 64];
ImagePlot[Grains,
 "Simulated
   Grain Structure",
 normalcontrast, Large]
```

2



A: This is a definition of the function *GrainStructure* (suppressed in class-notes). Its first argument is the number of pixels along one side of the image, and the second is the number of pixels along the side of the grains. It is best to make the grain size a divisor of the image size.

2: This is an example of creating data and imaging it for a 2×2 grain structure.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

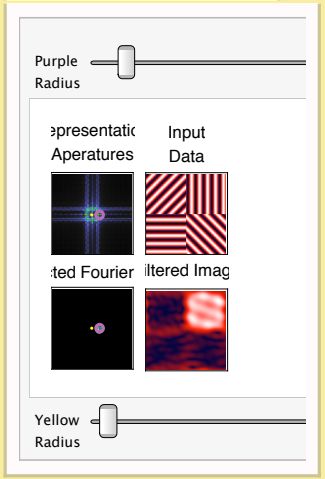
Simulated Diffraction Imaging on a Polycrystal

We use our simulated grain structure as input to our diffraction simulator, *DiffractionMicroscopy* .

Diffraction Spot Microscopy on grain structure.

```
DiffractionMicroscopy[
 Grains]
```

1



1: Here is an example of our interactive function, *DiffractionMicroscopy* , on a polycrystal. You can move each aperture by mouse-dragging and control their sizes with the sliders. You can only move the apertures in the upper-left image. Try these simple experiments first:

Image Orientation of a Single Grain: Move the purple aperture over on of the green diffraction peaks. Notice that only one grain is imaged in the reconstruction. Because the yellow aperture is picking up data from the $\vec{k} = 0$ spot, the other aperture is producing the modulation of a single sine wave.

Imaging a Single Grain: Shrink the yellow aperture to zero; this is a "dark-field" simulation. Move the purple aperture over a single peak; notice that a single grain is imaged, but its modulation has disappeared.

Imaging a Defect: Keeping the yellow aperture at zero-radius, move the purple apertures over one of the streaks in the diffraction pattern. Notice that a *grain boundary* is imaged—pay attention to the orientation of the grain boundary relative to that of the streak in the diffraction image.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

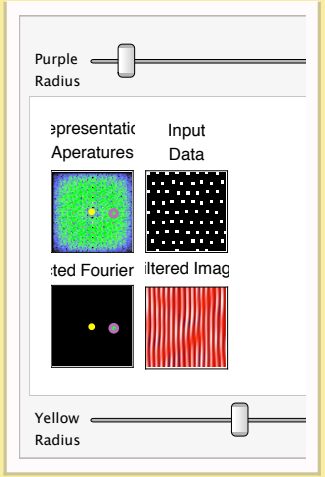
notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Bright-Field and Dark-Field Imaging of a Lattice with Thermal Noise

We use our ideal lattice of squares and a similar one with a bit of thermal noise to continue our investigation of diffraction phenomena.

```
DiffractionMicroscopy[
 NoNoise]
```
1

```
DiffractionMicroscopy[
 SomeNoise]
```
2



1: Here, we use our computational microscope, *DiffractionMicroscopy* , on an ideal lattice. Here are some experiments to try:
Image a Set of Planes Leave the yellow aperture over the $\vec{k} = 0$ peak. Move the purple aperture one of the nearby spots. Notice the orientation and periodicity of the planes in the reconstructed image.
Detect Periodicity Leave the yellow aperture at $\vec{k} = 0$. Move the purple aperture from one of the nearby peaks to a similarly oriented one along the same ray from the origin. Notice that the orientation of the planes do not change, but the period of modulation is increased. (Also, recall that this is a complex superposition of sine-waves to make squares, in the simple superposition of sine-waves, there we fewer peaks.)

2: Here, we do the same example with our randomly perturbed lattice of squares. There is a significant amount of diffuse scattering, but by observing carefully, you will see the the same peaks that were present for the perfect lattice.
Discover Robustness of Imaging with Noise If you leave the yellow aperture over $\vec{k} = 0$, and move the purple aperture over one of the "perfect peaks," you will see a reconstruction of perfect planes even though they are barely discernible in the original image. If you shrink the purple radius, while leaving it over the peak, you will see the quality of the planes improves as less diffuse scattering is included. This simulates how a microscopist can image individual atom planes at finite temperatures where atoms are vibrating around their equilibrium positions.

3.016 Home
Full Screen
Close
Quit

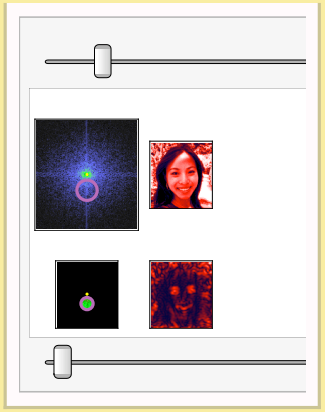
notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Selected Area Diffraction on Image Data

DiffractionMicroscopy is used on data that is extracted from a gray-scale image-file.

```
AnImage = Import[
  "http://pruffle.mit.edu/
    3.016-2007/MsChang.
    pgm"]
```
1

```
DiffractionMicroscopy[
 1 - ChangData]
```
2



1: We read in an image that is stored on the 'net. We do a bit of plastic surgery on this data to put it into a form that is ready for our microscope (plastic surgery algorithm suppressed in class-notes)

2: We perform selected area diffraction on this image. Notice that we can highlight different aspects of the image by selecting different aperture locations.

3.016 Home

Full Screen

Close

Quit

Lecture 19: Ordinary Differential Equations: Introduction

Reading:
Kreyszig Sections: 1.1, 1.2, 1.3 (pages2–8, 9–11, 12–17)

Differential Equations: Introduction

Ordinary differential equations are relations between a function of a single variable, its derivatives, and the variable:

$$F\left(\frac{d^n y(x)}{dx^n}, \frac{d^{n-1} f(x)}{dx^{n-1}}, \ldots, \frac{d^2 y(x)}{dx^2}, \frac{dy(x)}{dx}, y(x), x\right) = 0 \tag{19-1}$$

A *first-order* Ordinary Differential Equation (ODE) has only first derivatives of a function.

$$F(\frac{dy(x)}{dx}, y(x), x) = 0 \tag{19-2}$$

A *second-order* ODE has second and possibly first derivatives.

$$F\left(\frac{d^2 y(x)}{dx^2}, \frac{dy(x)}{dx}, y(x), x\right) = 0 \tag{19-3}$$

For example, the one-dimensional time-independent Shrödinger equation,

$$-\frac{\hbar}{2m}\frac{d^2\psi(x)}{dx^2} + U(x)\psi(x) = E\psi(x)$$

or

$$-\frac{\hbar}{2m}\frac{d^2\psi(x)}{dx^2} + U(x)\psi(x) - E\psi(x) = 0$$

3.016 Home
Full Screen
Close
Quit

is a second-order ordinary differential equation that specifies a relation between the wave function, $\psi(x)$, its derivatives, and a spatially dependent function $U(x)$.

Differential equations result from physical models of anything that varies—whether in space, in time, in value, in cost, in color, etc. For example, differential equations exist for modeling quantities such as: volume, pressure, temperature, density, composition, charge density, magnetization, fracture strength, dislocation density, chemical potential, ionic concentration, refractive index, entropy, stress, etc. That is, almost all models for physical quantities are formulated with a differential equation.

The following example illustrates how some first-order equations arise:

Iterative Application of Function

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Iteration: First-Order Sequences from a Fixed Boundary Condition

Sequences are developed in which the next iteration only depends on the current value; in this most simple case simulate exponential growth and decay.

Suppose a function, F[i], changes proportional to its current size, i.e., F[i+1] = F[i] + αF[i]

```
ExplFun[i_, α_] :=
 ExplFun[i, α] =
  ExplFun[i - 1, α] +
   α * ExplFun[i - 1, α]
```
1

In the above, the symbol is assigned (ExplFun[i,alpha] = ...) as part of the function definition, so that intermediate values are ``remembered.''
The function needs some value at some time (an initial condition) from which it obtains all its other values:

```
ExplFun[0, 0.25] = π / 4
```
2

```
ExplFun[18, 0.25]
```
3

1: *ExpleFun* taking two arguments is defined: the first argument represents the iteration and the second represents a single parameter expressing how the current iteration grows. The value at the $i+1^{\text{th}}$ iteration is the sum of the value of the i^{th} plus α times value of the i^{th}iteration. If this is a bank account and interest is compounded yearly, then the i^{th}iteration is the value of an account after i years at a compounded annual interest rate of α. This function has improved performance (but consumes more memory) by storing its intermediate values.

2: Of course, the function would iterate for ever if an initial value is not specified; and so it is specified here.

3: For, example this would produce the 18^{th} iteration of growth with a compounding rate of 25% with $\pi/4$ at the initial state.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Iteration: First-Order Sequences with a Generalized Boundary Condition

The previous example is generalized so that the iteration function is generalized for an arbitrary initial values.

```
ExplFun[0 , α_,
  InitVal_] := InitVal
ExplFun[Inc_Integer,
  α_, InitVal_] :=
 ExplFun[Inc, α, InitVal] =
  ExplFun[Inc - 1, α,
    InitVal] + α * ExplFun[
     Inc - 1, α, InitVal]
```

1

```
Traj[
  Steps_Integer?Positive,
  α_, InitVal_] :=
 Traj[Steps, α, InitVal] =
  AppendTo[Traj[Steps - 1,
    α, InitVal], ExplFun[
    Steps, α, InitVal]]
Traj[0, _, _] = {};
```

2

```
Traj[12, .01, .001]
```

3

We define a function, Evolve, producing an interactive tool with input : Initial values and α.producing an interactive tool with input : Initial values and α.

A

```
Evolve[300]
```

5

1: Because the initial value and the 'growth factor' α determine all subsequent iterations, it is sensible to 'overload' *ExplFun* (i.e., define the function to behave differently depending on the number and type of its arguments) to take an extra argument for the initial value. Here, if *ExplFun* is called with three arguments *and the first argument is zero*, then the initial value is set; otherwise it is a recursive definition with intermediate value storage.

2: *Traj* is an example of a function that builds a list by first-order iteration. It produces a result that is suitable for input to `ListPlot`. The second part of the definition defines the 0^{th} item of the list to always be an empty list, no matter what other values are passed to it. *Traj* does its work by calculating new pairs with the help of *ExplFun* and then recursively appends the current value to the growing list.

3: Here is an example which will produce a list of twelve entries, starting from the first iteration of 0.001 with growth factor of 1%.

8: To visualize the behavior as a function of its initial value, an interactive function, *Evolve* , is defined (definition suppressed in notes, but available via the links). It takes an argument for the maximum number of iterations, and the initial value and growth factor are controlled with `Manipulate`.

3.016 Home

◀◀ ◀ ▶ ▶▶

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Space-Covering Sequences: Families of Trajectories

The previous example is generalized so that the iteration function is generalized for an arbitrary initial values. several plots. Once the growth rate is fixed, we visualize how each curve "belongs" to a particular initial value; the set of all initial values generates a family of curves that fill the plane—each point belongs to one and only one trajectory.

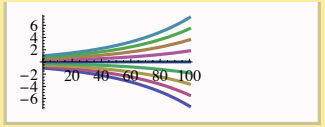
Plotting a bunch of curves for the same positive α value, but each corresponding to a different initial value.

```
PlotTrajs[α_] := Block[
  {$RecursionLimit = 10^4},
  ListPlot[Evaluate[
    Table[Traj[300, α, iv],
     {iv, -1, 1, 0.25}]],
   PlotRange → All,
   Joined → True,
   PlotStyle → Thick]]
```

1

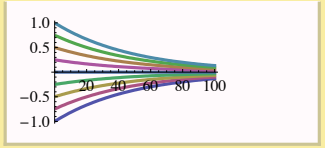
```
PlotTrajs[0.02]
```

2




```
PlotTrajs[-0.02]
```

3



1: *PlotTrajs* is a function that provides a visualization of trajectories for an input growth. It works by generating a set of initial values to pass to *Traj* and then plots them with `ListPlot`.

2: If $\alpha > 0$, the function goes to $\pm\infty$ depending on the sign of the initial value. For a fixed α every point in the plane belongs to one and only one trajectory associated with an initial value and that α.

3: If $\alpha < 0$, the function asymptotically goes to zero, independent of the initial value. In this case as well, the plane is completely covered by non-intersecting trajectories.

3.016 Home
Full Screen
Close
Quit

The previous example is generalized to a discrete change Δt of a continuous (i.e., time-like) parameter t. The following example demonstrates the simplest method of numerically solving a simple first-order ODE. *first-order explicit finite differencing* or *Euler integration.*

We begin by approximating the derivative dy/dt at time t with a finite difference approximation:

$$\Delta y/\Delta t = [y(t + \Delta t) - y(t)]/[(t + \Delta t) - t] \tag{19-4}$$

We can write down a formula for $y(t + \Delta t)$ in terms of current values at t, and thus 'project y into the future. Suppose we use fixed small time steps Δt and the short-hand $y_n = y(n\Delta t)$, $y_{n+1} = y(n\Delta t + \Delta t)$. Now, we must determine which value to use for $f(y(t))$ in $dy/dt = f(y)$: the current value $f(y_n)$, the future value $f(y_{n+1})$, an average value $([f(y_n) + f(y_{n+1})]/2$, or something else. The simplest thing to do is use the current value and then every term (but y_{n+1} is in terms of n:

$$y_{n+1} = y_n + \Delta t f(y_n) \tag{19-5}$$

This is called explicit forward-differencing or Euler's method,

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

First-Order Finite Differences: Method 1 Explicit Finite Differences

We implement this simple method described in Eq. 19-5 be creating a function which 'projects' the current value of y into the future.

Approximate f(y) with f(y$_{i-1}$);

```
PushMethod1[f_,
  {ti_, yi_}, Δt_] :=
 {ti + Δt, yi + Δt f[yi]}
```
1

```
FuncEx[y_] := -Sin[y]
```
2

```
PushMethod1[
 FuncEx, {0, 1}, .01]
```
3

```
PushMethod1[FuncEx,
 PushMethod1[FuncEx,
  {0, 1}, .01], .01]
```
4

```
Nest[PushMethod1[FuncEx,
   #, .01] &, {0, 1}, 2]
```
5

```
NestList[
 PushMethod1[FuncEx,
   #, .01] &, {0, 1}, 2]
```
6

```
NestWhileList[
 PushMethod1[FuncEx,
   #, .01] &, {0, 1},
 (First[#] < 0.03) &]
```
7

1: The function *PushMethod1* takes three arguments: argument 1 is a place-holder for *another function* that determines how each increment changes (i.e., the function $f = dy/ft$); argument 2 is the current value; argument 3 is the discrete forward difference (i.e., Δt).

2: *FuncEx* is defined to to pass to sequence-generating functions—it plays the role of $f(y_n)$ in Eq. 19-5.

3: For example, this pushes a value `{0,1}` by $\Delta t = 0.01$ into the future with `FuncEx[1]`.

4: Calling the function, *PushMethod1* , recursively on itself (once) pushes the value iteravely into the future (twice).

5: We can generalize this recursion method by using `Nest` (`Nest[f,x,3]` *rightarrow* `f[f[f[x]]]`). However, we must turn *PushMethod1* into a function of a single argument, so there is no ambiguity about which value is being iteratively pushed forward. This is done by creating a *Pure Function* version of *PushMethod1* . The pure function is indicated by the trailing ampersand, `&`, and the `#` becomes a place holder for the single argument. Thus, `Nest[(PushMethod[FuncEx,#,0.01])&, {0,1}, 2]` nests *PushMethod1* with *fixed* first and third arguments (`FuncEx` and `0.01`) on the initial value `{0,1}` twice.

6: `NestList` is another version of `Nest`, but it stores each increment in a growing list and returns a list structure.

7: `NestListWhile` is another version of `NestList`, but with a switch to tell it when to stop 'Nesting.' We use this method to indicate "at what time" the nesting should stop, and not "after how many nests." For `NestListWhile`' test-argument, we use another pure function: it takes the current value of `{t,y}` and tests to see if `t` is less than `0.03`.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

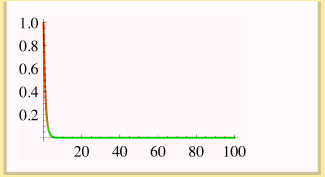
notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualizing Trajectories from Explicit Forward Differences

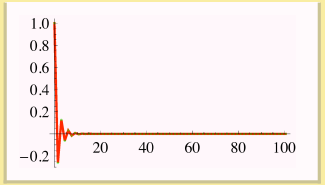
Examples of the explicit forward differencing function *PushMethod1* called recursively with `NestListWhile` are illustrated. An example of *Numerical Instability* appears.

A Function PlotM1, taking arguments for α and initial condition is used with NestListWhile and ListPlot to produce graphics with a red line and green points. **A**

`PlotM1[0.1, 1]` **2**



`PlotM1[1.5, 1]` **3**



A: *PlotM1* is defined which takes a first argument for a time-step, and a second argument is y_0. It uses `ListPlot` to create a trajectory, and show line segments between the computed points. (The definition is suppressed in class-notes, it is available via the links given above)

2: Here is an example of a stable numerical integration of a first-order ODE. We have not evaluated how *accurate* the numerical algorithm is, but only that it is well-behaved.

3: Using a larger time-step, we can see that the algorithm is becoming less well-behaved. This introduces the concept *maximum stable time-step*.

3.016 Home
Full Screen
Close
Quit

Forward Differencing Methods: Implicit Methods

As in the implicit method, we begin by approximating the derivative dy/dt at time t with a finite difference approximation:

$$\Delta y/\Delta t = [y(t+\Delta t) - y(t)]/[(t+\Delta t) - t] \tag{19-6}$$

However, in this case we will use the expected future value, y_{n+1} as the argument to $f(y)$.

$$\begin{aligned} y_{n+1} &= y_n + \Delta t f(y_{n+1}) \\ &= y_n + \Delta t \left[f(y_n) + \left.\frac{df}{dy}\right|_{y_n} (y_{n+1} - y_n) \right] \end{aligned} \tag{19-7}$$

Because y_{n+1} appears on both sides, we have to solve for it (this is the implicit step),

$$y_{n+1} = \frac{y_n + \Delta t (f(y_n) - \left.\frac{df}{dy}\right|_{y_n} y_n)}{1 - \Delta t \left.\frac{df}{dy}\right|_{y_n}} \tag{19-8}$$

This is called implicit forward-differencing.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

First-Order Finite Differences: Method 1 Explicit Finite Differences

We implement this implicit method described in Eq. 19-8

Approximate f(y) with $f(y_i)$; then solving the finite difference equation above,
$y_i = y_{i-1} + \Delta t\,[f(y_i)]$.
So, $y_i = y_{i-1} + \Delta t\,(f(y_i) + f'(y_{i-1})dy)$
$y_i = y_{i-1} + \Delta t\,(f(y_i) + f'(y_{i-1})(y_i - y_{i-1}))$
$y_i = (y_{i-1} - \Delta t\,[f(y_{i-1}) - f'(y_{i-1})y_{i-1}])/(1 - \Delta t\,f'(y_{i-1}))$

```
PushMethod2[f_,
  df_, {ti_, yi_},
  Δt_] := {ti + Δt,
  (yi + (Δt (f[yi] -
         df[yi] yi))) /
   (1 - Δt df[yi])}
```

1

```
dFuncEx[y_] :=
 Evaluate[D[FuncEx[y], y]]
```

2

```
NestList[
 PushMethod2[FuncEx,
   dFuncEx, #, 0.1] &,
 {0, 1}, 3]
```

3

We define a function, PlotM2, which takes arguments Δt and InitialCondition and then uses ListPlot with Blue lines and Gray points.

A

1: *PushMethod2* implements the implicit differencing strategy. However, we must also provide this method with a function representing the derivative of f.

2: We define the derivative function, but use `Evaluate` on the right-hand side of the delayed assignment (`:=`) so that the derivative operator `D` is not called each time the function is used.

3: We can use the `Nest`-family of functions as before.

A: A function to plot the implicit function results is defined for comparison to the explicit method.

3.016 Home

Full Screen

Close

Quit

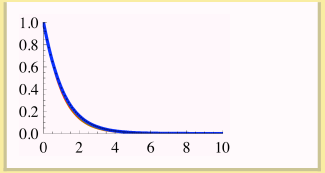
notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Comparison of Implicit and Explicit Methods

We plot the results from the two different time-stepping methods and show that the implicit method is more stable. We still have not evaluated the accuracy of either method.

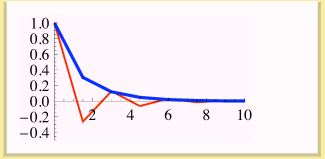
```
Show[PlotM1[0.1, 1],
 PlotM2[0.1, 1], PlotRange →
  {{0, 10}, {0, 1}}]
```

1




```
Show[PlotM1[1.5, 1.0],
 PlotM2[1.5, 1.0],
 PlotRange →
  {{0, 10}, {-0.5, 1}}]
```

2



Method 2 will fail if the step size is increased to 2

1: With a time step of $\Delta t = 0.1$, the two methods give results that are barely discernible. This gives us confidence in the hypothesis that the solutions are also accurate at this time step.

2: At larger time steps, the implicit method is more well-behaved. However, if the step size is made a little larger, both methods will become unstable.

3.016 Home

Full Screen

Close

Quit

Geometrical Interpretation of Solutions

The relationship between a function and its derivatives for a first-order ODE,

$$F(\frac{dy(x)}{dx}, y(x), x) = 0 \tag{19-9}$$

can be interpreted as a level set formulation for a two-dimensional surface embedded in a three-dimensional space with coordinates (y', y, x). The surface specifies a relationship that must be satisfied between the three coordinates.

If $y'(x)$ can be solved for exactly,

$$\frac{dy(x)}{dx} = f(x, y) \tag{19-10}$$

then $y'(x)$ can be thought of as a height above the x-y plane.

For a very simple example, consider Newton's law of cooling which relates the change in temperature, dT/dt, of a body to the temperature of its environment and a *kinetic coefficient* k:

$$\frac{dT(t)}{dt} = -k(T - T_o) \tag{19-11}$$

It is very useful to "non-dimensionalize" variables by scaling via the physical parameters. In this way, a single ODE represents *all* physical situations and provides a way to describe universal behavior in terms of the single ODE. For Newton's law of cooling, this can be done by defining non-dimensional temperatures and time with $\Theta = T/T_o$ and $\tau = kt$, then if T_o and k are constants:

$$\frac{d\Theta(\tau)}{d\tau} = (1 - \Theta)$$

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visual Understanding of the Behavior of First-Order ODES

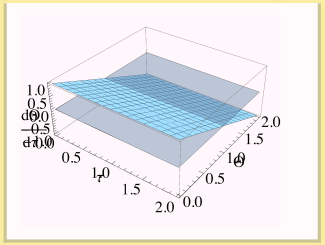
The surface representation provides a useful way to think about differential equations—much can be inferred about a solution's behavior without computing the solution exactly. This is shown for a simple case of Newton's law of cooling Equation 19 and an artificial case.

```
ZeroPlane[xmin_,
  xmax_, ymin_, ymax_] :=
Graphics3D[{Gray,
  Opacity[0.25], Cuboid[
   {xmin, ymin, -.001},
   {xmax, ymax, .001}]}]
```

1

```
Show[
 Plot3D[1 - Θ, {tau, 0, 2},
  {Θ, 0, 2}, AxesLabel →
   {"τ", "Θ", "dΘ/dτ"},
  DisplayFunction ->
   Identity],
 ZeroPlane[0, 2, 0, 2]]
```

2



1: For first-order ODEs, behavior is dominated by whether the derivative term is positive or negative. Here, a `Graphics3D` object is created for a gray-colored opaque horizontal plane (in reality we use a very thin slab) at $z = 0$. We will use this function to evaluate when the derivative is positive and the value is increasing or negative and the value is decreasing.

2: This will create the surface associated with Newton's law of cooling with the zero plane. This case is very simple. The sign of the change of Θ depends only the sign of $1 - \Theta$ and therefore $d\Theta/dt = 0$ is the parametric curve (a line in this case) $(d\Theta/dt = 0, \Theta = 1, \tau)$. That is, if $\Theta = 1$ at any time τ it will stay there at all subsequent times (also, at all previous times as well). Because $\Theta(\tau)$ will always increase when $\Theta < 1$ and will always decrease when $\Theta > 1$, the solutions will asymptotically approach $\Theta = 1$.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

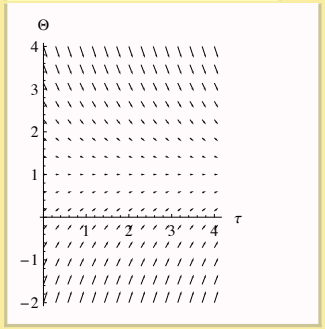
Visualizing the Geometry of Flows for First-Order ODES

By creating vector field which 'points' toward subsequent points as inferred from the ODE, we produce a very useful way to understand solution behavior for a variety of initial conditions, without computing a solution to the ODE. This is shown again for a simple case of Newton's law of cooling

Plot the vectorf-ield (dτ,dΘ) = dτ(1, $\frac{d\Theta}{d\tau}$) We can do so by plotting vectors of the form {dτ, dΘ} = dτ{1, $\frac{d\Theta}{d\tau}$} which will be proportional to the vector {1, 1-Θ}. This is done as follows:

```
Needs["VectorFieldPlots`"];
VectorFieldPlots`VectorFie\
  ldPlot[{1, 1 - Θ},
{tau, 0, 4},
{Θ, -2, 4}, Axes → True,
AxesLabel → {"τ", "Θ"},
ImageSize → Full]
```

1



1: The asymptotic behavior can be further visualized by plotting a first-order difference representation of how the solution is changing in time, i.e, $(d\tau, d\Theta) = d\tau\left(1, \frac{d\Theta}{d\tau}\right)$ This can be obtained with `VectorFieldPlot` from the `VectorFieldPlots` package. Here the magnitude of the arrows is scaled by setting $d\tau = 1$.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualizing the Geometry of Flows for First-Order ODES

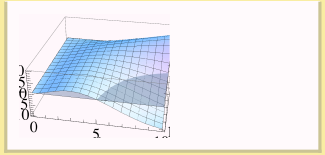
We utilize our visualization methods for intuitive understanding of the behavior of ODES for the case:

$$\frac{dy}{dt} = y \sin\left(\frac{yt}{1+y+t}\right)$$

Slightly more complicated example:
$\frac{dy}{dt} = y \sin(\frac{yt}{1+t+y})$,
$(dt,dy) = dt(1, y\sin(\frac{yt}{1+t+y}))$

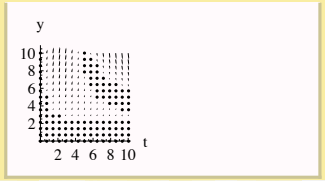
```
Show[Plot3D[
  y Sin[y t / (t + y + 1)],
  {t, 0, 10}, {y, 0, 10}, ,
  ZeroPlane[0, 10, 0, 10]]]
```

1



```
VectorFieldPlot[
 {1, y Sin[y t / (t + y + 1)]},
 {t, 0, 10}, {y, 0, 10}]
```

2



1: This case can be visualized as well and the behavior can be inferred whether the derivative lies above or below the zero-plane (i.e., the sign of the derivative). Where $dy/dt < 0$, y decreases as time marches forward; thus it moves toward the intersection of the zero plane and the dy/dt-surface. We see that the slope of the surface evaluated along the curve of intersection determines whether there is an "attractor-manifold" in the ODE.

2: `VectorFieldPlot` provides another method to follow a solution trajectories: we plot vectors proportional to $dt(1, y\sin[yt/(1+y+t)]$.

3.016 Home
Full Screen
Close
Quit

Separable Equations

If a first-order ordinary differential equation $F(y', y, x) = 0$ can be rearranged so that only one variable, for instance y, appears on the left-hand-side multiplying its derivative and the other, x, appears only on the right-hand-side, then the equation is said to be 'separated."

$$\begin{aligned} g(y)\frac{dy}{dx} &= f(x) \\ g(y)dy &= f(x)dx \end{aligned} \tag{19-12}$$

Each side of such an equation can be integrated with respect to the variable that appears on that side:

$$\int_{y(x_o)}^{y} g(\eta)d\eta = \int_{x_o}^{x} f(\xi)d\xi \tag{19-13}$$

if the initial value, $y(x_o)$ is known. If not, the equation can be solved with an integration constant C_0,

$$\int g(y)dy = \int f(x)dx + C_0 \tag{19-14}$$

where C_0 is determined from initial conditions. or

$$\int_{y_{\text{init}}}^{y} g(\eta)d\eta = \int_{x_{\text{init}}}^{x} f(\zeta)d\zeta \tag{19-15}$$

where the initial conditions appear explicitly.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Using MATHEMATICA® 's Built-in Ordinary Differential Equation Solver

MATHEMATICA® has built-in exact and numerical differential equations solvers. DSolve takes a representation of a differential equation with initial and boundary conditions and returns a solution if it can find one. If insufficient initial or boundary conditions are specified, then "integration constants" are added to the solution.

```
dsol = DSolve[
  {y'[t] == FuncEx[y[t]]},
  y[t], t]
```

1

$$\{\{y[t] \to 2\,\text{ArcTan}\left[e^{-t+C[1]}\right]\}\}$$

```
dsol = DSolve[
  {y'[t] == FuncEx[y[t]],
   y[0] == 1}, y[t], t]
```

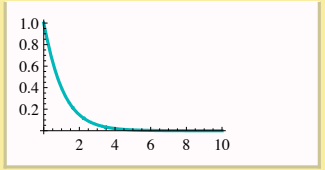
2

$$\{\{y[t] \to 2\,\text{ArcTan}\left[e^{-t}\,\text{Tan}\left[\frac{1}{2}\right]\right]\}\}$$

The next statement extracts y (x) for plotting ..

```
ExactPlot =
 Plot[ y[t] /. dsol,
  {t, 0, 10}, PlotStyle →
   {Thick, Darker[Cyan]},
  PlotRange → All]
```

3



1: `DSolve` operates like `Solve` . It takes a list of equations containing symbolic derivatives, the function to be solved for, and the dependent variable. In this case, the general solution of the example we used for finite differencing examples: $\frac{dy(x)}{dx} = FuncEx\,[y]$ `DSolve` returns a list of rules. The solutions are be obtained by applying the rules (i.e., `y[x]/.dsol`). The solution will depend on an integration constant(s) in general. MATHEMATICA® uses the symbols `C[1],C[2],etc` as place-holders for the integration constants.

2: If additional If more constraints (i.e., equations) are provided, then (provided a solution exists) the integration constant is determined as well. This is the exact solution to what we were numerically approximating above.

3: The solution is plotted by turning the "solution rule" into a plot-table `y[t] Flatten`. The plot is stored as a graphics object `ExactPlot`.

3.016 Home
Full Screen
Close
Quit

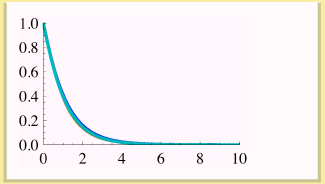
notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Comparision of Exact Solutions to Finite Difference Methods

We compare the plots of implicit, explicit finite differencing to the exact solution obtained by `DSolve`.

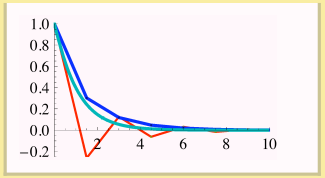
```
Show[PlotM1[0.1, 1],
 PlotM2[0.1, 1],
 ExactPlot, PlotRange →
  {{0, 10}, {0, 1}}]
```

1




```
Show[PlotM1[1.5, 1],
 PlotM2[1.5, 1],
 ExactPlot, PlotRange →
  {{0, 10}, {-0.25, 1}}]
```

2



1: To see how finite differencing compares to the exact solution, we plot all three trajectories together. The less-stable explicit method is more accurate for intermediate values of t.

2: This shows the comparison at larger time steps.

3.016 Home

Full Screen

Close

Quit

While the accuracy of the first-order differencing scheme can be determined by comparison to an exact solution, the question remains of how to establish accuracy and convergence with the step-size δ for an arbitrary ODE. This is a question of primary importance and studied by Numerical Analysis.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Using MATHEMATICA® 's Differential Equation Solver on a First-Order ODE: Less Trivial Example

We solve $y'(x) + xy(x) = 0$ for a 'strange' condition $y'(5) = 1$ and plot the solution.

```
dsol = DSolve[
  y'[x] + x * y[x] == 0,
  y[x], x]
```

1

Boundary conditions other than y[0]:

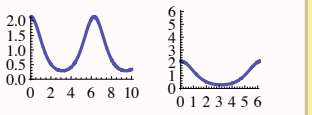
```
dsol = DSolve[
  {y'[x] + Sin[x] * y[x] ==
    0, y'[5] == 1}, y[x], x]
```

2

```
{{y[x] → -e^(-Cos[5]+Cos[x]) Csc[5]}}
```

```
GraphicsRow[
 {p = Plot[y[x] /. dsol,
    {x, 0, 10},
    PlotStyle → Thick],
  Show[p, PlotRange →
    {{0, 6}, {0, 6}},
   AspectRatio → 1]}]
```

3



1: This demonstrates the use of `DSolve`, because we have not supplied enough conditions to determine the solution exactly, MATHEMATICA® introduces all the undetermined constants of integration. In this case, there is only one undetermined constant.

2: Here, the solution is required to have a slope of unity at $x = 5$. If such a value is possible, then MATHEMATICA® will compute the corresponding value of `C[1]`.

3: This demonstrates how to extract the solution and plot it. It is plotted a second time with the same y and x scales so we can see that the slope is indeed one at $x = 5$.

3.016 Home

Full Screen

Close

Quit

Lecture 20: Linear Homogeneous and Heterogeneous ODEs

Reading:
Kreyszig Sections: 1.4, 1.5 (pages19–25, 26–32)

Ordinary Differential Equations from Physical Models

In engineering and physics, modeling physical phenomena is the means by which technological and natural phenomena are understood and predicted. A model is an abstraction of a physical system, often with simplifying assumptions, into a mathematical framework. Every model should be verifiable by an experiment that, to the greatest extent possible, satisfies the approximations that were used to obtain the model.

In the context of modeling, differential equations appear frequently. Learning how to model new and interesting systems is a learned skill—it is best to learn by following a few examples. Grain growth provides some interesting modeling examples that result in first-order ODES.

Grain Growth

In materials science and engineering, a grain usually refers a single element in an ensemble that comprises a polycrystal. In a single phase polycrystal, a grain is a contiguous region of material with the same crystallographic orientation. It is separated from other grains by *grain boundaries* where the crystallographic orientation changes abruptly.

A grain boundary contributes extra free energy to the entire system that is proportional to the grain boundary area. Thus, if the boundary can move to reduce the free energy it will.

Consider simple, uniformly curved, isolated two- and three-dimensional grains.

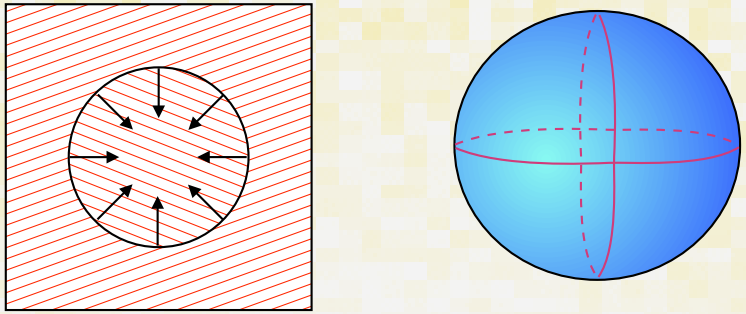


Figure 20-18: Illustration of a two-dimensional isolated circular grain and a three-dimensional isolated spherical grain. Because there is an extra energy in the system $\Delta G_{2D} = 2\pi R\gamma_{gb}$ and $\Delta G_{3D} = 4\pi R^2\gamma_{gb}$, there is a driving force to reduce the radius of the grain. A simple model for grain growth is that the velocity (normal to itself) of the grain boundary is $v_{gb} = M_{gb}\gamma_{gb}\kappa$ where M_{gb} is the grain boundary mobility and κ is the mean curvature of the boundary. The normal velocity v_{gb} is towards the center of curvature.

A relevant question is "how fast will a grain change its size assuming that grain boundary migration velocity is proportional to curvature?"

For the two-dimensional case, the rate of change of area can be formulated by considering the following illustration.

3.016 Home

Full Screen

Close

Quit

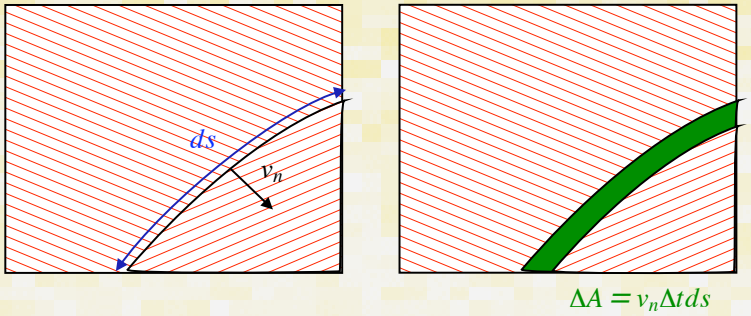


Figure 20-19: A segment of a grain boundary moving with normal velocity v_n will move a distance $v_n \Delta t$ in a short time Δt. The motion will result in a change of area $-\Delta A$ for the shrinking grain. Each segment, ds, of boundary contributes to the loss of area by $\Delta A = -v_n \Delta t ds$.

Because for a circle, the curvature is the same at each location on the grain boundary, the curvature is uniform and $v_n = M_{gb}\kappa_{gb}\gamma_{gb} = M_{gb}\gamma_{gb}/R$. Thus

$$\frac{dA}{dt} = -M_{gb}\gamma_{gb}\frac{1}{R}2\pi R = -2\pi M_{gb}\gamma_{gb} \tag{20-1}$$

Thus, the area of a circular grain changes at a constant rate, the rate of change of radius is:

$$\frac{dA}{dt} = \frac{d\pi R^2}{dt} = 2\pi R\frac{dR}{dt} = -2\pi M_{gb}\gamma_{gb} \tag{20-2}$$

which is a first-order, separable ODE with solution:

$$R^2(t) - R^2(t=0) = -2M_{gb}\gamma_{gb}t \tag{20-3}$$

For a spherical grain, the change in volume ΔV due to the motion of a surface patch dS in a time Δt is $\Delta V = v_n \Delta t\, dS$. The curvature of a sphere is

$$\kappa_{sphere} = \left(\frac{1}{R} + \frac{1}{R}\right) \tag{20-4}$$

Therefore the velocity of the interface is $v_n = 2M_{gb}\gamma_{gb}/R$. The rate of change of volume due to the contributions of each surface patch is

$$\frac{dV}{dt} = -M_{gb}\gamma_{gb}\frac{2}{R}4\pi R^2 = -8\pi M_{gb}\gamma_{gb}R == -4(6\pi^2)^{1/3}M_{gb}\gamma_{gb}V^{1/3} \tag{20-5}$$

which can be separated and integrated:

$$V^{2/3}(t) - V^{2/3}(t=0) = -\text{constant}_1 t \tag{20-6}$$

or

$$R^2(t) - R^2(t=0) = -\text{constant}_2 t \tag{20-7}$$

which is the same functional form as derived for two-dimensions.

The problem (and result) is more interesting if the grain doesn't have uniform curvature.

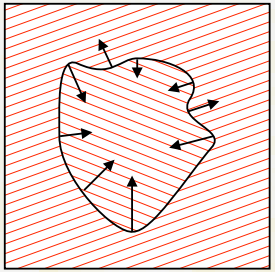


Figure 20-20: For a two-dimensional grain with non-uniform curvature, the local normal velocity (assumed to be proportional to local curvature) varies along the grain boundary. Because the motion is in the direction of the center of curvature, the velocity can be such that its motion increases the area of the interior grain for some regions of grain boundary and decreases the area in other regions.

However, it can still be shown that, even for an irregularly shaped two-dimensional grain, $A(t) - A(t=0) = -(\text{const})t$.

Exact Differential Forms

In classical thermodynamics for simple fluids, expressions such as

$$\begin{aligned} dU =& TdS - PdV \\ =& \left(\frac{\partial U}{\partial S}\right)_V dS + \left(\frac{\partial U}{\partial V}\right)_S dV \\ =& \delta q + \delta w \end{aligned} \tag{20-8}$$

represent the differential form of the combined first and second laws of thermodynamics. If $dU = 0$, meaning that the differential Eq. 20-8 is evaluated on a surface for which internal energy is constant, $U(S,V) = \text{const}$, then the above equation becomes a *differential form*

$$0 = \left(\frac{\partial U}{\partial S}\right)_V dS + \left(\frac{\partial U}{\partial V}\right)_S dV \tag{20-9}$$

This equation expresses a relation between changes in S and changes in V that are necessary to remain on the surface $U(S,V) = \text{const}$.

Suppose the situation is turned around and you are given the first-order ODE

$$\frac{dy}{dx} = -\frac{M(x,y)}{N(x,y)} \tag{20-10}$$

which can be written as the differential form

$$0 = M(x,y)dx + N(x,y)dy \tag{20-11}$$

Is there a function $U(x,y) = \text{const}$ or, equivalently, is it possible to find a curve represented by $U(x,y) = \text{const}$?

If such a curve exists then it depends only on one parameter, such as arc-length, and on that curve $dU(x,y) = 0$.

The answer is, "Yes, such a function $U(x,y) = \text{const}$ exists if an only if $M(x,y)$ and $N(x,y)$ satisfy the Maxwell relations"

$$\frac{\partial M(x,y)}{\partial y} = \frac{\partial N(x,y)}{\partial x} \tag{20-12}$$

Then if Eq. 20-12 holds, the differential form Eq. 20-11 is called *an exact differential* and a U exists such that $dU = 0 = M(x,y)dx + N(x,y)dy$.

Integrating Factors and Thermodynamics

For fixed number of moles of ideal gas, the internal energy is a function of the temperature only, $U(T) - U(T_o) = C_V(T - T_o)$. Consider the heat that is transferred to a gas that changes it temperature and volume a very small amount:

$$\begin{aligned} dU =& C_V dT = \delta q + \delta w = \delta q - PdV \\ \delta q =& C_V dT + PdV \end{aligned} \tag{20-13}$$

Can a Heat Function $q(T,V) = \text{constant}$ be found?

To answer this, apply Maxwell's relations.

Homogeneous and Heterogeneous Linear ODES

A linear differential equation is one that does not contain any powers (greater than one) of the function or its derivatives. The most general form is:

$$Q(x)\frac{dy}{dx} + P(x)y = R(x) \tag{20-14}$$

Equation 20-15 can always be reduced to a simpler form by defining $p = P/Q$ and $r = R/Q$:

$$\frac{dy}{dx} + p(x)y = r(x) \tag{20-15}$$

If $r(x) = 0$, Eq. 20-15 is said to be a *homogeneous linear first-order ODE*; otherwise Eq. 20-15 is a *heterogeneous linear first-order ODE*.

The reason that the homogeneous equation is linear is because solutions can superimposed—that is, if $y_1(x)$ and $y_2(x)$ are solutions to Eq. 20-15, then $y_1(x)+y_2(x)$ is also a solution to Eq. 20-15. This is the case if the first derivative and the function are themselves linear. The heterogeneous equation is also called *linear* in this case, but it is important to remember that sums and/or multiples of heterogeneous solutions are also solutions to the heterogeneous equation.

It will be demonstrated below (directly and with a MATHEMATICA® example) that the homogeneous equation has a solution of the form

$$y(x) = \text{const } e^{-\int p(x)dx} \tag{20-16}$$

To show this form directly, the homogeneous equation can be written as

$$\frac{dy}{dx} = -p(x)y$$

Dividing each side through by through by y and integrate:

$$\int \frac{dy}{y} = \log y = -\int p(x)dx + \text{const}$$

which has solution

$$y(x) = \text{const} e^{-\int p(x)dx}$$

For the case of the heterogeneous first-order ODE, A trick (or, an integrating factor which amounts to the same thing) can be employed. Multiply both sides of the heterogeneous equation by $e^{\int p(x)}$:[12]

$$\exp\left[\int_a^x p(z)dz\right]\frac{dy(x)}{dx} + \exp\left[\int_a^x p(z)dz\right]p(x)y(x) = \exp\left[\int_a^x p(z)dz\right]r(x) \tag{20-17}$$

Notice that the left-hand-side can be written as a derivative of a simple expression

$$\exp\left[\int_a^x p(z)dz\right]\frac{dy(x)}{dx} + \exp\left[\int_a^x p(z)dz\right]p(x)y(x) = \frac{d}{dx}\left\{\exp\left[\int_a^x p(z)dz\right]y(x)\right\} \tag{20-18}$$

[12] The statistical definition of entropy is $S(T,V) = k\log\Omega(U(T,V))$ or $\Omega(U(T,V)) = \exp(S/k)$. Entropy plays the role of integrating factor.

3.016 Home

Full Screen

Close

Quit

therefore

$$\frac{d}{dx}\left\{\exp\left[\int_a^x p(z)dz\right] y(x)\right\} = \exp\left[p(x)\right] r(x) \tag{20-19}$$

which can be integrated and then solved for $y(x)$:

$$y(x) = \exp\left[-\int_a^x p(z)dz\right]\left\{y(x=a) + \int_a^x r(z)\exp\left[\int_a^z p(\eta)d\eta\right] dz\right\} \tag{20-20}$$

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Solutions to the General Homogeneous Linear First-Order ODE

The form of MATHEMATICA® 's solution for Eq. 20 is demonstrated.

```
DSolve[
 y'[x] + p[x] y[x] == 0,
 y[x], x]
```

1

$$\left\{\left\{y[x] \to e^{\int_1^x -p[K[1]]\,dK[1]}\, C[1]\right\}\right\}$$

The dummy integration variables (K[1] in the above) and any integration constants (C[1] above) are picked by Mathematica . Mathemat ica returns the most general form of homoge- neous linear first-order solutiion,

```
DSolve[
 y'[x] + (2 x + 1) y[x] == 0,
 y[x], x]
```

2

$$\left\{\left\{y[x] \to e^{-x-x^2}\, C[1]\right\}\right\}$$

There is an integration constant above, that will take on a specific value if an additional condition (such as an initial condition, or a boundary condition) is specified

```
DSolve[
 {y'[x] + (2 x + 1) y[x] == 0,
  y[0] == 4}, y[x], x]
```

3

$$\left\{\left\{y[x] \to 4\, e^{-x-x^2}\right\}\right\}$$

1: `DSolve` solves the linear homogeneous equation first-order ODE $dy/dx + p(x)y = 0$. Two variables are introduced in the solution: one is the 'dummy-variable' of the integration in Eq. 20 which MATHEMATICA® introduces in the form `K[N]` and an integration constant which is given the form `C[N]`.

2: Here, a specific $p(x)$ is given, so the dummy variable doesn't appear if $p(\zeta)$ can be integrated symbolically, as in this case for $p(\zeta) = 2x + 1$.

3: Furthermore, if enough boundary conditions are given to solve for the integration constants, then the `C[N]` are not needed either.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Solutions to the General Heterogeneous Linear First-Order ODE

This demonstrates the use of `DSolve` to find symbolic solutions of heterogeneous linear homogeneous first-order ODEs: Eq. 20-20. We will see how the homogeneous solution is always part of the sum for a heterogeneous solution.

1

2

```
DSolve[
 y'[x] + p[x] y[x] == r[x],
 y[x], x]
```

$$\left\{\left\{y[x] \to e^{\int_1^x -p[K[1]]\,dK[1]}\,C[1] + e^{\int_1^x -p[K[1]]\,dK[1]} \int_1^x e^{-\int_1^{K[2]} -p[K[1]]\,dK[1]}\,r[K[2]]\,dK[2]\right\}\right\}$$

The solution is general—two dummy integration variables and one constant of integration.

3

```
homsol =
 DSolve[y'[x] - y[x] == 0,
  y[x], x]
hetsol = DSolve[
  y'[x] - y[x] == e^2 x,
  y[x], x]
```

$$\{\{y[x] \to e^x\,C[1]\}\}$$

$$\{\{y[x] \to e^{2x} + e^x\,C[1]\}\}$$

1: `DSolve` solves the general linear heterogeneous equation, $dy/dx + p(x)y = r(x)$, to give the form Eq. 20-20. Note how the homogeneous solution (i.e., the part that depends on `C[1]`) is part of the solution.

2: This is an example for a specific case: $p(x) = -1$ and $r(x) = e^{2x}$. The homogeneous solution is displayed alongside to reinforce that it is always part of the solution.

3.016 Home

Full Screen

Close

Quit

Example: The Bernoulli Equation

The linear first-order ODEs always have a closed form solution in terms of integrals. In general non-linear ODEs do not have a general expression for their solution. However, there are some non-linear equations that can be reduced to a linear form; one such case is the Bernoulli equation:

$$\frac{dy}{dx} + p(x)\, y = r(x)\, y^a \tag{20-21}$$

Reduction relies on a clever change-of-variable, let $u(x) = [y(x)]^{1-a}$, then Eq. 20-21 becomes

$$\frac{du}{dx} + (1-a)p(x)\, u = (1-a)\, r(x) \tag{20-22}$$

which is a linear heterogeneous first-order ODE and has a closed-form solution.

However, not all non-linear problems can be converted to a linear form. In these cases, numerical methods are required.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Changing Variables in Symbolic Differential Equations

The Bernoulli equation, Eq. 20-21, is used to demonstrate how to change variables in an ODE.

```
BernoulliEquation =
 y'[x] + p[x] y[x] ==
  r[x] (y[x]) ^ (a)                                    1

yRep = u[x]^(1/(1-a))
DyRep = D[yRep, x]                                     2

step1 = BernoulliEquation /.
  {y[x] → yRep,
   y'[x] → DyRep}                                      3

step2 = PowerExpand[step1]                             4

step3 = Simplify[step2]                                5

BE = Solve[step3, u'[x]]                               6

uprime = u'[x] /. BE                                   7

usol =
 u[x] /. DSolve[u'[x] ==
    uprime[[1]], u[x], x]                              8

ysol =
 (usol[[1]]) ^ (1 / (1 - a))                           9

BernoulliEquation
Simplify[
 p[x] ysol + D[ysol, x]]                              10
```

1: The Bernoulli equation is a non-linear first order ODE, but a series of transformations can turn it into an equivalent linear form.

2: Symbols for what will be used as replacements for $y(x)$ and its derivative in *BernoulliEquation* are defined.

3: For `step1`, the symbols are used for a rule-replacement.

4: Using the form with replacements, the assumption that all variables are real is employed by using `PowerExpand`.

5: `Simplify` produces an equation for which the right-hand-side is zero; thus assuming that $u(x)$ is not identically zero, it can be factored out of the equation.

6: Using `Solve` (n.b, not `DSolve`) to find $u'(x)$ reveals the linear form of Bernoulli's equation in terms of the new variable.

7: The rule that is produced by `Solve` is used to extract the symbolic form of $u'(x)$; the symbolic form of $u'(x)$ is assigned to `uprime`.

8: To extract the solution (`usol`), we use the rule produced by `DSolve` on the equation $u'(x) =$ `usol`.

9: The back-transformation is used to find the general solution $y(x)$ to the non-linear form of the Bernoulli equation (`ysol`).

10: The solution, `ysol`, is plugged back into the left-hand-side of the Bernoulli equation and, with `Simplify`, is shown to be $r(x)$`ysol`^{a}.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Numerical Solutions to Non-linear First-Order ODEs

An example of computing the numerical approximation to the solution to a non-linear ODE is presented. The solutions are returned in the forms of a list of replacement rules to `InterpolatingFunction`. An `InterpolatingFunction` is a method to use numerical interpolation to extract an approximation for any point—it works just like a function and can be called on a variable like `InterpolatingFunction[0.2]`. In addition to the interpolation table, the definition specifies the domain over which the interpolation is considered valid.

Mathematica cannot find a direct solution to the following nonlinear ODE

```
DSolve[
 Sin[2 Pi y'[x]^2] ==
  y[x] x, y[x], x]
```
1

NDSolve is a numerical method for finding a solution. An initial condition and the desired range of solution are required.

```
solution = NDSolve[
  {Sin[2 Pi y'[x]^2] ==
    y[x] x, y[0] == 1},
  y, {x, 0, 3.5}]
```
2

```
{{y → InterpolatingFunction[
     {{0., 3.5}}, <>]},
 {y → InterpolatingFunction[
     {{0., 3.5}}, <>]}}
```

```
y[0.5] /. solution
```
3

```
{0.907437, 1.09733 + 0. i}
```

```
y[Pi] /. solution
```
4

```
{0.0524983,
 2.50186 - 0.61067 i}
```

1: This shows that `DSolve` cannot find a symbolic solution to $\sin[2\pi(y')^2] = y(x)x$.

2: Using NDSolve on a non-linear ODE, the solution is returned as a `InterpolatingFunction` replacement list. Note that there is a warning about "inverse functions" being used to find the solution; this is because of the sin-function which is causing Mathematica to assume a particular domain. There may be more solutions than the two that were that were returned as an `InterpolatingFunction`.

[: 3–4] This demonstrates how the numerical approximation to the non-linear ODE is obtained at particular values of x.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Plotting Numerical Solutions to Non-linear First-Order ODEs

This is an example of how to extract plot-table expressions from the rules for `InterpolationFunctions` that are returned from `NDSolve`.

```
PStyle = {{Red, Thick},
   {Darker[Green], Thick}};
```
1

```
PlotVanilla =
  Plot[Evaluate[y[x] /.
     solution], {x, 0, 3.5},
   PlotStyle → PStyle,
   PlotRange → {0, 2},
   PlotLabel → "Plot"];
```
2

```
PlotReal = Plot[Evaluate[
    Re[y[x] /. solution]],
  {x, 0, 3.5}, PlotStyle →
   PStyle, PlotLabel →
   "Real Part"];
```
3

```
PlotIm = Plot[Evaluate[
    Im[y[x] /. solution]],
  {x, 0, 3.5}, PlotStyle →
   PStyle, PlotLabel →
   "Imaginary Part"];
```
4

```
GraphicsRow[{PlotVanilla,
  PlotReal, PlotIm},
 ImageSize → Small]
```
5

1: Because `solution` obtained above is a list containing two rules, two curves will be plotted. Here we define a short-hand for the expression that will be passed to `PlotStyle` in the plots below. The first curve will be red, and the second will be `Darker` green.

2: Here, `Plot` is called on the `y[x]` with replacements defined the rule-set for `InterpolatingFunctions`, `solution`, that was obtained from `NDSolve` previously. Using `Evaluate` here immediately creates a list of length two, and plot recognizes this as two curves to which the `PlotStyles` can be applied. If `Evaluate` were not used, then both curves would be be red.

`Plot` only produces curves where the numerical value can be represented by a real number; if a solution has a point where it transforms from real to complex, `Plot` will show a curve that appears to end.

3–4: To determine the solution behavior, the real and imaginary parts are extracted with `Re` and `Im`.

5: This `GraphicsRow` indicates the solution behavior: the first solution is real over the domain where the interpolation is valid; the second solution transforms from real to complex near $x = 0.8$.

3.016 Home
Full Screen
Close
Quit

Lecture 21: Higher-Order Ordinary Differential Equations

Reading:
Kreyszig Sections: 2.1, 2.2 (pages45–52, 53–58)

Higher-Order Equations: Background

For first-order ordinary differential equations (ODEs), $F(y'(x), y(x), x)$, one value $y(x_o)$ was needed to specify a particular solution. Recall the example in Lecture 19 of a first-order differencing scheme: at each iteration the function grew proportionally to its current size. In the limit of very small forward differences, the scheme converged to exponential growth.

Now consider a situation in which function's current rate of growth increases proportionally to two terms: its current rate of growth and its size.

$$\text{Change in Value's Rate of Change} + \alpha\ (\text{the Value}) + \beta\ (\text{Value's Rate of Change}) = 0$$

To calculate a forward differencing scheme for this case, let Δ be the forward-differencing increment.

$$\left(\frac{\frac{F_{i+2}-F_{i+1}}{\Delta} - \frac{F_{i+1}-F_i}{\Delta}}{\Delta} \right) + \alpha F_i + \beta \left(\frac{F_{i+1} - F_i}{\Delta} \right) = 0$$

and then solve for the "next increment" F_{i+2} if F_{i+1} and F_i are known.

This indicates that, for second-order equations, two independent values are needed to generate the 'solution trajectory.'

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

A Second-Order Forward Differencing Example

A second order differencing formula is developed for second-order equations. The current value of the right-hand side is used, and therefore this is an explicit method.

```
ChangePerΔ[F_, i_, Δ_] :=
 (F[i + 1] - F[i]) / Δ
```
1

```
ChangeofChangeperΔ[
  F_, i_, Δ_] :=
 Simplify[(ChangePerΔ[
      F, i + 1, Δ] / Δ -
    ChangePerΔ[F, i, Δ])]
```
2

```
DifferenceRelation =
 ChangeofChangeperΔ[
   F, i, Δ] ==
 -β ChangePerΔ[F, i, Δ] -
  α F[i]
```
3

```
ForDiffSol = Collect[
  Solve[DifferenceRelation,
   F[i + 2]], Δ]
```
4

Replace to find the form of the solution

```
ForDiffSolV2 =
 ForDiffSol /. i → j - 2
```
5

```
{{F[j] →
    -α Δ^2 F[-2 + j] + F[-1 + j] +
     Δ (-F[-2 + j] + β F[-2 + j] +
        F[-1 + j] - β F[-1 + j])}}
```

1: *ChangeperΔ* is an example of a first-order finite difference approximation to the first derivative.

2: *ChangeofChangeperΔ* is the second order difference operation, it is obtained by applying the first-order difference operator twice. Note that two sequential values appear and that the differencing operator is proportional to $1/\Delta^2$. This is a general feature of high-order difference operators—with higher difference operations goes, the number of surrounding points required to evaluate the difference gets larger and larger (e.g., for the second order difference, function values are needed at three different i compared to two different i for the first-order case.

3: For a particular case of $d^2y/dx^2 = -\alpha dy/dx - \beta y$, the two difference operators replace the derivatives and a *difference relation* can be derived as a function of parameters α and β. We assign the equation representing the difference relation to *DifferenceRelation* .

4: The difference operator is derived by solving the difference relation for F_{i+2}—it will depend on the immediate last value F_{i+1} and that value's antecedent F_i. Therefore, any value—*including the first one calculated*—requires *two values* to be specified.

5: Typically, the current j–value is expressed in terms of the $(j-1)$ and $(j-2)$–values. This form is generated by the replacement $i \rightarrow j-2$.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

A Second-Order Forward Differencing Example

Iteration sequences are produced using the explicit differencing operators produced above. A particular example of a second-order method with constant coefficients is presented.

```
GrowList[ValuesList_List,
  Δ_ , α_ , β_] :=
 Module[{Minus1 =
    ValuesList[[-1, 2]],
   Minus2 = ValuesList[[-2,
     2]], LastX =
    ValuesList[[-1, 1]]},
  Append[ValuesList,
   {LastX + Δ,
    2*Minus1 - Minus2 +
     Δ*(β*(Minus2 -
           Minus1) -
        α*Δ*Minus2)}]]
```

1

```
InitVals =
  {{0, 1}, {.001, 1}};
```

2

```
result = GrowList[
  InitVals, .001, 1, .1]
```

3

```
result = GrowList[
  result, .001, 1, .1]
```

4

```
NestWhile[
 GrowList[#, .001, 1, .1] &,
 InitVals,
 (Last[#][[1]] < 0.02) &]
```

5

1: The difference operator is incorporated in *GrowList* : a function that grows a list (input as `ValuesList`) using a difference Δ and parameters α and β. The two previous values in the list become *localized variables* in a `Module` function. The `Module` returns a new list that is created using `Append` to place the current value at end of the input list.

2: Because two values are required, a list of size two must be provided. These values could represent the initial value and the initial value of the derivative.

3: Here is an example of using *GrowList* once.

4: Using `NestWhile` the list can be grown iteratively until the first value (i.e., the current value of $x_i = x_{i-1} + \Delta$) reaches are specified value (here 0.02).

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

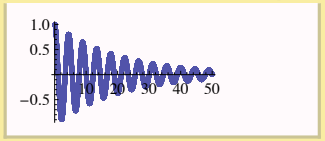
Visualization of Second-Order Forward Differencing

A second order differencing formula is developed for the case of constant growth and acceleration coefficients.

Visualize results out to x=50 (takes about 24 seconds to visualize)

```
ListPlot[
 NestWhile[GrowList[#,
    .001, 2, .1] &, InitVals,
  (Last[#][[1]] < 50) &]]
```

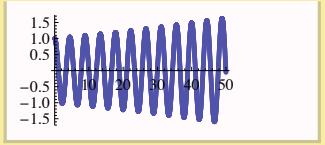
1



Change parameters for Growth Function (this example shows that the numerical solution does not converge to the accurate solution):

```
ListPlot[NestWhile[
  GrowList[#, 0.01, 2, 0] &,
  InitVals,
  (Last[#][[1]] < 50) &]]
```

2



1: `ListPlot` visualizes the results for growth constants $\alpha = 2$ and $\beta = 0.1$. `NestWhile` operates on a pure function constructed from *GrowList* until $x = 50$. The solution is oscillatory with fixed frequency, but with exponentially decreasing amplitude. This behavior is correct, although we have not analyzed the numerical stability or the accuracy of this method.

2: Here is another example with $\beta = 0$. With a small enough time step, this should mimic a harmonic oscillator. The increasing amplitude indicates a numerical inaccuracy at this time-step size.

3.016 Home

Full Screen

Close

Quit

Linear Differential Equations; Superposition in the Homogeneous Case

A linear differential equation is one for which the function and its derivatives are each linear—that is they appear in distinct terms and only to the first power. In the case of a homogeneous linear differential equation, the solutions are *superposable*. In other words, sums of solutions and their multiples are also solutions.

Therefore, a linear heterogeneous ordinary differential equation can be written as a product of general functions of the dependent variable and the derivatives for the n-order linear case:

$$\begin{aligned} 0 &= f_0(x) + f_1(x)\frac{dy}{dx} + f_2(x)\frac{d^2y}{dx^2} + \cdots + f_n(x)\frac{d^ny}{dx^n} \\ &= (f_0(x), f_1(x), f_2(x), \ldots, f_n(x)) \cdot \left(1, \frac{dy}{dx}, \frac{d^2y}{dx^2}, \ldots, \frac{d^ny}{dx^n}\right) \\ &= \vec{f}(x) \cdot \vec{D_n}y \end{aligned} \tag{21-1}$$

The homogeneous n^{th}-order linear ordinary differential equation is defined by $f_0(x) = 0$ in Eq. 21-1:

$$\begin{aligned} 0 &= f_1(x)\frac{dy}{dx} + f_2(x)\frac{d^2y}{dx^2} + \cdots + f_n(x)\frac{d^ny}{dx^n} \\ &= (0, f_1(x), f_2(x), \cdots, f_n(x)) \cdot \left(1, \frac{dy}{dx}, \frac{d^2y}{dx^2}, \ldots, \frac{d^ny}{dx^n}\right) \\ &= \vec{f_{hom}}(x) \cdot \vec{D_n}y \end{aligned} \tag{21-2}$$

Equation 21-1 can always be multiplied by $1/f_n(x)$ to generate the general form:

$$\begin{aligned} 0 &= F_0(x) + F_1(x)\frac{dy}{dx} + F_2(x)\frac{d^2y}{dx^2} + \cdots + \frac{d^ny}{dx^n} \\ &= (F_0(x), F_1(x), F_2(x), \ldots, 1)) \cdot (1, \frac{dy}{dx}, \frac{d^2y}{dx^2}, \ldots, \frac{d^ny}{dx^n}) \\ &= \vec{F}(x) \cdot \vec{D_n}y \end{aligned} \tag{21-3}$$

3.016 Home
Full Screen
Close
Quit

For the second-order linear ODE, the heterogeneous form can always be written as:

$$\frac{d^2y}{dx^2} + p(x)\frac{dy}{dx} + q(x)y = r(x) \tag{21-4}$$

and the homogeneous second-order linear ODE is:

$$\frac{d^2y}{dx^2} + p(x)\frac{dy}{dx} + q(x)y = 0 \tag{21-5}$$

Basis Solutions for the homogeneous second-order linear ODE

Because two values must be specified for each solution to a second order equation—the solution can be broken into two basic parts, each deriving from a different constant. These two independent solutions form a *basis pair* for any other solution to the homogeneous second-order linear ODE that derives from any other pair of specified values.

The idea is the following: suppose the solution to Eq. 21-5 is found the particular case of specified parameters $y(x = x_0) = A_0$ and $y(x = x_1) = A_1$, the solution $y(x; A_0, A_1)$ can be written as the sum of solutions to two *other problems*.

$$y(x; A_0, A_1) = y(x, A_0, 0) + y(x, 0, A_1) = y_1(x) + y_2(x) \tag{21-6}$$

where

$$\begin{aligned} y(x_0, A_0, 0) = A_0 \quad &\text{and} \quad y(x_1, A_0, 0) = 0 \\ y(x_0, 0, A_1) = 0 \quad &\text{and} \quad y(x_1, 0, A_1) = A_1 \end{aligned} \tag{21-7}$$

from these two solutions, any others can be generated.

The two arbitrary integration constants can be included in the definition of the general solution:

$$\begin{aligned} y(x) &= C_1 y_1(x) + C_1 y_2(x) \\ &= (C_1, C_2) \cdot (y_1, y_2) \end{aligned} \tag{21-8}$$

Second Order ODEs with Constant Coefficients

The most simple case—but one that results from models of many physical phenomena—is that functions in the homogeneous second-order linear ODE (Eq. 21-5) are constants:

$$a\frac{d^2y}{dx^2} + b\frac{dy}{dx} + cy = 0 \tag{21-9}$$

If two independent solutions can be obtained, then any solution can be formed from this basis pair.

Surmising solutions seems a sensible strategy, certainly for shrewd solution seekers. Suppose the solution is of the form $y(x) = \exp(\lambda x)$ and put it into Eq. 21-9:

$$(a\lambda^2 + b\lambda + c)e^{\lambda x} = 0 \tag{21-10}$$

which has solutions when and only when the quadratic equation $a\lambda^2 + \lambda x + c = 0$ has solutions for λ.

Because two solutions are needed and because the quadratic equation yields two solutions:

$$\begin{aligned} \lambda_+ &= \frac{-b + \sqrt{b^2 - 4ac}}{2a} \\ \lambda_- &= \frac{-b - \sqrt{b^2 - 4ac}}{2a} \end{aligned} \tag{21-11}$$

or by removing the redundant coefficient by diving through by a:

$$\begin{aligned} \lambda_+ &= \frac{-\beta}{2} + \sqrt{(\frac{\beta}{2})^2 - \gamma} \\ \lambda_- &= \frac{-\beta}{2} - \sqrt{(\frac{\beta}{2})^2 - \gamma} \end{aligned} \tag{21-12}$$

where $\beta \equiv b/a$ and $\gamma \equiv c/a$.

Therefore, any solution to Eq. 21-9 can be written as

$$y(x) = C_+ e^{\lambda_+ x} + C_- e^{\lambda_- x} \tag{21-13}$$

This solution recreated with a slightly different method in the following MATHEMATICA® example.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Deriving the Solutions to the Homogeneous Linear Second Order ODE with Constant Coefficients

Even though MATHEMATICA® is able to determine solutions to linear second-order ODEs with constant coefficients directly, it is still instructive to use MATHEMATICA® to derive these solutions.

```
TheODE[function_, var_] :=
 D[function[var],
   {var, 2}] +
  β D[function[var], var] +
  γ function[var]
```
1

```
TheODE[y, x]
```
2

Guess a solution and substitute it into the left-hand side of the ODE:

```
TheGuess[x_] := Exp[λ x]
```
3

```
TheODE[TheGuess, x]
```
4

This will be a solution when the resulting quadratic expression in λ is equal to 0:

```
λSolution = Solve[
  TheODE[TheGuess, x] == 0,
  λ]
```
5

```
{λMinus, λPlus} =
 λ /. λSolution
```
6

```
GenSol[x_] :=
 C[LPlus] Exp[λPlus x] +
  C[LMinus] Exp[λMinus x]
```
7

```
TheODE[GenSol, z]
```
8

```
Simplify[TheODE[GenSol, z]]
```
9

1–2: *TheODE* represents the left-hand side of any second-order ODE with constant coefficients—it is the differential representation of the second-order differencing method that was developed above. *TheODE* an argument for the name of the function (i.e., y) and the dependent variable (i.e., x in $y(x)$).

3: This will serve as a 'guess' of a solution—if we can find λ(s) that satisfy the ODE, then the solution(s) are determined.

4: The guess is inserted as the first argument to *TheODE* . The property of the exponential function, $de^{\alpha x}/dx = d(\text{guess})/dx = \alpha e^{\alpha x} = \alpha\text{guess}$ will permit factoring of the 'guessed' solution.

5: Using `Solve` with the guess inserted into *TheODE* will determine solution conditions on λ—this will be a quadratic equation in λ. The quadratic equation's solution ensures that, if the solution is complex, the two λ are complex conjugates.

6: By extraction the solution from the rules returned from `Solve`, assignments can be made to the two possible λ.

7: This is the form of the general solution in terms of two arbitrary constants.

8: This should show that the general solution always satisfies the ODE.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Characterizing the Solution Behavior for the Second-Order ODE with Constant Coefficients

Because the fundamental solution depend on only two parameters β and γ, the behavior (i.e., whether $\Re\lambda \gtrless 0$ and $\Im\lambda \stackrel{?}{=} 0$) of all solutions can be visualized in the γ-β plane.

```
RealsCond =
 Reduce[λPlus ∈ Reals &&
   λMinus ∈ Reals,
  {β, γ}, Reals]
```

1

The Complex and Real Domains — A

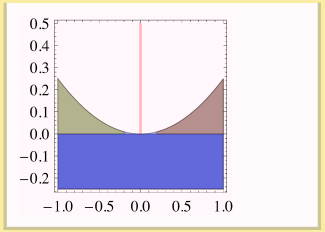
The Sign of the Real Part of Complex Solutions — B

Real Solutions, Both Positive or Both Negative — C

Real Solutions, Opposite Signs — D

```
Show[CplexPlot,
 RealRootsPos,
 MixedRealRoots,
 RealRootsNeg]
```

15



1: `Reduce` is a function for determining the conditions on parameters (here β and γ assumed to be real numbers) such that an expression satisfies particular constraints. The results from `Reduce` will be used to build up a graphical representation of solution behavior by incremental steps.

A: (Algorithm suppressed in class-notes) `RegionPlot` is used in conjunction with the results produced by `Reduce` to illustrate the domains where the λ are complex and real. The regions are annotated and saved as a graphic to be concatenated below. This will create a plot that distinguishes two regions in the γ-β plane: above $\gamma = \beta^2/4$, the λ are real; below, the λ are complex and oscillatory solutions appear (because $\exp(r + \imath\theta) = \exp(r)(\cos(\theta) + \imath\sin(\theta))$).

B: The sign of the real part of the complex λ changes at $\beta = 0$. Here a graphic and its notation are created.

C: Using `Reduce` and `RegionPlot` in a manner that parallels **A**, domains for the λ having two real roots with the same sign are plotted and annotated.

D: Finally, the domain when the λ are real with opposite sign are illustrated.

2: Concatenating all the graphical objects together with `Show` produces the image that was used to construct Fig. 21-21.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

The behavior of all solutions can be collected into a simple picture:

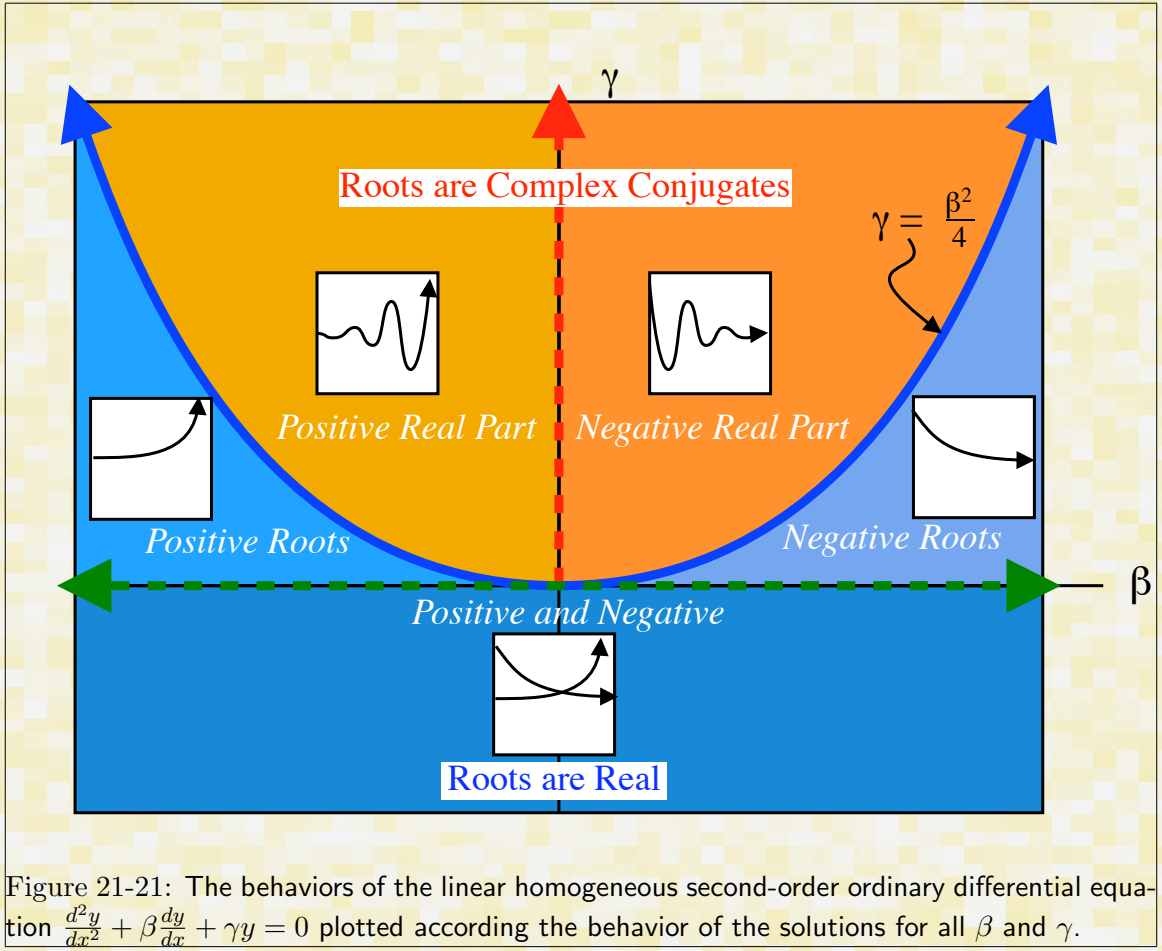


Figure 21-21: The behaviors of the linear homogeneous second-order ordinary differential equation $\frac{d^2y}{dx^2} + \beta\frac{dy}{dx} + \gamma y = 0$ plotted according the behavior of the solutions for all β and γ.

The case that separates the complex solutions from the real solutions, $\gamma = (\beta/2)^2$ must be treated separately, for the case $\gamma = (\beta/2)^2$ it can be shown that $y(x) = \exp(\beta x/2)$ and $y(x) = x\exp(\beta x/2)$ form an independent basis pair (see Kreyszig

3.016 Home
Full Screen
Close
Quit

AEM, p. 74).

Boundary Value Problems

It has been shown that all solutions to $\frac{d^2y}{dx^2} + \beta\frac{dy}{dx} + \gamma y = 0$ can be determined from a linear combination of the basis solution. Disregard for a moment whether the solution is complex or real, and ignoring the special case $\gamma = (\beta/2)^2$. The solution to any problem is given by

$$y(x) = C_+ e^{\lambda_+ x} + C_- e^{\lambda_- x} \tag{21-14}$$

How is a solution found for a particular problem? Recall that *two values* must be specified to get a solution—these two values are just enough so that the two constants C_+ and C_- can be obtained.

In many physical problems, these two conditions appear at the boundary of the domain. A typical problem is posed like this:

Solve

$$m\frac{d^2y(x)}{dx^2} + \nu\frac{dy(x)}{dx} + ky(x) = 0 \qquad \text{on } 0 < x < L \tag{21-15}$$

subject to the boundary conditions

$$y(x = 0) = 0 \qquad \text{and} \qquad y(x = L) = 1$$

or, solve

$$m\frac{d^2y(x)}{dx^2} + \nu\frac{dy(x)}{dx} + ky(x) = 0 \qquad \text{on } 0 < x < \infty \tag{21-16}$$

subject to the boundary conditions

$$y(x = 0) = 1 \qquad \text{and} \qquad y'(x = L) = 0$$

When the value of the function is specified at a point, these are called *Dirichlet* conditions; when the derivative is specified, the boundary condition is called a *Neumann* condition. It is possible have boundary conditions that are mixtures of Dirichlet and Neumann.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Determining Solution Constants from Boundary Values

Here is an example of taking the general solution with undetermined constants and using boundary conditions to determine a specific solution.

*Second order ODEs require that **two** conditions to generate a particular solution.*

```
GenSol[x]
```
1

Example of y(0) = 0 and y(L)=1:

```
SolutionOne =
 Solve[{GenSol[0] == 0,
   GenSol[L] == 1},
  {C[LPlus], C[LMinus]}]
```
2

Write the resulting solution:

```
SpecificSolutionOne =
 Simplify[
  GenSol[x] /. SolutionOne]
```
3

Second example: y(0) = 1 and y'(0)=0

```
DGen = D[GenSol[x], x]
```
4

Now the constants CPlus and CMinus are found by solving:

```
SolutionTwo =
 Solve[{GenSol[0] == 1,
   (DGen /. x → 0) == 0},
  {C[LPlus], C[LMinus]}]
```
5

```
SpecificSolutionTwo =
 Simplify[
  GenSol[x] /. SolutionTwo]
```
6

1: *GenlSol* is the solution to $y'' + \beta y' + \gamma y = 0$ with undetermined constants `Cplus` and `Cminus` that we derived above.

2: To demonstrate the method of using boundary conditions for a particular solution the boundary conditions, we use the example $y(0) = 0$ and $y(L) = 1$. `Solve` is used for the two conditions to find a rule-set for solutions in terms of the general-solution constants.

3: The form of the particular solution is obtained by back-substituting the solution for the constants into the general solution.

4: For application of a Neumann condition, the symbolic form of the derivative is required.

5–6: The particular solution for boundary conditions $y'(0) = y(0) = 0$ is obtained by inserting these equations into `Solve` and subsequent replacement into the general solution.

3.016 Home
Full Screen
Close
Quit

Another linear ODE that has important applications in materials science is that for the deflection of a beam. The beam deflection $y(x)$ is a linear fourth-order ODE:

$$\frac{d^2}{dx^2}\left(EI\frac{d^2y(x)}{dx^2}\right) = w(x) \tag{21-17}$$

where $w(x)$ is the load density (force per unit length of beam), E is Young's modulus of elasticity for the beam, and I is the moment of inertia of the cross section of the beam:

$$I = \int_{\mathrm{A}_{\times-sect}} y^2 dA \tag{21-18}$$

is the second-moment of the distribution of heights across the area.

If the moment of inertia and the Young's modulus do not depend on the position in the beam (the case for a uniform beam of homogeneous material), then the beam equation becomes:

$$EI\frac{d^4y(x)}{dx^4} = w(x) \tag{21-19}$$

The homogeneous solution can be obtained by inspection—it is a general cubic equation $y_{homog}(x) = C_0 + C_1x + C_2x^2 + C_3x^3$ which has the four constants that are expected from a fourth-order ODE.

The particular solution can be obtained by integrating $w(x)$ four times—if the constants of integration are included then the particular solution naturally contains the homogeneous solution.

The load density can be discontinuous or it can contain Dirac-delta functions $F_o\delta(x-x_o)$ representing a point load F_o applied at $x = x_o$.

It remains to determine the constants from boundary conditions. The boundary conditions can be determined because each derivative of $y(x)$ has a specific meaning as illustrated in Fig. 21-22.

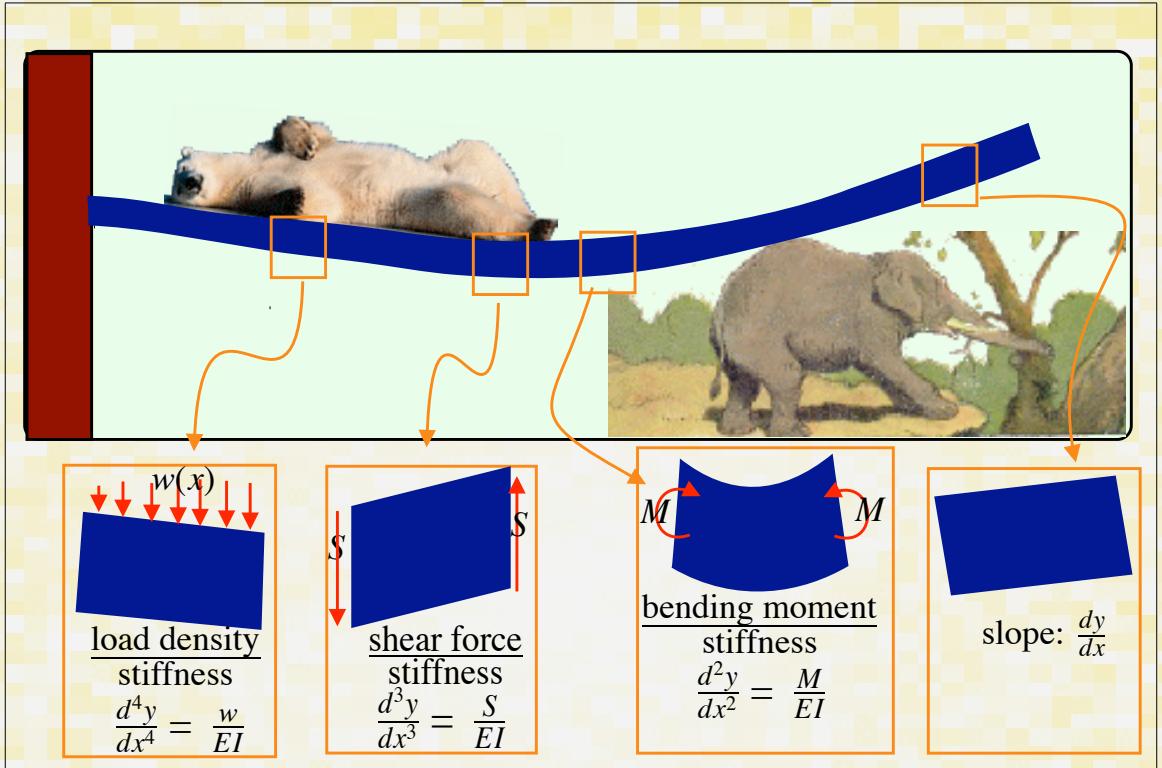


Figure 21-22: The shape of a loaded beam is determined by the loads applied over its length and its boundary conditions. The beam curvature is related to the local moment (imagine two handles rotated in opposite directions on a free beam) divided by the effective beam stiffness. Shear forces are related to the rate of change of moment along the beam.
(Polar Bear Photo Art Wolfe The Zone Network
http://classic.mountainzone.com/climbing/greenland/graphics/polar-bear.html)

There are common loading conditions that determine boundary conditions:

Free No applied moments or applied shearing force:

$$M = \left.\frac{d^2y}{dx^2}\right|_{boundary} = 0$$

$$S = \left.\frac{d^3y}{dx^3}\right|_{boundary} = 0$$

Point Loaded local applied moment, displacement specified.

$$M = \left.\frac{d^2y}{dx^2}\right|_{boundary} = M_o$$

$$y(x)|_{boundary} = y_o$$

Clamped Displacement specified, slope specified

$$\left.\frac{dy}{dx}\right|_{boundary} = s_o$$

$$y(x)|_{boundary} = y_o$$

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

A Function to Solve Beam Deflections for Common Boundary Conditions

A function for solving and visualizing the deflection of a uniform beam is developed for typical boundary conditions and load distributions

Set up the ODE solution (we will specify a unit beam stiffness) EI = 1

```
BeamEquation[y_ , x_ ,
  w_, BC1_ , BC2_] :=
 DSolve[{y''''[x] ==
     w[x], BC1, BC2},
   y[x], x] // Flatten
```

1

Functions for Typical Boundary Conditions and Beam Loading

A

```
BeamEquation[y, x,
 noload, Clamp[y, 0, 0, 0],
 Knob[y, 1, -.1, 0]]
```

10

$$\{y[x] \to -0.15\, x^2 + 0.05\, x^3\}$$

1: *BeamEquation* takes arguments for the (unknown) deflection y and its dependent argument x, a loading density $w(x)$, and boundary condition lists `BC1` and `BC2`, and uses `DSolve` to return replacement rules for a particular solution to the beam deflection equation (i.e., $d^4y/dx^4 = w(x)$).

A: The boundary conditions are defined as functions that return lists of equations for many common conditions (definitions suppressed in class-notes):

i *Clamp* `[y,x,position,slope]` fixed position and derivative at a specified point x of y.

ii *Knob* `[y,x,position,moment]` fixes the position and the moment at specified point.

iii *FreeBeam* `[y,x]` specifies that the beam is free to move at a specified point.

Particular types of applied loading functions ($w(x)$) are defined.

i *noload* `[x]`: no distributed load.

ii *unitload* `[x]`: unit (uniform) distributed load (i.e. 1 load unit/length).

iii *midload* `[x]` specifies that a (downward) load of magnitude 10 is applied at $x = 1/2$.

iv *boxload* `[x,position, width, magnitude]` specifies a uniform distributed load with `magnitude` over the domain `(position - width/2 < x < position + width/2)`

v *linearload* `[x]` specifies a linearly increasing load is applied, starting at -250 at $x = 0$ and increasing to +250 at $x = 1$.

10: An example of the use of *BeamEquation* with a clamp and a know is demonstrated.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualization of Beam Deflections

A graphical function is produced to visualize beam deflections for the specified boundary conditions and distributed loads.

```
Plot[Evaluate[
  y[x] /. BeamEquation[
    y, x, noload,
    Clamp[y, 0, 0, 0],
    Knob[y, 1, -.25, 0]]
], {x, 0, 1}]
```

1

BeamViz: Visualization Function

A

```
BeamViz[unitload,
 Clamp[y, 0, 0, 0],
 FreeBeam[y, 1]]
```

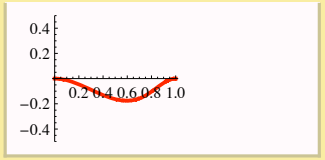
3

```
BeamViz[linearload,
 Knob[y, 0, 0, 0],
 Knob[y, 1, 0, 0]]
```

4

```
BeamViz[boxload[#,
   3 / 4, 1 / 8, -500] &,
 Clamp[y, 0, 0, 0],
 Clamp[y, 1, 0, 0]]
```

5



1: To plot the beam deflection, the solution condition is used as rule for replacement to $y(x)$

A: The function *BeamViz* collects the solution with the visualization for beams of unit normalized length, and uniform normalized stiffness EI.

3–4: These are examples of different loading conditions and boundary conditions are visualized as examples of *BeamViz* .

5: In this case, we have to force the loading condition $w(x)$ to be a function of a single variable. Because *boxload* takes four arguments, it is necessary to use it to construct a pure function.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

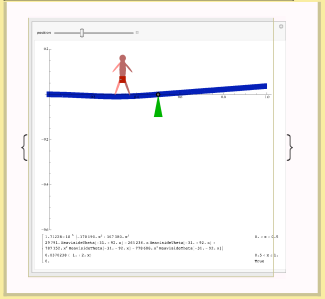
A Gratuitous Animations of Deflections of a Diving Board

A diving board can be approximated as a beam of uniform stiffness which is cantilevered at one end, and has an adjustable pin near its center. We work out a solution and visualization for a diver walking out on a diving board with the pin fixed at the center.

We wish to simulate the deflection of a diving board as a diver walks toward the end. A diving board may be modeled as beam with constant cross-section. The boundary conditions are that the board is clamped at the beginning; has a pivot located somewhere near the center; and the "diving" end . For "competitive" diving boards, the pivot point is adjustable. Here we fix the pivot at x=1/2.

This next function is a silly little graphic for the walking diver.

Graphics Functions for Animating Board and Diver **A**



A: (Solution and Algorithm Suppressed) We start by defining a set of graphics that take a single argument to represent the diver, the argument is designed to make the diver appear to be walking.

The solution to this deflection of the diving board is not trivial. The diver is modeled as a point load. The board is modeled as a piecewise solution for two beams that share a common boundary condition. We will need to match boundary conditions at the pin. The method is to find a solution in terms of an arbitrary coefficient, and then solve for the instance of that coefficient that makes the board deflection continuous.

If the diver is located between the cantilever and the pin, then the deflection in this region is determined with simple clamp and pin boundary conditions and the `DiracDelta` function for the distributed load. From this solution, the slope at the mid-point is calculated and this value is used as one of the boundary condition between the pin and the free-end.

If the diver is located between the pin and the free-end, the the slope at the pin is an undermined constant first in this region and then in the region between the cantilever and the pin. `Solve` is used to match the slope at the pin, and that solution is applied to the previously obtained solutions (i.e., the solutions from `DSolve`.

`Piecewise` is used in the definition of the solution-function that is returned.

`Manipulate` is used to specify the position of the diver.

3.016 Home

Full Screen

Close

Quit

Lecture 22: Differential Operators, Harmonic Oscillators

Reading:
Kreyszig Sections: 2.3,2.4, 2.7 (pages59–60, 61–69, 78–83)

Differential Operators

The idea of a function as "something" that takes a value (real, complex, vector, etc.) as "input" and returns "something else" as "output" should be very familiar and useful.

This idea can be generalized to *operators* that take a function as an argument and return another function.

The derivative operator operates on a function and returns another function that describes how the function changes:

$$\begin{aligned}
\mathcal{D}[f(x)] &= \frac{df}{dx} \\
\mathcal{D}[\mathcal{D}[f(x)]] = \mathcal{D}^2[f(x)] &= \frac{d^2 f}{dx^2} \\
\mathcal{D}^n[f(x)] &= \frac{d^n f}{dx^n} \\
\mathcal{D}[\alpha f(x)] &= \alpha D[f(x)] \\
\mathcal{D}[f(x) + g(x)] &= D[f(x)] + D[g(x)]
\end{aligned} \tag{22-1}$$

The last two equations above indicate that the "differential operator" is a linear operator.

The integration operator is the right-inverse of $\mathcal{D}$

$$\mathcal{D}[\mathcal{I}[f(x)]] = \mathcal{D}[\int f(x)dx] \tag{22-2}$$

but is only the left-inverse up to an arbitrary constant.

Consider the differential operator that returns a constant multiplied by itself

$$\mathcal{D}f(x) = \lambda f(x) \tag{22-3}$$

which is another way to write the the homogenous linear first-order ODE and has the same form as an eigenvalue equation. In fact, $f(x) = \exp(\lambda x)$, can be considered an *eigenfunction* of Eq. 22-3.

For the homogeneous second-order equation,

$$\left(\mathcal{D}^2 + \beta\mathcal{D} - \gamma\right)[f(x)] = 0 \tag{22-4}$$

It was determined that there were two eigensolutions that can be used to span the entire solution space:

$$f(x) = C_+ e^{\lambda_+ x} + C_- e^{\lambda_- x} \tag{22-5}$$

Operators can be used algebraically, consider the inhomogeneous second-order ODE

$$\left(a\mathcal{D}^2 + b\mathcal{D} + c\right)[y(x)] = x^3 \tag{22-6}$$

By treating the operator as an algebraic quantity, a solution can be found[13]

$$\begin{aligned} y(x) &= \left(\frac{1}{a\mathcal{D}^2 + b\mathcal{D} + c}\right)[x^3] \\ &= \left(\frac{1}{c} - \frac{b}{c^2}\mathcal{D} + \frac{b^2 - ac}{c^3}\mathcal{D}^2 - \frac{b(b^2 - 2ac)}{c^3}\mathcal{D}^3 + \mathcal{O}(\mathcal{D}^4)\right)x^3 \\ &= \frac{x^3}{c} - \frac{3bx^2}{c^2} + \frac{6(b^2 - ac)x}{c^3} - \frac{6b(b^2 - 2ac)}{c^3} \end{aligned} \tag{22-7}$$

[13] This method can be justified by plugging back into the original equation and verifying that the result is a solution.

which solves Eq. 22-6.

The Fourier transform is also a linear operator:

$$\begin{aligned}\mathcal{F}[f(x)] =& g(k) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} f(x)e^{\imath kx}dx \\ \mathcal{F}^{-1}[g(k)] =& f(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} g(k)e^{-\imath kx}dk\end{aligned} \tag{22-8}$$

Combining operators is another useful way to solve differential equations. Consider the Fourier transform, $\mathcal{F}$, operating on the differential operator, $\mathcal{D}$:

$$\mathcal{F}[\mathcal{D}[f]] = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{\infty} \frac{df(x)}{dx}e^{ikx}dx \tag{22-9}$$

Integrating by parts,

$$= \frac{1}{\sqrt{2\pi}}f(x)\,|_{x=-\infty}^{x=\infty} - \frac{\imath k}{\sqrt{2\pi}}\int_{-\infty}^{\infty} f(x)e^{ikx}dx \tag{22-10}$$

If the Fourier transform of $f(x)$ exists, then *typically*[14] $\lim_{x\to\pm\infty} f(x) = 0$. In this case,

$$\mathcal{F}[\mathcal{D}[f]] = -ik\mathcal{F}[f(x)] \tag{22-11}$$

and by extrapolation:

$$\begin{aligned}\mathcal{F}[\mathcal{D}^2[f]] &= -k^2\mathcal{F}[f(x)] \\ \mathcal{F}[\mathcal{D}^n[f]] &= (-1)^n\imath^n k^n\mathcal{F}[f(x)]\end{aligned} \tag{22-12}$$

Operational Solutions to ODEs

Consider the heterogeneous second-order linear ODE which represent a forced, damped, harmonic oscillator that will be discussed later in this lecture.

$$M\frac{d^2y(t)}{dt^2} + V\frac{dy(t)}{dt} + K_s y(t) = \cos(\omega_o t) \tag{22-13}$$

[14] It is not necessary that $\lim_{x\to\pm\infty} f(x) = 0$ for the Fourier transform to exist but it is satisfied in most every case. The condition that the Fourier transform exists is that $\int_{-\infty}^{\infty}|f(x)|dx$ exists and is bounded.

Apply a Fourier transform (mapping from the time (t) domain to a frequency (ω) domain) to both sides of 22-13:

$$\begin{aligned}\mathcal{F}[M\frac{d^2y(t)}{dt^2}+V\frac{dy(t)}{dt}+K_sy(t)] &= \mathcal{F}[\cos(\omega_o t)] \\ -M\omega^2\mathcal{F}[y]-\imath\omega V\mathcal{F}[y]+K_s\mathcal{F}[y] &= \sqrt{\frac{\pi}{2}}\left[\delta(\omega-\omega_o)+\delta(\omega+\omega_o)\right]\end{aligned} \tag{22-14}$$

because the Dirac Delta functions result from taking the Fourier transform of $\cos(\omega_o t)$.

Equation 22-14 can be solved for the Fourier transform:

$$\mathcal{F}[y] = \sqrt{\frac{-\pi}{2}}\frac{\left[\delta(\omega-\omega_o)+\delta(\omega+\omega_o)\right]}{M\omega^2+\imath\omega V-K_s} \tag{22-15}$$

In other words, the particular solution Eq. 22-13 can be obtained by finding the function $y(t)$ that has a Fourier transform equal the the right-hand-side of Eq. 22-15–or, equivalently, operating with the inverse Fourier transform on the right-hand-side of Eq. 22-15.

MATHEMATICA® does have built-in functions to take Fourier (and other kinds of) integral transforms. However, using operational calculus to solve ODEs is a bit clumsy in MATHEMATICA® . Nevertheless, it may be instructive to force it—if only as an an example of using a good tool for the wrong purpose.

3.016 Home

Full Screen

Close

Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Linear Operators and Derivatives

A check is made to see if `FourierTransform` obeys the rules of a linear operator (Eq. 22-1) and define rule-patterns for those cases where it doesn't.

Does Mathematica apply the Fourier Transform/Derivative Rule Automagically?

```
FourierTransform[
 D[f[x], {x, 1}], x, k]
```
1

Does Mathematica apply the rules according to the Fourier Transform being a linear operator?

```
FourierTransform[
 a f[x] + b g[x], x, k]
```
2

Apparently not--so we make some rules that can be applied. It may be instructive to see how to do this.

Two rules are defined for Linear Operators A

```
FourierTransform[
  a g[x] f[x] , x, k] //.
 ConstantRule
```
4

```
FourierTransform[
   a x f[x] + b v[x] g[x] +
    d p[x], x, k] //.
 DistributeRule //.
 ConstantRule
```
5

1: As of MATHEMATICA® 5.0, `FourierTransform` automatically implements Eqs. 22-12.

2: However, this will demonstrate that the "distribution-rule" isn't implemented automatically (n.b., although `Distribute` would implement this rule).

A: Define rules so that the FourierTransform acts as a linear functional operator (definitions suppressed in class-notes). *ConstantRule* is an example of a `RuleDelayed` (`:>`) that will allow replacement with patterns that will be evaluated when the rule is applied with `ReplaceAll` (`/.`); in this case, a `Condition` (`/;`) is appended to the rule so that those cofactors which don't depend on the transformation variable, x, can be identified with `FreeQ` and those that depend on x can be identified with `MemberQ`. *DistributeRule* uses `Distribute` to replace the Fourier transform of a sum with a sum of Fourier transforms.

4–5: The linear rules are dispatched by a `ReplaceRepeated` (`//.`) that will continue to use the replacement until the result stops changing. These are examples of $\mathcal{F}[ag(x)] = a\mathcal{F}[g(x)]$ and $\mathcal{F}[ag(x) + bh(x)] = a\mathcal{F}[g(x)] + b\mathcal{F}[h(x)]$.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Fourier Transforming the Linear-Damped-Forced Harmonic Oscillator Equation into the Frequency Domain

The left- and right-hand sides of the damped harmonic oscillator ODE are Fourier transformed, producing an algebraic equation between the the solution in Fourier-space and the Fourier k-parameter.

Here is the second-order ODE for a damped harmonic oscillator

```
ODE2nd =
 Mass D[y[t], {t, 2}] +
  Viscosity D[y[t], t] +
  SpringK y[t]
```

1

```
SpringK y[t] +
 Viscosity y'[t] + Mass y''[t]
```

Let's Fourier Transform the left-hand side of a second-order ODE:

```
FrrODE2nd = Factor[
  FourierTransform[
     ODE2nd, t, ω] //.
    DistributeRule //.
   ConstantRule
]
```

2

And now Fourier Transform the right-hand side for a prototype "forced" oscillator with an arbitary frequency ω0. (i.e., K y'' + η y' + m y = cos(ω0 t)

```
rhs = FourierTransform[
  Cos[ ω0 t] , t, ω]
```

3

1: This is the linear second-order for the "internal forcing" term of the harmonic oscillator. One could read this equation as Inertial Force+Frictional Force+Force to Restore to Minimal Potential Energy or $ma + \eta v + kx = m\ddot{x} + \eta\dot{x} + kx$

2: Fourier transforming (with `FourierTransform`) into the time domain converts the differential equation in the space domain into an algebraic equation in the time-domain.

3: If there is no "external force" on the harmonic oscillator, then the sum of the internal forces is zero. For the periodically-forced harmonic-oscillator, the right-hand-side of the equation can be expanded in a Fourier series. Here is a prototype of a right-hand-side, $\cos(\omega_o t)$, where ω_o is the forcing frequency. The forced-damped linear equation in the time-domain is obtained by transforming the external forces, or right-hand-side, of the harmonic oscillator.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Fourier Transform Solution to the Damped-Forced Linear Harmonic Oscillator

The harmonic oscillator is algebraically solved in the time-domain, and then the solution is back-transformed into the real-space domain.

```
ftsol =
 Solve[FrrODE2nd == rhs,
  FourierTransform[
   y[t], t, ω]]
```
1

```
DampedHOAssumptions =
  ω0 > 0 && Mass > 0 &&
   Viscosity > 0 &&
   SpringK > 0;
```
2

```
FullSimplify[
 InverseFourierTransform[
  FourierTransform[y[t], t,
    ω] /. Flatten[ftsol],
  ω, t], Assumptions →
  DampedHOAssumptions]
```
3

```
GenSol = DSolve[
  Mass D[y[t], {t, 2}] +
    Viscosity D[y[t], t] +
    SpringK y[t] ==
   Cos[ω₀ t], y[t], t]
```
4

```
FullSimplify[
 y[t] /. Flatten[GenSol],
 Assumptions →
  DampedHOAssumptions]
```
5

1: `Solve` is used to find the algebraic solution to the Fourier-transformed solution to the harmonic oscillator.

2: *DampedHOAssumptions* is a collection of physical solution that will be passed to `FullSimplify`.

3: The real-space solution is obtained with `InverseFourierTransform` operating on the general form FourierTransform[y[t], t, ω] as a pattern-replacement for the rule obtained by `Solve`. This produces only the *particular solution* (i.e., the homogeneous solutions that depend on undetermined constants is not part of the particular solution.)

4–5: Here, `DSolve` is used to produced the full solution for comparison to the Fourier technique. It is the solution to the homogeneous equation plus the particular solution that was obtained by the Fourier transform method. The solution is extracted from the solution-rule and simplified with the *DampedHOAssumptions* .

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Equally powerful is the concept of a *functional* which takes a function as an argument and returns a value. For example $\mathcal{S}[y(x)]$, defined below, operates on a function $y(x)$ and returns its surface of revolution's area for $0 < x < L$:

$$\mathcal{S}[y(x)] = 2\pi \int_0^L y\sqrt{1 + \left(\frac{dy}{dx}\right)^2}\, dx \tag{22-16}$$

This is the functional to be minimized for the question, "Of all surfaces of revolution that span from $y(x = 0)$ to $y(x = L)$, which is the $y(x)$ that has the smallest surface area?"

This idea of finding "which function maximizes or minimizes something" can be very powerful and practical.

Suppose you are asked to run an "up-hill" race from some starting point $(x = 0, y = 0)$ to some ending point $(x = 1, y = 1)$ and there is a ridge $h(x, y) = x^2$. Of all the possible routes, which is the shortest route $y(x)$? The solution $y(x)$ is called the geodesic.

As an introductory example, we write a functional that returns a scalar length associated with a curve $y(x)$ that starts at x_b and terminates at x_b.

$$F[y(x)] = \int_{(x_b, y(x_b))}^{(x_e, y(x_e))} ds = \int_{(x_b, y(x_b))}^{(x_e, y(x_e))} \sqrt{dx^2 + dy^2} = \int_{x_b}^{x_e} \sqrt{1 + \left(\frac{dy}{dx}\right)}\, dx \tag{22-17}$$

In Equation 22-17, F takes any $y(x)$ (with some technical restrictions, such as integrability) and returns the arc-length associated with that $y(x)$ between two fixed points. The geodesic is defined by the function that minimizes Equation 22-17

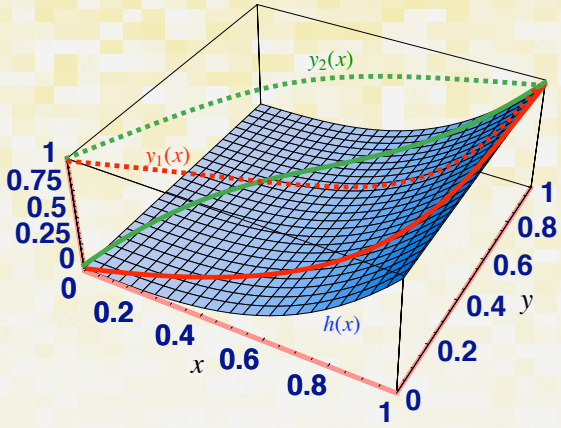


Figure 22-23: The terrain separating the starting point $(x = 0, y = 0)$ and ending point $(x = 1, y = 1)$. What is the shortest path between the starting and ending points? If the rate of climbing (or descending) is a known function of the slope, what is the quickest path? Assuming a model for how much running speed slows with the steepness of the path—which route would be quicker, one $(y_1(x))$ that starts going up-hill at first or another $(y_2(x))$ that initially traverses a lot of ground quickly?

These problems have some simularity to extrema in basic calculus—what is the parameter, variable, or point at which a given function is a maximum or a minimum. However, there is an important difference in the nature of the question that is being asked—what is the *function* that minimizes or maximizes a given functional. For basic calculus, the solution, or solutions, come from domain of possible solutions is the domain x over which the function $f(x)$ is defined (a line). In multivariable calculus, the solution(s) typically come from areas, volumes, and higher-dimensional spaces. For functionals, the solutions come from a "much larger" space. There is no obvious way to enumerate the set of trajectories that begin and end at a given point; such functions are uncountable.

The methods for finding such extremal functions derive from *variational calculus*, and the extension of basic calculus' derivative

3.016 Home

Full Screen

Close

Quit

to functionals is called the *variational derivative*.

Introduction to Variational Calculus: Variation of Parameters

To introduce the idea of variational calculus, we will minimize the functional Equation 22-17, but only for an enumerable set of functions.

Suppose the starting point is $x_b, y(x_b)) = (0, 0)$ and the ending point is $(1, 1)$. Instead of choosing from *all functions* that connect the two points, we consider a smaller set of quadratic polynomials:

$$y(x) = a + bx + cx^2 \tag{22-18}$$

The two boundary conditions x_b and x_e constrain two of the three parameters (a, b, c). Inserting Equation 22-18 into Equation 22-17, the problem is reduced to a basic calculus problem of minimizing over a single variable (e.g., b if the boundary conditions are used to solve for a and c). This method is demonstrated in the following examples.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Approximating the Geodesic

The quadratic polynomials (three parameters a, b, and c) is used to match boundary conditions, leaving a single parameter b. The constrained polynomial is used in the integral for total length, Equation 22-17.

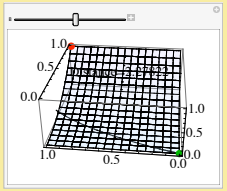
```
h = x^2;
$Assumptions = b ∈ Reals;
YGen = a + b x + c x^2;
```
1

Illustrate this surface/end points **A**

```
YBCs = YGen /. (Solve[
     {(YGen /. x → 0) == 0,
      (YGen /. x → 1) == 1},
     {a, c}] // Flatten)
```
3

```
TotalDistanceQuad =
 FullSimplify[
  Integrate[Sqrt[
    1 + (D[YBCs, x])^2 +
     (D[h, x])^2],
   {x, 0, 1}]]
```
4

Interactive Path/Length **B**



1: The shape of the surface over which the trajectories is defined as a function of x, as is the general quadratic that will be used as the function for variation of parameters. Here, we use a kernel default-assumption, `$Assumptions`, that will be automatically for functions such as `Integrate` and `Simplify`.

A: A graphic, *TheSurface* , is constructed with indicated initial- and end-points.

3: Here, the quadratic approximation is constrained to its initial- and end-points with replacement and the rule produced by `Solve`.

4: The will produce a closed form for the total distance as a function of a single parameter, b

B: `Manipulate` is used to produce an interactive graphic that illustrates the quadratic approximation as a function of b and the computed length.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Variation of Parameters for the Geodesic Approximation

This example shows that the quadratic approximation obtained by variation of parameters is close to the exact geodesic that is calculated by the calculus of variations. The method to find the exact geodesic is described below.

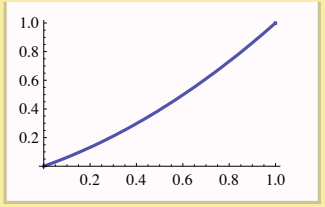
```
Plot[TotalDistanceQuad,
 {b, -2, 6}]
```
1

```
BminsolGeodesicQuad =
 FindMinimum[
  TotalDistanceQuad,
  {b, 0, 1}]
```
2

Use the minimizing b to find the approximation.

```
GeodesicQuadSolution =
 YBCs /.
  BminsolGeodesicQuad[[2]]
```
3

```
GeodesicQuadPlot =
 Plot[GeodesicQuadSolution,
  {x, 0, 1},
  PlotStyle → Thick]
```
4



1: Plotting the remaining parameter shows that a minimum exists.

2: `FindMinimum` returns a list with the minimal value and a rule for the minimizing b.

3–4: Using the minimizing rule for b, we can replace b in the constrained quadratic approximation and plot it. This is the quadratic approximation to the geodesic for the given surface and boundary conditions.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Comparison of the Approximation to the Exact Geodisic

The quadratic polynomial is shown to have a mininum length with respect to a single unconstrained parameter. The minimizing approximation is computed and visualized.

The exact minimizing path can be found by using Calculus of Variations (demonstrated below). The solution is obtained from a boundary-value problem which we do not take up at this point, but it is interesting to see the exact solution and compare it with the approximate one we obtained above. The closed-form expression for the function that minimizes the climbing time is:

```
GeodesicExact =
  (2 x Sqrt[1 + 4 x^2] + ArcSinh[2 x]) / (2 Sqrt[5] + ArcSinh[2])
```

1

Graphical Comparisons A

```
Distance[f_] := Integrate[
  Sqrt[1 + (D[f, x])^2 +
    (D[h, x])^2], {x, 0, 1}]
```

4

```
Distance[GeodesicExact]
```

5

```
Distance[GeodesicExact] <
 Distance[
  GeodesicQuadSolution] <
 Distance[x]
```

6

```
True
```

1: *GeodesicExact* is the exact geodesic for the specified surface ($h = x^2$) and boundary conditions. (This calculation is provided in a subsequent example).

A: Visual comparisons between the approximation and the exact solution show that the approximation is quite good. This is not a general rule, and we cannot know in advance if an approximation by variation of parameters will be good or not.

4–5: The functional is encoded in this `Distance` function, which takes a function of x as an argument and integrates over $0 < x < 1$.

6: This shows that the geodesic is shorter than the approximation, and the approximation is shorter than a straight line ($y(x) = x$) projected onto the x–y plane.

3.016 Home

Full Screen

Close

Quit

The geodesic gave the shortest-distance path between two points—a related question is, "Given a velocity, what is the quickest (shortest time) path between two points?" The answer is related the *brachistochrone* which is the path of most rapid descent with constant acceleration. I don't know what to call the shortest-time path, so I am making up a name " *brachiostone*". Perhaps a better name would be the *Fermatic*, becase the curve is related to a generalized Fermat's theorem. However, this could be confused with *fermata* which is a pause of unspecified length; so I suppose that *MiniFermatizoid* might be the best choice of all. However, in future editions to these notes, I am going to change the name to *Brakkes' Chrone*, in honor of one of my heros, *Ken Brakke* http://www.susqu.edu/brakke/. Neologisms are so entertaining—and a delightful waste of time.

For traversing a hill, it is a reasonable model for running speed to be a decreasing function of climbing-angle α, and to have the speed fall to zero when the trajectory is "straight-up." Thus, we select a model such as

$$v(s) = \cos(\alpha(s)) \tag{22-19}$$

where s is the arclength along the path. The maximum speed occurs on flat ground $\alpha = 0$ and running speed monotonically falls to zero as $\alpha \to \pi/2$. To calculate the time required to traverse *any* path $y(x)$ with endpoints $y(0) = 0$ and $y(1) = 1$,

$$\frac{ds}{dt} = v(s) = \cos(\alpha(s)) = \frac{\text{local horizontal}}{\text{local arclength}} = \frac{\sqrt{dx^2 + \frac{dy}{dx}^2}}{\sqrt{dx^2 + dy^2 + dh^2}}$$

$$\text{therefore } dt = \frac{ds}{v(s)} = \frac{dx^2 + dy^2 + dz^2}{\sqrt{dy^2 + dx^2}} = \frac{1 + \frac{dy}{dx}^2 + \frac{dh}{dx}^2}{\sqrt{1 + \frac{dy}{dx}^2}}\, dx \tag{22-20}$$

$$\text{therefore time}[y(x)] = \int_{x_b}^{x_e} \frac{1 + \frac{dy}{dx}^2 + \frac{dh}{dx}^2}{\sqrt{1 + \frac{dy}{dx}^2}}\, dx$$

Tthe hill $h(x) = x^2$ can be inserted into Equation 22-20 for the time as a functional of the path between fixed points $(0, 0, 0)$ and $(1, 1, 1)$.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Approximating the Brachiostone by Variation of Parameters

The same method for finding an approximation to the geodesic is applied to the minimum-time functional in Equation 22-20. (See preceding text on definition of brachiostone.)

```
TotalTimeQuad =
 FullSimplify[Integrate[
   (1 + D[YBCs, x]^2 +
      D[h, x]^2) /
    Sqrt[1 + D[YBCs, x]^2],
   {x, 0, 1}],
  Assumptions → b ≠ 1]
```
1

Visualizing the Approximation to the Brachiostone A

```
Plot[TotalTimeQuad,
 {b, -2, 2}]
```
4

```
BminsolBrachioQuad =
 FindMinimum[
  TotalTimeQuad, {b, 0, 1}]
```
5

```
BrachioQuadSolution =
 YBCs /.
  BminsolBrachioQuad[[2]]
```
6

```
BrachioQuadPlot =
 Plot[BrachioQuadSolution,
  {x, 0, 1},
  PlotStyle → Thick]
```
7

1: The same quadratic (constrained to the boundary conditions) as was used for the geodesic is utilized for the brachiostone (Equation 22-20). In this case, there is a closed-form solution for the undetermined parameter, but this not typical for other functionals.

A: The brachiostone approximation is visualized by superposing onto the "hill" with the exact geodesic.

4: Plotting the time as a function of b indicates that there is a minimizing b.

5–7: The minimizing b is inserted back into the quadratric approximation

3.016 Home

Full Screen

Close

Quit

Suppose the functional depends on one function of single variable, $y(x)$, and its derivative, $y'(x)$. Furthermore, consider the fixed end-point problem (i.e., $y(x_b) = y_b$ and $y(x_e) = y_e$ are specified.

The general form of the functional is:

$$F[y(x)] = \int_{x_b}^{x_e} f[x, y(x), y'(x)]\, dx \tag{22-21}$$

We want to introduce a notation for functions that are "nearby" to a function $y(x)$[15] To do this, let a function near to $y(x)$ be described as $y(x) + v(x)\Delta t$. (It may be useful to think of t as a time-like variable and $v(x)$ is the instantaneous local velocity away from $y(x)$; but, generally, t, could be any scalar parameter.) Because $y(x)$ is assumed to match the boundary conditions, any admissible variation $y + v\Delta t$ must also match the boundary conditions, so the 'velocity' $v(x)$ at the boundaries must vanish. Therefore, for functions near to $y(x)$,

$$F[y + v\Delta t] = \int_{x_b}^{x_e} f[x, y(x) + v(x)\Delta t, y'(x) + v'(x)\Delta t]\, dx \tag{22-22}$$

Both sides depend on the scalar quantity Δt, and so we will expand about $\Delta t = 0$. We treat the integrand $f(x, y, y')$ as a function of three variables (it is afterall, because f is being evaluated pointwise in the integral). Therefore, partial derivative must appear in the expansion:

$$F[y] + \left.\frac{\delta F}{\delta y}\right|_{\Delta t=0} v\Delta t = \int_{x_b}^{x_e} f[x, y(x), y'(x)]\, dx + \int_{x_b}^{x_e} \left[\frac{\partial f}{\partial y} v(x) + \frac{\partial f}{\partial y'} v'(x)\right] \Bigg|_{\Delta t=0} \Delta t\, dx \tag{22-23}$$

where we use a "δ" to indicate the *variational derivative* of a functional. Cancelling common terms and integrating by parts,

$$\left.\frac{\delta F}{\delta y}\right|_{\Delta t=0} v\Delta t = \Delta t \left\{ y(x)v(x) \Big|_{x_b}^{x_e} + \int_{x_b}^{x_e} \left[\left(\frac{\partial f}{\partial y} - \frac{d}{dx}\frac{\partial f}{\partial y'}\right) \Bigg|_{\Delta t=0} v(x)\right] dx \right\} \tag{22-24}$$

[15] A precise definition of "closeness" of functions is somewhat arbitrary and depends on the 'norm' defined for functions (and because gradients have a length and a direction, variational gradients also depend on the norm). Typically, variational calculus is introduced with the *l2-norm*, $f(x) \cdot g(x) \equiv \int f(x)g(x)dx$, and the resulting variation becomes $\delta F \cdot v\Delta t = \int [\partial F/\partial y - (d/dx)\partial F/\partial y']v\Delta t$ which, for any h that satifies the boundary conditions, can equal zero only if the integrand of the variation vanishes (i.e, if the variation is 'orthogonal' to an arbitrary h. For several applications of other norms to materials science, see "Variational methods for microstructural-evolution theories", Carter W.C., Taylor J.E, Cahn J.W., JOM (Journal of the Materials Soc.), 49(12) 30–36, 1997

Because $v(x)$ must vanish at the end-points, and because the terms that are being evaluated at $t = 0$ do not depend on t, then

$$\frac{\delta F}{\delta y} \cdot v = \int_{x_b}^{x_e} \left[\frac{\partial f}{\partial y} - \frac{d}{dx}\frac{\partial f}{\partial y'} \right] v(x)\, dx \tag{22-25}$$

Because $v(x)$ is arbitrary (except for satisying the boundary conditions), the only way that the functional derivative can vanish is for

$$\frac{\partial f}{\partial y} - \frac{d}{dx}\frac{\partial f}{\partial y'} = 0 \tag{22-26}$$

which is called the *Euler equation* and is the condition for a functional to be extremal with respect to a variation of its function-argument, $y(x)$.

We could also think of Equation 22-25 as representing the integral-sum of the instantaneous changes in the scalar value of the functional as its function $y(x)$ changes. The functional is stationary (i.e., a necessary condition for an extremum) if the variational derivative vanishes everywhere on $x_b < x < x_e$. Because we have a condition as a function of a single variable, the form of Euler's equation in Equation 22-26 is an ordinary differential equation of derivatives of $y(x)$ in x.

For example, consider the geodesic problem from the above example on the surface $h(x) = x^2$, with fixed end-points $y(x = 0) = 0$ and $y(x = 1) = 1$. The functional is

$$F[y(x)] = \int_0^1 \sqrt{1 + \frac{dy}{dx}^2 + \frac{dh}{dx}^2}\, dx = \int_0^1 \sqrt{1 + \frac{dy}{dx}^2 + 4x^2}\, dx \tag{22-27}$$

therefore,

$$\begin{aligned} \frac{\partial f}{\partial y} =& 0 \text{ and } \frac{\partial f}{\partial y'} = \frac{\frac{dy}{dx}}{\sqrt{1 + 4x^2 + \frac{dy}{dx}^2}} \\ \frac{d}{dx}\frac{\partial f}{\partial y'} =& \frac{\frac{d^2y}{dx^2}}{\sqrt{1 + 4x^2 + \frac{dy}{dx}^2}} - \frac{\frac{dy}{dx}\left(8x + 2\frac{dy}{dx}\frac{d^2y}{dx^2}\right)}{\left(1 + 4x^2 + \frac{dy}{dx}^2\right)^{3/2}} \\ =& \frac{\left(1 + 4x^2\right)\frac{d^2y}{dx^2} - 4x\frac{dy}{dx}}{\left(1 + 4x^2 + \frac{dy}{dx}^2\right)^{3/2}} \end{aligned} \tag{22-28}$$

The Euler equation becomes

$$\frac{4x\frac{dy}{dx} - (1+4x^2)\frac{d^2y}{dx^2}}{(1+4x^2+\frac{dy}{dx}^2)^{3/2}} = 0 \qquad (22\text{-}29)$$

The numerator can be set equal to zero, and the result is an integrable second-order linear ODE.

This and the example for the brachiostone is demonstrated in the following examples.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Euler's equation and Exact Solution to Geodesic

The variational derivative from the package `VariationalMethods` is used with `DSolve` to calculate the exact geodesic for which an approximate solution was found above.

```
Needs[
 "VariationalMethods`"]
```
1

```
DistanceIntegrand = Sqrt[
  (1 + (D[y[x], x])^2 +
    (D[h, x])^2)]
```
2

```
VariationalD[
 DistanceIntegrand, y[x], x]
```
3

```
DistanceExtremalCondition =
 EulerEquations[
  DistanceIntegrand,
 y[x], x]
```
4

```
DistanceMinimizingFunction
 = DSolve[
  {DistanceExtremalConditi\
    on, y[0] == 0,
   y[1] == 1}, y[x], x]
```
5

```
DistanceYExactSolution =
y[x] /.
 DistanceMinimizingFuncti\
   on[[1]]
```
6

1: The `VariationalMethods` package has functions for many methods in the calculus of variations. We will use only a few of the simpler methods in this and the following example.

2: *DistanceIntegrand* is the integrand for total length; it will play the role of $f[x, y, y']$ in Equation 22-26

3: `VariationalD` computes the right-hand-side of the expression in Equation 22-25.

4: `EulerEquations` gives the equations, for a given integrand, for the extremal solution. *DistanceExtremalCondition* will be the ordinary differential equation, Equation 22-28.

4–5: Using `DSolve` to solve Euler's equation with the boundary conditions produces the exact solution.

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Euler's equation and Numerical Solution to Brachiostone

The Euler's equation for the total-time function defined in Equation 22-20 is solved by numerical methods.

This is the form of the total time's integrand as derived above for v(s) = cos(α)

```
TimeIntegrand =
 (1 + (D[y[x], x])^2 +
    (D[h, x])^2) /
  Sqrt[1 + (D[y[x], x])^2]
```
1

```
TimeExtremalCondition =
 EulerEquations[
  TimeIntegrand, y[x], x]
```
2

This ODE doesn't have a closed-form solution; so we find a numerical approximation to the solution to the Euler equation, extract it and then plot it.

```
BrachioMinimizerNumerical =
 NDSolve[
  {TimeExtremalCondition,
   y[0] == 0,
   y[1] == 1}, y[x], x]
```
3

```
BrachioNumerical = y[x] /.
  BrachioMinimizerNumerical
   [[1]]
```
4

1: *TimeIntegrand* is the term that was derived in Equation 22-20 and used in the approximate method for the brachiostone calculation.

2: The Euler equations for this integrand produces a non-linear second-order ODE that doesn't have a closed form solution.

3–4: However, a numerical solver `NDSolve` can be employed. In this case `NDSolve` runs into a few numerical difficulties around $x = 0.5$, but it produces a very reasonable solution that 'beats' the approximation given above.

3.016 Home
Full Screen
Close
Quit

notebook (non-evaluated) pdf (evaluated, color) pdf (evaluated, b&w) html (evaluated)

Visualizing the Brachiostone and Comparison to the Approximation Obtained by Variation of Parameters

The numerical solution obtained above is plotted and compared to the approximate solution.

```
BrachioExactPlot =
 Plot[BrachioNumerical,
  {x, 0, 1}, PlotStyle →
   {Thick, Darker[Green]}]
```
1

```
GraphicsRow[
 {Show[BrachioQuadPlot,
   BrachioExactPlot] ,
  Show[ Plot[
    BrachioNumerical -
     BrachioQuadSolution,
    {x, 0, 1}, PlotStyle →
     {Thickness[0.005],
      Hue[1]}]]}]
```
2

```
Time[f_] :=
 Integrate[TimeIntegrand /.
   y'[x] → D[f, x],
  {x, 0, 1.0}]
```
3

```
N[Time[BrachioNumerical]] <
 Chop[Time[
   BrachioQuadSolution]] <
 Time[x]
```
4

```
True
```

1–2: This visualized the computed brachistone. It indicates that a better (and reasonable) strategy that it is advantageous to run up-hill when the slope is small, and then traverse over a longer distance with reduced slope (as in a "switch-back" in a hiking trail). In this case, the quadratic approximation is still quite good, but not as good as in the geodesic.

3: This function takes a function of x for an argument and returns the total time assuming the $v(s) = \cos\alpha(s)$ model on a hill given by $h = x^2$.

4: This demonstrate that the numerical solution to the Euler equation has a shorter time than the quadratic approximation which, in turn, is shorter than the "projected-straight-line." In this inequality, we use `N` with `Integrate` which is equivalent to using `NIntegrate`. The quadratic solution has a small imaginary part that arises from numerical imprecision—this is removed with `Chop`.

3.016 Home
◀◀ ◀ ▶ ▶▶
Full Screen
Close
Quit

Methods for finding general solution to the linear inhomogeneous second-order ODE

$$a\frac{d^2y(t)}{dt^2}+b\frac{dy(t)}{dt}+cy(t)=F(t) \tag{22-30}$$

have been developed and worked out in MATHEMATICA® examples.

Eq. 22-30 arises frequently in physical models, among the most common are:

$$\begin{aligned}\text{Electrical circuits:}\quad & L\frac{d^2I(t)}{dt^2}+\rho l_o\frac{dI(t)}{dt}+\frac{1}{C}I(t)=V(t)\\ \text{Mechanical oscillators:}\quad & M\frac{d^2y(t)}{dt^2}+\eta l_o\frac{dy(t)}{dt}+K_s y(t)=F_{app}(t)\end{aligned} \tag{22-31}$$

where:

	Mechanical	Electrical
Second Order	**Mass** M: Physical measure of the ratio of momentum field to velocity	**Inductance** L: Physical measure of the ratio of stored magnetic field to current
First Order	**Drag Coefficient** $c=\eta l_o$ (η is viscosity l_o is a unit displacement): Physical measure of the ratio environmental resisting forces to velocity—or proportionality constant for energy dissipation with square of velocity	**Resistance** $R=\rho l_o$ (ρ is resistance per unit material length l_o is a unit length): Physical measure of the ratio of voltage drop to current—or proportionality constant for power dissipated with square of the current.
Zeroth Order	**Spring Constant** K_s: Physical measure of the ratio environmental force developed to displacement—or proportionality constant for energy stored with square of displacement	**Inverse Capacitance** $1/C$: Physical measure of the ratio of voltage storage rate to current—or proportionality constant for energy storage rate dissipated with square of the current.
Forcing Term	**Applied Voltage** $V(t)$: Voltage applied to circuit as a function of time.	**Applied Force** $F(t)$: Force applied to oscillator as a function of time.

For the homogeneous equations (i.e. no applied forces or voltages) the solutions for physically allowable values of the coefficients can either be oscillatory, oscillatory with damped amplitudes, or, completely damped with no oscillations. (See Figure 21-21). The homogeneous equations are sometimes called *autonomous* equations—or *autonomous systems*.

Simple Undamped Harmonic Oscillator

The simplest version of a homogeneous Eq. 22-30 with no damping coefficient ($b = 0$, $R = 0$, or $\eta = 0$) appears in a remarkably wide variety of physical models. This simplest physical model is a simple harmonic oscillator—composed of a mass accelerating with a linear spring restoring force:

$$\begin{aligned} \text{Inertial Force} &= \text{Restoring Force} \\ M\,\text{Acceleration} &= \text{Spring Force} \\ M\frac{d^2y(t)}{dt^2} &= -K_s y(t) \\ M\frac{d^2y(t)}{dt^2} + K_s y(t) &= 0 \end{aligned} \tag{22-32}$$

Here y is the displacement from the equilibrium position–i.e., the position where the force, $F = -dU/dx = 0$. Eq. 22-32 has solutions that oscillate in time with frequency ω:

$$\begin{aligned} y(t) &= A\cos\omega t + B\sin\omega t \\ y(t) &= C\sin(\omega t + \phi) \end{aligned} \tag{22-33}$$

where $\omega = \sqrt{K_s/M}$ is the natural frequency of oscillation, A and B are integration constants written as amplitudes; or, C and ϕ are integration constants written as an amplitude and a phase shift.

3.016 Home
Full Screen
Close
Quit

The simple harmonic oscillator has an *invariant*, for the case of mass-spring system the invariant is the total energy:

$$\begin{aligned}
\text{Kinetic Energy} + \text{Potential Energy} &= \\
\frac{M}{2}v^2 + \frac{K_s}{2}y^2 &= \\
\frac{M}{2}\frac{dy}{dt}^2 + \frac{K_s}{2}y^2 &= \\
A^2\omega^2\frac{M}{2}\cos^2(\omega t+\phi) + A^2\frac{K_s}{2}\sin^2(\omega t+\phi) &= \\
A^2(\omega^2\frac{M}{2}\cos^2(\omega t+\phi) + \frac{M\omega^2}{2}\sin^2(\omega t+\phi) &= \\
A^2M\omega^2 &= \text{constant}
\end{aligned} \tag{22-34}$$

There are a remarkable number of physical systems that can be reduced to a simple harmonic oscillator (i.e., the model can be reduced to Eq. 22-32). Each such system has an analog to a mass, to a spring constant, and thus to a natural frequency. Furthermore, every such system will have an invariant that is an analog to the total energy—an in many cases the invariant will, in fact, be the total energy.

The advantage of reducing a physical model to a harmonic oscillator is that *all of the physics follows from the simple harmonic oscillator.*

Here are a few examples of systems that can be reduced to simple harmonic oscillators:

Pendulum By equating the rate of change of angular momentum equal to the torque, the equation for pendulum motion can be derived:

$$MR^2\frac{d^2\theta}{dt^2} + MgR\sin\theta = 0 \tag{22-35}$$

for small-amplitude pendulum oscillations, $\sin(\theta) \approx \theta$, the equation is the same as a simple harmonic oscillator.

It is instructive to consider the invariant for the non-linear equation. Because

$$\frac{d^2\theta}{dt^2} = \frac{d\theta}{dt}\left(\frac{d\frac{d\theta}{dt}}{d\theta}\right) \tag{22-36}$$

Eq. 22-35 can be written as:

$$MR^2\frac{d\theta}{dt}\left(\frac{d\frac{d\theta}{dt}}{d\theta}\right)+MgR\sin(\theta)=0 \tag{22-37}$$

$$\frac{d}{d\theta}\left[\frac{MR^2}{2}\left(\frac{d\theta}{dt}\right)^2-MgR\cos(\theta)\right]=0 \tag{22-38}$$

which can be integrated with respect to θ:

$$\frac{MR^2}{2}\left(\frac{d\theta}{dt}\right)^2-MgR\cos(\theta)=\text{constant} \tag{22-39}$$

This equation will be used as a level-set equation to visualize pendulum motion.

Buoyant Object Consider a buoyant object that is slightly displaced from its equilibrium floating position. The force (downwards) due to gravity of the buoy is $\rho_{bouy}gV_{bouy}$ The force (upwards) according to Archimedes is $\rho_{water}gV_{sub}$ where V_{sub} is the volume of the buoy that is submerged. The equilibrium position must satisfy $V_{sub-eq}/V_{bouy}=\rho_{bouy}/\rho_{water}$.

If the buoy is slightly perturbed at equilibrium by an amount δx the force is:

$$\begin{aligned}F=&\rho_{water}g(V_{sub-eq}+\delta xA_o)-\rho_{buoy}gV_{buoy}\\F=&\rho_{water}g\delta xA_o\end{aligned} \tag{22-40}$$

where A_o is the cross-sectional area at the equilibrium position. Newton's equation of motion for the buoy is:

$$M_{buoy}\frac{d^2y}{dt^2}-\rho_{water}gA_oy=0 \tag{22-41}$$

so the characteristic frequency of the buoy is $\omega=\sqrt{\rho_{water}gA_o/M_{bouy}}$.

Single Electron Wave-function The one-dimensional Schrödinger equation is:

$$\frac{d^2\psi}{dx^2}+\frac{2m}{\hbar^2}\left(E-U(x)\right)\psi=0 \tag{22-42}$$

where $U(x)$ is the potential energy at a position x. If $U(x)$ is constant as in a free electron in a box, then the one-dimensional wave equation reduces to a simple harmonic oscillator.

In summation, just about any system that oscillates about an equilibrium state can be reduced to a harmonic oscillator.

Index

3.016 Home
Full Screen
Close
Quit

3.016 Home
Full Screen
Close
Quit

3.016 Home
Full Screen
Close
Quit

3.016 Home
Full Screen
Close
Quit